Microsoft

Covers advanced troubleshooting and crash analysis

4

Fourth Edition

Microsoft® WINDOWS® INTERNALS

Microsoft Windows Server 2003, Windows XP, and Windows 2000

Foreword by Jim Allchin

Historical perspective by David N. Cutler

Mark E. Russinovich
David Solomon

PUBLISHED BY
Microsoft Press
A Division of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

Library of Congress Control Number 2005921847

Printed and bound in the United States of America.

1 2 3 4 5 6 7 8 9   QWT   9 8 7 6 5 4

Distributed in Canada by H.B. Fenn and Company Ltd.

A CIP catalogue record for this book is available from the British Library.

Microsoft Press books are available through booksellers and distributors worldwide. For further information about international editions, contact your local Microsoft Corporation office or contact Microsoft Press International directly at fax (425) 936-7329. Visit our Web site at www.microsoft.com/learning/. Send comments to rkpinput@microsoft.com.

The example companies, organizations, products, domain names, e-mail addresses, logos, people, places, and events depicted herein are fictitious. No association with any real company, organization, product, domain name, e-mail address, logo, person, place, or event is intended or should be inferred.

This book expresses the author's views and opinions. The information contained in this book is provided without any express, statutory, or implied warranties. Neither the authors, Microsoft Corporation, nor its resellers, or distributors will be held liable for any damages caused or alleged to be caused either directly or indirectly by this book.

Acquisitions Editors: Robin Van Steenburgh, Ben Ryan
Project Editor: Valerie Woolley
Development Editor: Sally Stickney
Copy Editor: Roger LeBlanc
Indexer: Lynn Armstrong

SubAssy Part No. X11-16607
Body Part No. X11-16608
Section Part No. X11-20530

To Dave Cutler, father of the Windows kernel

Contents at a Glance

1 Concepts and Tools 1

2 System Architecture. 35

3 System Mechanisms. 85

4 Management Mechanisms 183

5 Startup and Shutdown 251

6 Processes, Threads, and Jobs 289

7 Memory Management 375

8 Security 485

9 I/O System. 537

10 Storage Management 615

11 Cache Manager. 655

12 File Systems. 689

13 Networking 787

14 Crash Dump Analysis. 845

Table of Contents

Historical Perspective . . . . . **xix**

Foreword . . . . . **xxiii**

Acknowledgments . . . . . **xxv**

Introduction . . . . . **xxvii**

1 Concepts and Tools 1

- Windows Operating System Versions 1
- Foundation Concepts and Terms 3
  - Windows API 3
  - Services, Functions, and Routines 5
  - Processes, Threads, and Jobs 6
  - Virtual Memory 14
  - Kernel Mode vs. User Mode 16
  - Terminal Services and Multiple Sessions 21
  - Objects and Handles 22
  - Security 23
  - Registry 24
  - Unicode 25
- Digging into Windows Internals 25
  - Performance Tool 27
  - Windows Support Tools 27
  - Windows Resource Kits 27
  - Kernel Debugging 28
  - Platform Software Development Kit (SDK) 33
  - Device Driver Kit (DDK) 34
  - Sysinternals Tools 34
- Conclusion 34

2 System Architecture 35

- Requirements and Design Goals 35
- Operating System Model 36

Architecture Overview. 37
Portability 40
Symmetric Multiprocessing 41
Scalability 46
Differences Between Client and Server Versions 47
Checked Build 49
Key System Components 51
Environment Subsystems and Subsystem DLLs 53
Ntdll.dll 63
Executive 63
Kernel. 65
Hardware Abstraction Layer. 67
Device Drivers. 69
System Processes 75
Conclusion 84

3 System Mechanisms. 85

Trap Dispatching 85
Interrupt Dispatching 87
Exception Dispatching 109
System Service Dispatching 119
Object Manager 124
Executive Objects. 126
Object Structure. 128
Synchronization 149
High-IRQL Synchronization 151
Low-IRQL Synchronization 155
System Worker Threads. 166
Windows Global Flags. 168
Local Procedure Calls (LPCs) 171
Kernel Event Tracing 175
Wow64. 178
Wow64 Process Address Space Layout 179
System Calls 179
Exception Dispatching 179
User Callbacks 179
File System Redirection. 180

Registry Redirection and Reflection. 180
I/O Control Requests 181
16-Bit Installer Applications 182
Printing 182
Restrictions 182
Conclusion. 182

4 Management Mechanisms 183

The Registry. 183
Viewing and Changing the Registry 183
Registry Usage 184
Registry Data Types 185
Registry Logical Structure 186
Troubleshooting Registry Problems. 192
Registry Internals 197
Services. 211
Service Applications. 212
Service Accounts 217
The Service Control Manager. 223
Service Startup 225
Startup Errors 229
Accepting the Boot and Last Known Good 230
Service Failures 231
Service Shutdown. 232
Shared Service Processes. 233
Service Control Programs 236
Windows Management Instrumentation 237
WMI Architecture 237
Providers 239
The Common Information Model and
the Managed Object Format Language 240
The WMI Namespace. 243
Class Association. 244
WMI Implementation 247
WMI Security. 248
Conclusion. 249

5 Startup and Shutdown 251

Boot Process 251

x86 and x64 Preboot 251

The x86/x64 Boot Sector and Ntldr 255

The IA64 Boot Process 264

Initializing the Kernel and Executive Subsystems 266

Smss, Csrss, and Winlogon 269

Images that Start Automatically 273

Troubleshooting Boot and Startup Problems 274

Last Known Good 274

Safe Mode 274

Recovery Console 279

Solving Common Boot Problems 281

Shutdown 286

Conclusion 288

6 Processes, Threads, and Jobs 289

Process Internals 289

Data Structures 289

Kernel Variables 297

Performance Counters 297

Relevant Functions 298

Flow of CreateProcess 300

Stage 1: Opening the Image to Be Executed 302

Stage 2: Creating the Windows Executive Process Object 304

Stage 3: Creating the Initial Thread and Its Stack and Context 308

Stage 4 : Notifying the Windows Subsystem about the New Process 309

Stage 5: Starting Execution of the Initial Thread 310

Stage 6: Performing Process Initialization in the Context of the New Process 310

Thread Internals 313

Data Structures 313

Kernel Variables 320

Performance Counters 321

Relevant Functions 322

Birth of a Thread 322

Examining Thread Activity 323

Thread Scheduling 325
Overview of Windows Scheduling 326
Priority Levels 327
Windows Scheduling APIs. 330
Relevant Tools 331
Real-Time Priorities 333
Thread States. 334
Dispatcher Database 338
Quantum 340
Scheduling Scenarios 345
Context Switching 347
Idle Thread 348
Priority Boosts 348
Multiprocessor Systems 357
Multiprocessor Thread-Scheduling Algorithms. 366
Job Objects 368
Conclusion 373

7 Memory Management 375

Introduction to the Memory Manager 375
Memory Manager Components 376
Internal Synchronization 377
Configuring the Memory Manager. 378
Examining Memory Usage 378
Services the Memory Manager Provides 382
Large and Small Pages. 382
Reserving and Committing Pages 384
Locking Memory 385
Allocation Granularity 385
Shared Memory and Mapped Files 386
Protecting Memory 388
No Execute Page Protection. 390
Copy-on-Write 392
Heap Manager 394
Address Windowing Extensions 399
System Memory Pools 401
Configuring Pool Sizes. 401
Monitoring Pool Usage 404

- Look-Aside Lists . . . 408
- Driver Verifier . . . 409

Virtual Address Space Layouts . . . 413
- x86 User Address Space Layouts . . . 415
- x86 System Address Space Layout . . . 417
- x86 Session Space . . . 418
- System Page Table Entries . . . 421
- 64-Bit Address Space Layouts . . . 422

Address Translation . . . 425
- x86 Virtual Address Translation . . . 425
- Translation Look-Aside Buffer . . . 434
- Physical Address Extension (PAE) . . . 435
- IA-64 Virtual Address Translation . . . 437
- x64 Virtual Address Translation . . . 438

Page Fault Handling . . . 439
- Invalid PTEs . . . 440
- Prototype PTEs . . . 441
- In-Paging I/O . . . 443
- Collided Page Faults . . . 444
- Page Files . . . 444

Virtual Address Descriptors . . . 448

Section Objects . . . 450

Working Sets . . . 457
- Demand Paging . . . 458

Logical Prefetcher . . . 458
- Placement Policy . . . 462
- Working Set Management . . . 463
- Balance Set Manager and Swapper . . . 466
- System Working Set . . . 467

Page Frame Number Database . . . 469
- Page List Dynamics . . . 472
- Modified Page Writer . . . 475
- PFN Data Structures . . . 476
- Low and High Memory Notification . . . 479

Conclusion . . . 483

8 Security 485

Security System Components 488

Protecting Objects 492

Access Checks 493

Security Descriptors and Access Control 506

Account Rights and Privileges 516

Account Rights 517

Privileges 518

Super Privileges 523

Security Auditing 524

Logon 526

Winlogon Initialization 528

User Logon Steps 529

Software Restriction Policies 533

Conclusion 535

9 I/O System 537

I/O System Components 537

The I/O Manager 539

Typical I/O Processing 540

Device Drivers 541

Types of Device Drivers 541

Structure of a Driver 548

Driver Objects and Device Objects 550

Opening Devices 555

I/O Processing 561

Types of I/O 561

I/O Request Packets 564

I/O Request to a Single-Layered Driver 569

I/O Requests to Layered Drivers 577

I/O Completion Ports 585

Driver Verifier 589

The Plug and Play (PnP) Manager 590

Level of Plug and Play Support 591

Driver Support for Plug and Play 592

Driver Loading, Initialization, and Installation 594

Driver Installation 603

The Power Manager 607
Power Manager Operation 609
Driver Power Operation 610
Driver Control of Device Power 613
Conclusion 613
10 Storage Management 615
Storage Terminology 615
Disk Drivers 616
Ntldr. 616
Disk Class, Port, and Miniport Drivers 617
Disk Device Objects 620
Partition Manager 622
Volume Management 622
Basic Disks. 624
Dynamic Disks 626
Multipartition Volume Management 632
The Volume Namespace. 638
Volume I/O Operations. 646
Virtual Disk Service 648
Volume Shadow Copy Service. 649
Conclusion 654
11 Cache Manager. 655
Key Features of the Cache Manager. 655
Single, Centralized System Cache 656
The Memory Manager 656
Cache Coherency. 656
Virtual Block Caching 658
Stream-Based Caching 658
Recoverable File System Support 658
Cache Virtual Memory Management. 660
Cache Size. 662
LargeSystemCache. 662
Cache Virtual Size 663
Cache Working Set Size 665
Cache Physical Size 667

Cache Data Structures 668
Systemwide Cache Data Structures 669
Per-File Cache Data Structures 670
File System Interfaces 674
Copying to and from the Cache 676
Caching with the Mapping and Pinning Interfaces 677
Caching with the Direct Memory Access Interfaces 678
Fast I/O 679
Read Ahead and Write Behind 682
Intelligent Read-Ahead 682
Write-Back Caching and Lazy Writing 683
Write Throttling 686
System Threads 687
Conclusion 688

12 File Systems 689

Windows File System Formats 690
CDFS 690
UDF 691
FAT12, FAT16, and FAT32 691
NTFS 694
File System Driver Architecture 694
Local FSDs 695
Remote FSDs 696
File System Operation 700
File System Filter Drivers 705
Troubleshooting File System Problems 711
Filemon Basic vs. Advanced Modes 711
Filemon Troubleshooting Techniques 712
NTFS Design Goals and Features 717
High-End File System Requirements 717
Advanced Features of NTFS 719
NTFS File System Driver 729
NTFS On-Disk Structure 732
Volumes 732
Clusters 732
Master File Table 733

File Reference Numbers 739
File Records. 740
Filenames 742
Resident and Nonresident Attributes 744
Data Compression and Sparse Files 747
The Change Journal File 752
Indexing 753
Object IDs 754
Quota Tracking. 755
Consolidated Security 756
Reparse Points 758
NTFS Recovery Support 758
Evolution of File System Design 759
Logging 761
Recovery 767
NTFS Bad-Cluster Recovery 771
Encrypting File System Security. 775
Encrypting a File for the First Time. 778
The Decryption Process 783
Backing Up Encrypted Files 784
Conclusion 785

13 Networking. 787

Windows Networking Architecture. 787
The OSI Reference Model. 787
Windows Networking Components 789
Networking APIs 791
Windows Sockets. 791
Remote Procedure Call 798
Web Access APIs 803
Named Pipes and Mailslots 804
NetBIOS. 811
Other Networking APIs. 813
Multiple Redirector Support. 815
Multiple Provider Router 816
Multiple UNC Provider 818

Name Resolution 820

Domain Name System 820

Windows Internet Name Service 820

Protocol Drivers 821

TCP/IP Extensions 824

NDIS Drivers 828

Variations on the NDIS Miniport 832

Connection-Oriented NDIS. 832

Remote NDIS. 835

QOS 836

Binding 838

Layered Network Services 839

Remote Access 839

Active Directory 840

Network Load Balancing 841

File Replication Service 843

Distributed File System 843

Conclusion 844

14 Crash Dump Analysis 845

Why Does Windows Crash? 845

The Blue Screen 846

Crash Dump Files 849

Crash Dump Generation 852

Windows Error Reporting 853

Online Crash Analysis. 854

Basic Crash Dump Analysis 855

Notmyfault 855

Basic Crash Dump Analysis 856

Verbose Analysis 858

Using Crash Troubleshooting Tools 860

Buffer Overrun and Special Pool 861

Code Overwrite and System Code Write Protection 863

Advanced Crash Dump Analysis. 864

Stack Trashes 865

Hung or Unresponsive Systems 866

When There Is No Crash Dump 869

Glossary 871

Index. 895

Historical Perspective

Again I find myself indebted to David Solomon and Mark Russinovich for providing the opportunity to write a few words about their newest edition to a series of books on Windows Internals. It has been over three years since the last publication in this series, and this passage of time has seen two major releases: a very significant update to the client system, and another very significant update to the server system, currently being readied for shipment.

Two of the growing problems faced by the authors of a book such as this are tracing the implementation evolution of the Microsoft Windows NT system and documenting the way in which feature implementation has changed in each version. To this end, the authors have done a remarkable job of providing examples and explanations throughout the book.

(Left to right) David Solomon, David Cutler, and Mark Russinovich

I first met David Solomon when I was working at Digital Equipment Corporation on the VMS operating system for VAX, and he was only 16. Since that time, he has been involved with operating system development and teaching operating system internals. I met Mark Russinovich more recently but have been aware of his expertise in the area of operating systems for some time. He has done some amazing work, such as his NTFS file system running on Microsoft Windows 98 and his “live” Windows kernel debugger, which can be used to peer into the Windows system while it is running.

The beginnings of Windows NT started in October 1988 with a set of goals to produce a portable system that addressed OS/2 compatibility, security, POSIX, multiprocessing, integrated networking, and reliability. With the advent and huge success of Windows 3.0, the system goals were soon changed to natively address Windows compatibility directly and move OS/2 compatibility to a subsystem.

We originally thought we could produce the first Windows NT system in a little over two years. It actually ended up taking us four and a half years for the first release in the summer of 1993, and that release supported the Intel i386, Intel i486, and the MIPS R4000 processors. Six weeks later, we also introduced support for the Digital Alpha processors.

The first release of Windows NT was larger and slower than expected, so the next major push was a project called Daytona, named after the speedway in Florida. The main goals for this release were to reduce the size of the system, increase the speed of the system, and, of course, to make it more reliable. Six months after the release of Windows NT 3.5 in the fall of 1994, we released Windows NT 3.51, an updated version containing support for the IBM PowerPC processor.

The goal for the next version of Windows NT was to update the user interface to be compatible with Windows 95 and to incorporate the Cairo technologies that had been under development at Microsoft for a couple of years. This system took two more years to develop and was introduced in the summer of 1996 as Windows NT 4.0.

The following version of NT saw a name change to Windows 2000 and was the last system for which the client and server systems were released at the same time. This version was built on the same Windows NT technology as the previous versions and introduced significant new features such as Active Directory. Windows 2000 took three and a half years to produce and was the most tested and tuned version of Windows NT technology produced at the time. Windows 2000 was the culmination of over eleven years of development spanning implementations on four architectures.

At the end of Windows 2000 development, we embarked on an ambitious plan to implement new versions of the client and server systems, which would include new enhanced consumer features and improved server capabilities. As plans developed, it became clear that implementation of the server features would cause a lag in the implementation of the client features, and therefore, the releases were split. In August of 2001, Windows XP Professional and Windows XP Home Edition were released, and a little over a year later, in March of 2003, Microsoft Windows Server 2003 was released. In addition to the Intel x86 architecture, these systems contained support for the Intel IA-64, marking Windows NT's first move to 64-bit processing.

This book is the definitive work on the internal structures and workings of Windows XP and Windows Server 2003. In addition, it offers a glimpse into the future of Windows' move to 64-bit computing by covering AMD's introduction of the x64 architecture (AMD64) in 2003 and Intel's announced support (EM64T) in February 2004. A fully supported x64 client and server release is planned in the first half of 2005, and this book contains many insights into the implementation details of the x64 system.

The x64 architecture is the beginning of a new era for Windows NT at a time when the x86 architecture is beginning to show signs of old age. This architecture offers 32-bit x86 compatibility at speed to protect legacy software investments, and provides 64-bit addressing capability to address the most ambitious of new applications. This will protect 32-bit software investments while providing Windows NT with a breath of new life well into the next decade and beyond.

Although the Windows NT system has undergone several name changes over the past several years, it remains entirely based on the original Windows NT code base. As time has marched on and invention has thrived, the implementation of many internal features has changed significantly. The authors have done a laudable job of assimilating the details of the Windows NT code base and its differing implementations from release to release and platform to platform, and of producing examples and tools that help the reader understand how things work. Every serious operating system developer should have a copy of this book on his or her desk.

David N. Cutler
Senior Distinguished Engineer
Microsoft Corporation

Foreword

Microsoft Windows has been a core part of my life for 14 years. During this time, over release after release, the operating system has evolved in breadth and depth. Producing Windows is one of the most important and complex projects in the world today. Easily 5,000 engineers work on Windows releases. Across virtually all cultures, Windows users comprise an entire spectrum from the largest mission-critical business to theyoungest child. Customers using Windows demand constant improvements in virtually every aspect—from being able to run the largest servers to being easy enough for a pre-schooler to use. Windows comes in many shapes and sizes, from embedded editions to media center editions to data center editions. All these products use the same core Windows internals, which evolve and improve with each release.

This is the definitive book on the core Windows internals. If you want to learn how Windows works internally, in the fastest way possible, then this is the book for you. Understanding all the pieces of such a large product is a daunting task. But if you start at the core concepts of the system and work out, the puzzle fits together a lot more easily. Just as Windows itself has evolved, so has the comprehensive nature of this book, now in its fourth edition. For years, we have used earlier editions of this material to train brand-new employees at Microsoft, so this material is tried and true.

If you're like me, you like to figure out how things really work. Reading "how to use" books or "tips and tricks" has never been sufficient for me. If you understand how something works internally, you know how to better use it, maximize performance and security, diagnose failures, and frankly, have more fun. If you're like me and want to see Windows from the "inside out," then you're starting in the right place.

David and Mark have done an outstanding job detailing the "inside" Windows technical story. The tools that they highlight are a great resource for direct hands-on training and diagnostics work. After you read this book, you'll have a much greater understanding of how the operating system fits together, the latest improvements made throughout the system, and how to get the most from these improvements.

It has been quite a journey—one that is still underway. So, start reading and dive deep into one of the most impressive operating systems ever created.

Jim Allchin
Group Vice President, Platforms
Microsoft Corporation

Acknowledgments

First, special thanks to the following people:

- **Dave Cutler,** Senior Distinguished Engineer and the original architect of Microsoft Windows NT. Dave originally approved David Solomon's source code access and has been supportive of his work to explain the internals of Windows NT through his training business as well as during the writing of Inside Windows NT, Second Edition, and Inside Microsoft Windows 2000, Third Edition. Besides reviewing the chapter on processes and threads, Dave answered many questions on the kernel architecture of the system and wrote a historical perspective for this edition.
- **Jim Allchin**, our executive sponsor, for writing the Foreword to this book and championing our cause within Microsoft.
- **Rob Short**, vice president, who made sure we had the resources we needed, as well as access to the relevant people.

We also thank two developers in the Windows division for writing new content that was incorporated into this edition:

- **Adrian Marinescu,** who wrote the greatly expanded heap manager section in the memory management chapter.
- **Samer Arafeh,** who wrote the description of Wow64.

Thanks to our friend Jeffrey Richter, for writing the "What about .NET and WinFX" sidebar in Chapter 1 and for continuing to remind us over many dinners together of his view on how few people should care about what we talk about in this book.

This book wouldn't contain the depth of technical detail or the level of accuracy it has without the review, input, and support of key members of the Microsoft Windows development team. Therefore, we want to thank the following people, who provided technical review and input to the book:

Murali Brahmadesam
Molly Brown
Duncan Bryce
Daniel Bucherer
Neal Christian
Neill Clift
Mike Danseglio
Joseph Davies
Cenk Ergan
Pat Hoffer
Anthony Jones
Tom Jones
Joseph Joy
Shreeniwas Kelkar
Connie La Chasse
Mike Lai
Paul Leach
Gerald Maffeo
Daniel Pravat
Dragos Sambotin
Jon Schwartz
Rob Short
Paul Sliwowicz
Chittur Subbaraman
Cristian Teodorescu
Andre Vachon
Landy Wang

Tom Fout
Nar Ganapathy
David Golds
Robert Gu
Jeff Hamblin
Aaron Margosis
Iain McDonald
Kamen Moutafov
Adi Oltean
Vince Orgovan
Richard Ward
Brad Waters
Bruce Worthington
Mark Zbikowsk
Khawar Zuberi

Others might have contributed by answering questions in the hallway or cafeteria and providing technical material—if we missed you, please forgive us!

Thanks also to Jamie Hanrahan of Azius Developer Training (www.azius.com), who coauthored with David the original Windows Internal Architecture class on which the second edition was based. Jamie, who has a real knack for explaining complicated concepts in a simple and practical fashion, developed several of the explanations, diagrams, and figures.

Thanks to Dave Probert for hosting the share where review drafts were distributed to internal Microsoft reviewers.

And thanks to Jonathan Sloves of AMD for arranging AMD64 test systems to be sent to us to help us develop the 64-bit content and port some of the Sysinternals tools to x64.

Finally, we want to thank the following people from Microsoft Press for their contribution to this book:

- Robin van Steenburgh, acquisitions editor, for patiently working with us to complete this project.
- Sally Stickney, who, for a time, continued as our project editor, but later got drawn into the management vortex. We missed working with you this time!
- Valerie Woolley, who took over as project editor from Sally. You were great (and not as rough on us as Sally was for the last two editions)!
- Roger LeBlanc, who laboriously went through all our chapters to tighten up text, find inconsistencies, and in general bring the manuscript up to the high standards of Microsoft Press.

David Solomon and Mark Russinovich
September, 2004

Introduction

Microsoft Windows Internals, Fourth Edition is intended for advanced computer professionals (both developers and system administrators) who want to understand how the core components of the Microsoft Windows 2000, Windows XP, and Microsoft Windows Server 2003 operating systems work internally. With this knowledge, developers can better comprehend the rationale behind design choices when building applications specific to the Windows platform. Such knowledge can also help developers debug complex problems. System administrators can benefit from this information as well, because understanding how the operating system works "under the covers" facilitates understanding the performance behavior of the system and makes it easier to troubleshoot system problems when things go wrong. After reading this book, you should have a better understanding of how Windows works and why it behaves as it does.

Structure of the Book

The first two chapters ("Concepts and Tools" and "System Architecture") lay the foundation with terms and concepts used throughout the rest of the book. The next three chapters—"System Mechanisms," "Management Mechanisms," and "Startup and Shutdown"—describe key underlying mechanisms in the system. The next eight chapters explain the core components of the operating system: processes, threads, and jobs; memory management; security; the I/O system; storage management; the cache manager; file systems; and networking. The last chapter covers crash dump analysis.

History of the Book

This is the fourth edition of a book that was originally called *Inside Windows NT* (Microsoft Press, 1992), written by Helen Custer (prior to the initial release of Microsoft Windows NT 3.1). *Inside Windows NT* was the first book ever published about Windows NT and provided key insight into the architecture and design of the system. *Inside Windows NT, Second Edition* (Microsoft Press, 1998) was written by David Solomon. It was updated to cover Windows NT 4.0 and had a greatly increased level of technical depth. *Inside Microsoft Windows 2000, Third Edition* (Microsoft Press, 2000) was authored by David Solomon and Mark Russinovich. It added many new topics such as startup and shutdown, service internals, registry internals, file system drivers, and networking, as well as kernel changes in Windows 2000 such as the Windows Driver Model (WDM), Plug and Play, power management, Windows Management Instrumentation (WMI), encryption, the job object, and Terminal Services.

Fourth Edition Changes

This latest edition, now called *Microsoft Windows Internals, Fourth Edition*, has been updated to cover the kernel changes made in Windows XP and Windows Server 2003, including support for 64-bit systems. Hands-on experiments have been updated to reflect changes in tools, and newly added experiments use new tools not available when the third edition was written.

Since the level of kernel change from Windows 2000 to these versions was relatively small (as compared to the changes between Windows NT 4.0 and Windows 2000), the vast majority of this text applies to Windows 2000, Windows XP, and Windows Server 2003. Therefore, unless explicitly stated, everything applies to all three versions.

Hands-On Experiments

Even without access to the source code, much can be gleaned about Windows internals from available tools such as the kernel debugger. When a tool can be used to expose or demonstrate some aspect of Windows internal behavior, the steps for trying the tool yourself are listed in "Experiment" boxes. These appear throughout the book, and we encourage you to try these as you're reading—seeing visible proof of how Windows works internally will make much more of an impression on you than just reading about it will.

Topics Not Covered

Windows is a large and complex operating system. This book doesn't cover everything relevant to Windows internals but instead focuses on the base system components. For example, this book doesn't describe COM+, the Windows distributed object-oriented programming infrastructure, or the Microsoft .NET Framework, the foundation of the next generation of managed code applications.

Because this is an internals book and not a user, programming, or system administration book, it doesn't describe how to use, program, or configure Windows.

A Warning and Caveat

Because this book describes undocumented behavior of the internal architecture and operation of the Windows operating system (such as internal kernel structures and functions), this content is subject to change between releases. (External interfaces, such as the Windows API, are not subject to incompatible changes.)

By "subject to change," we don't necessarily mean that details described in this book *will* change between releases, but you can't count on them not changing. Any software that uses these undocumented interfaces might not work on future releases of Windows. Even worse, software that runs in kernel mode (such as device drivers) and uses these undocumented interfaces might experience a system crash when running on a newer release of Windows.

Support

Every effort has been made to ensure the accuracy of this book. Should you run into any problems or issues, please refer to the sources listed below.

From the Authors

This book isn't perfect. No doubt it contains some inaccuracies, or possibly, we've omitted some topics we should have covered. If you find anything you think is incorrect, or if you believe we should have included material that isn't here, please feel free to send e-mail to *windowsinternals@sysinternals.com*. Updates and corrections will be posted on the page *www.sysinternals.com/windowsinternals*.

From Microsoft Press

Microsoft also provides corrections for books through the World Wide Web at the following address:

http://www.microsoft.com/learning/support

To connect directly with the Microsoft Learning Knowledge Base and enter a query regarding an issue you might have encountered, go to *http://www.microsoft.com/learning/support/search.asp*.

In addition to sending feedback directly to the authors, if you have comments, questions, or ideas regarding the presentation or use of this book, you can send them to Microsoft using either of the following methods:

Postal Mail:
Microsoft Press
Attn: *Windows Internals* Editor
One Microsoft Way
Redmond, WA 98052-6399

E-mail:
mspinput@microsoft.com

Please note that product support isn't offered through the above mail addresses. For support information regarding Microsoft Windows, go to *www.microsoft.com/windows*. You can also call Standard Support at (425) 635-7011 weekdays between 6 a.m. and 6 p.m. Pacific time, or you can search Microsoft's Support Online at *support.microsoft.com/support*.

System Requirements

To use the *Microsoft Windows Server 2003 Resource Kit* tools, eBooks, and other materials, you need to meet the following minimum system requirements:

- Microsoft Windows Server 2003 or Windows XP operating system
- PC with 233-megahertz (MHz) or higher processor; 550-MHz or higher processor is recommended
- 128 MB of RAM; 256 MB or higher is recommended
- 1.5 to 2 GB of available hard disk space
- Super VGA (800 x 600) or higher resolution video adapter and monitor
- CD or DVD drive
- Keyboard and Microsoft mouse or compatible pointing device
- Adobe Acrobat or Adobe Reader
- Internet connectivity for tools that are downloaded

> **Note** Resource Kit tools are written and tested in English only. Using these tools with a non-English version of Windows might produce unpredictable results. Resource Kit tools are not supported on 64-bit platforms.

An evaluation edition for Windows Server 2003 Enterprise Edition with Service Pack 1 will be available on release of Service Pack 1. You can download the evaluation software from the Microsoft Download Center at *http://www.microsoft.com/downloads/*. (Availability of software on the Download Center is at the discretion of Microsoft Corporation and is subject to change.) To use the evaluation, you need

- 133-MHz or higher processor; 733-MHz or higher processor is recommended for x86-based PCs and Itanium-based PCs.
- 128 MB of RAM; 256 MB of RAM is recommended; 32 GB is recommended for x86-based PCs (32-bit version) and 64 GB is recommended for Itanium-based PCs (64-bit version).
- 1.5 to 2 GB of available hard disk space.
- Super VGA (500 x 600) or higher resolution video adapter and monitor.
- Keyboard and Microsoft mouse or compatible pointing device.

> **Note** Actual requirements, including Internet and network access and any related charges, will vary based on your system configuration and the applications and features you choose to install. Additional hard disk space may be required if you are installing over a network.

Chapter 1

Concepts and Tools

In this chapter, we'll introduce the key Microsoft Windows operating system concepts and terms we'll be using throughout this book, such as the Windows API, processes, threads, virtual memory, kernel mode and user mode, objects, handles, security, and the registry. We'll also introduce the tools that you can use to explore Windows internals, such as the kernel debugger, the Performance tool, and key tools from *www.sysinternals.com*. In addition, we'll explain how you can use the Windows Device Driver Kit (DDK) and Platform Software Development Kit (SDK) as resources for finding further information on Windows internals.

Be sure that you understand everything in this chapter—the remainder of the book is written assuming that you do.

Windows Operating System Versions

This book covers the three most recent versions of the Microsoft Windows operating system based on the Windows NT code base: Windows 2000, Windows XP (32-bit and 64-bit versions), and Windows Server 2003 (32-bit and 64-bit versions). Unless specifically stated, the text applies to all three versions. As background information, Table 1-1 lists the releases of the Windows NT code base, their internal version number, and the external product name.

Table 1-1 Windows Operating System Releases

Product Name	Internal Version Number	Release Date
Windows NT 3.1	3.1	July 1993
Windows NT 3.5	3.5	September 1994
Windows NT 3.51	3.51	May 1995
Windows NT 4.0	4.0	July 1996
Windows 2000	5.0	December 1999
Windows XP	5.1	August 2001
Windows Server 2003	5.2	March 2003

Windows NT vs. Windows 95

From the initial announcement of Windows NT, Microsoft made it clear that it was to be the long-term replacement for Windows 95 (and its subsequent releases, Windows 98 and Windows Millennium Edition). The following list highlights some architectural differences and advantages that Windows NT (and its subsequent releases) has over Windows 95 (and its subsequent releases):

- Windows NT supports multiprocessor systems—Windows 95 doesn't.
- The Windows NT file system supports security (such as discretionary access control). The Windows 95 file system doesn't.
- Windows NT is fully a 32-bit (and now 64-bit) operating system—it contains no 16-bit code, other than support code for running 16-bit Windows applications. Windows 95 contains a large amount of old 16-bit code from its predecessors, Windows 3.1 and MS-DOS.
- Windows NT is fully reentrant—significant parts of Windows 95 are nonreentrant (mainly the 16-bit code taken from Windows 3.1). This nonreentrant code includes the majority of the graphics and window management functions (GDI and USER). When a 32-bit application on Windows 95 attempts to call a system service implemented in nonreentrant 16-bit code, the application must first obtain a system-wide lock (or mutex) to block other threads from entering the nonreentrant code base. And even worse, a 16-bit application holds this lock *while running*. As a result, although the core of Windows 95 contains a preemptive 32-bit multithreaded scheduler, applications often run single threaded because so much of the system is still implemented in nonreentrant code.
- Windows NT provides an option to run 16-bit Windows applications in their own address space—Windows 95 always runs 16-bit Windows applications in a shared address space, in which they can corrupt (and hang) each other.
- Process shared memory on Windows NT is visible only to the processes that are mapping the same shared memory section. On Windows 95, all shared memory is visible and writable from all processes. Thus, any process can write to and corrupt shared memory being used by other cooperating processes.
- Windows 95 has some critical operating system pages that are writable from user mode, thus allowing a user application to corrupt or crash the system.

The one thing Windows 95 can do that Windows NT–based systems will never do is run *all* older MS-DOS and Windows 3.1 applications (notably applications that require direct hardware access) as well as 16-bit MS-DOS device drivers. Whereas 100 percent compatibility with MS-DOS and Windows 3.1 was a mandatory goal for Windows 95, the original goal for Windows NT was to run *most* existing 16-bit applications while preserving the integrity and reliability of the system.

Foundation Concepts and Terms

In the course of this book, we'll be referring to some structures and concepts that might be unfamiliar to some readers. In this section, we'll define the terms we'll be using throughout. You should become familiar with them before proceeding to subsequent chapters.

Windows API

The Windows application programming interface (API) is the system programming interface to the Microsoft Windows operating system family, including Windows 2000, Windows XP, Windows Server 2003, Windows 95, Windows 98, Windows Millennium Edition (Me), and Windows CE. Each operating system implements a different subset of the Windows API. Windows 95, Windows 98, Windows Me, and Windows CE are not addressed in this book.

Note The Windows API is described in the Platform Software Development Kit (SDK) documentation. (See the section "Platform Software Development Kit (SDK)" later in this chapter.) This documentation is available for free viewing online at *msdn.microsoft.com*. It is also included with all subscription levels to the Microsoft Developer Network (MSDN), Microsoft's support program for developers. For more information, see *msdn.microsoft.com*. An excellent description of how to program the Windows base API is Jeffrey Richter's book *Programming Applications for Microsoft Windows* (4th ed., Microsoft Press, 1999).

Prior to the introduction of 64-bit versions of Windows XP and Windows Server 2003, the programming interface to the 32-bit version of the Windows operating systems was called the Win32 API, to distinguish it from the original 16-bit Windows API, which was the programming interface to the original 16-bit versions of Windows. In this book, the term *Windows API* refers to the 32-bit interface to Windows 2000 and both the 32-bit and 64-bit programming interfaces to Windows XP and Windows Server 2003.

The Windows API consists of thousands of callable functions, which are divided into the following major categories:

- Base Services
- Component Services
- User Interface Services
- Graphics and Multimedia Services
- Messaging and Collaboration
- Networking
- Web Services

This book focuses on the internals of the key base services, such as processes and threads, memory management, I/O, and security.

What About .NET and WinFX?

The .NET Framework consists of a library of classes called the Framework Class Library (FCL) and a Common Language Runtime (CLR) that provides a managed code execution environment with features such as just-in-time compilation, type verification, garbage collection, and code access security. By offering these features, the CLR provides a development environment that improves programmer productivity and reduces common programming errors. (For an excellent description of the .NET Framework and its core architecture, see *Applied Microsoft .NET Framework Programming* by Jeffrey Richter.)

The CLR is implemented as a classic COM server whose code resides in a standard user-mode Windows DLL. In fact, all components of the .NET Framework are implemented as standard user-mode Windows DLLs layered over unmanaged Windows API functions. (None of the .NET Framework runs in kernel mode.) Figure 1-1 illustrates the relationship of these components:

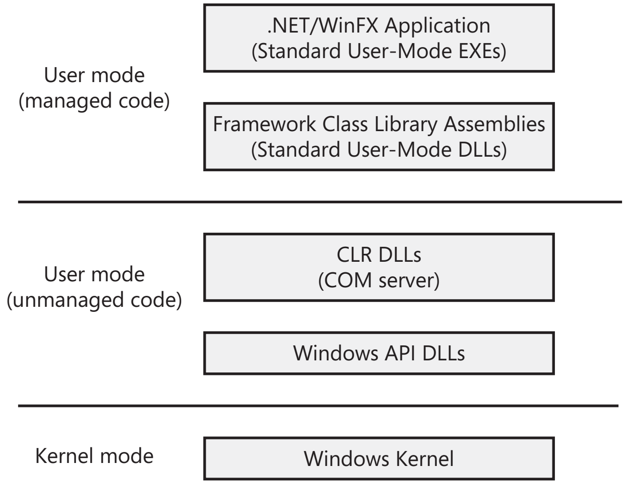


Figure 1-1 Relationship of .NET Framework components

WinFX is "the new Windows API." It is the evolution of the .NET Framework that ships with Windows "Longhorn," the next major release of Windows. It will also be installable on Windows XP and Windows Server 2003. WinFX provides the foundation for the next generation of applications built for the Windows operating system.

History of the Win32 API

Interestingly, Win32 wasn't slated to be the original programming interface to Microsoft Windows NT. Because the Windows NT project started as a replacement for OS/2 version 2, the primary programming interface was the 32-bit OS/2 Presentation Manager API. A year into the project, however, Microsoft Windows 3.0 hit the market and took off. As a result, Microsoft changed direction and made Windows NT the future replacement for the Windows family of products as opposed to the replacement for OS/2. It was at this juncture that the need to specify the Windows API arose—before this, the Windows API existed only as a 16-bit interface.

Although the Windows API would introduce many new functions that hadn't been available on Windows 3.1, Microsoft decided to make the new API compatible with the 16-bit Windows API function names, semantics, and use of data types whenever possible to ease the burden of porting existing 16-bit Windows applications to Windows NT. So those of you who are looking at the Windows API for the first time and wondering why many function names and interfaces seem inconsistent should keep in mind that one reason for the inconsistency was to ensure that the Windows API is compatible with the old 16-bit Windows API.

Services, Functions, and Routines

Several terms in the Windows user and programming documentation have different meanings in different contexts. For example, the word *service* can refer to a callable routine in the operating system, a device driver, or a server process. The following list describes what certain terms mean in this book:

- **Windows API functions** Documented, callable subroutines in the Windows API. Examples include *CreateProcess*, *CreateFile*, and *GetMessage*.
- **Native system services (or executive system services)** The undocumented, underlying services in the operating system that are callable from user mode. For example, *NtCreateProcess* is the internal system service the Windows *CreateProcess* function calls to create a new process. (For a definition of native functions, see the section "System Service Dispatching" in Chapter 3.)
- **Kernel support functions (or routines)** Subroutines inside the Windows operating system that can be called only from kernel mode (defined later in this chapter). For example, *ExAllocatePool* is the routine that device drivers call to allocate memory from the Windows system heaps.

- **Windows services** Processes started by the Windows service control manager. (Although the registry defines Windows device drivers as "services," we don't refer to them as such in this book.) For example, the Task Scheduler service runs in a user-mode process that supports the *at* command (which is similar to the UNIX commands *at* or *cron*).
- **DLL (dynamic-link library)** A set of callable subroutines linked together as a binary file that can be dynamically loaded by applications that use the subroutines. Examples include Msvcrt.dll (the C run-time library) and Kernel32.dll (one of the Windows API subsystem libraries). Windows user-mode components and applications use DLLs extensively. The advantage DLLs provide over static libraries is that applications can share DLLs, and Windows ensures that there is only one in-memory copy of a DLL's code among the applications that are referencing it.

Processes, Threads, and Jobs

Although programs and processes appear similar on the surface, they are fundamentally different. A *program* is a static sequence of instructions, whereas a *process* is a container for a set of resources used when executing the instance of the program. At the highest level of abstraction, a Windows process comprises the following:

- A *private virtual address space*, which is a set of virtual memory addresses that the process can use
- An executable program, which defines initial code and data and is mapped into the process's virtual address space
- A list of open handles to various system resources, such as semaphores, communication ports, and files, that are accessible to all threads in the process
- A security context called an *access token* that identifies the user, security groups, and privileges associated with the process
- A unique identifier called a *process ID* (internally called a *client ID*)
- At least one thread of execution

Each process also points to its parent or creator process. However, if the parent exits, this information is not updated. Therefore, it is possible for a process to point to a nonexistent parent. This is not a problem, as nothing relies on this information being present. The following experiment illustrates this case.

EXPERIMENT: Viewing the Process Tree

One unique attribute about a process that most tools don’t display is the parent or creator process ID. You can retrieve this value with the Performance tool (or programmatically) by querying the Creating Process ID. The Tlist.exe tool (in the Windows Debugging Tools) can show the process tree by using */t* switch. Here’s an example of output from *tlist /t*:

```
C:\>tlist /t
System Process (0)
System (2)
  smss.exe (21)
    csrss.exe (24)
    winlogon.exe (35)
      services.exe (41)
        spoolss.exe (69)
        llssrv.exe (94)
        LOCATOR.EXE (96)
        RpcSs.exe (112)
        inetinfo.exe (128)
      lsass.exe (44)
   nddeagnt.exe (119)
explorer.exe (123) Program Manager
  OSA.EXE (121)
  WINWORD.EXE (117) Microsoft Word - msch02(s).doc
  cmd.exe (72) Command Prompt - tlist /t
    tlist.EXE (100)
```

The list indents each process to show its parent/child relationship. Processes whose parents aren’t alive are left-justified (as is Explorer.exe in the preceding example) because even if a grandparent process exists, there’s no way to find that relationship. Windows maintains only the creator process ID, not a link back to the creator of the creator, and so forth.

To demonstrate the fact that Windows doesn’t keep track of more than just the parent process ID, follow these steps:

1. Open a Command Prompt window.
2. Type **start cmd** (which starts a second Command Prompt).
3. Bring up Task Manager.
4. Switch to the second Command Prompt.
5. Type **mspaint** (which runs Microsoft Paint).

6. Click the intermediate (second) Command Prompt window.
7. Type **exit**. (Notice that Paint remains.)
8. Switch to Task Manager.
9. Click the Applications tab.
10. Right-click on the Command Prompt task, and select Go To Process.
11. Click on the Cmd.exe process highlighted in gray.
12. Right-click on this process, and select End Process Tree.
13. Click Yes in the Task Manager Warning message box.

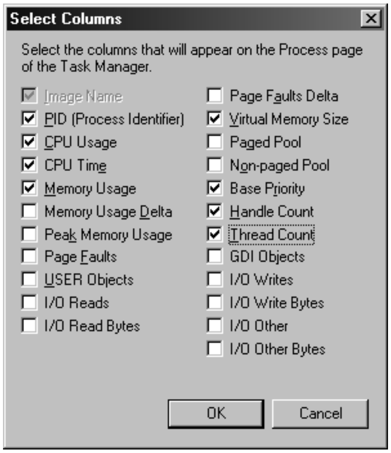
The first Command Prompt window will disappear, but you should still see the Paintbrush window because it was the grandchild of the Command Prompt process you terminated; and because the intermediate process (the parent of Paintbrush) was terminated, there was no link between the parent and the grandchild.

A number of tools for viewing (and modifying) processes and process information are available. The following experiments illustrate the various views of process information you can obtain with some of these tools. These tools are included within Windows itself and within the Windows Support Tools, Windows Debugging Tools, Windows resource kits, the Platform SDK, and from *www.sysinternals.com*. Many of these tools show overlapping subsets of the core process and thread information, sometimes identified by different names.

Probably the most widely used tool to examine process activity is Task Manager. (Interestingly, there is no such thing as a "task" in the Windows kernel, so Task Manager is really a tool to manage processes.) The following experiment shows the difference between what Task Manager lists as applications and processes.

EXPERIMENT: Viewing Process Information with Task Manager

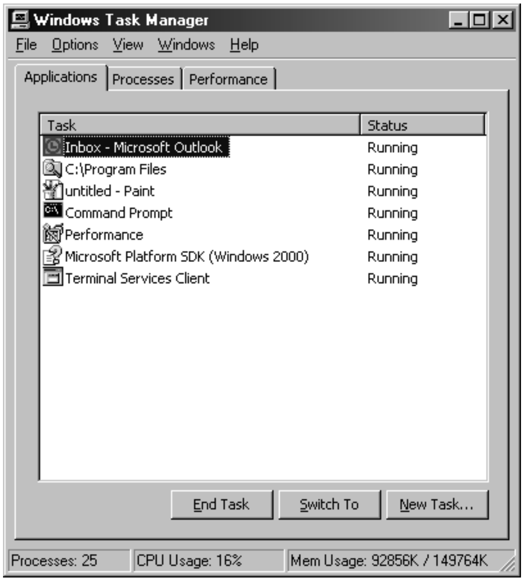
The built-in Windows Task Manager provides a quick list of the processes running on the system. You can start Task Manager in one of three ways: (1) press Ctrl+Shift+Esc, (2) right-click on the taskbar and select Task Manager, or (3) press Ctrl+Alt+Delete and click the Task Manager button. Once Task Manager has started, click the Processes tab to see the list of running processes. Notice that processes are identified by the name of the image of which they are an instance. Unlike some objects in Windows, processes can't be given global names. To display additional details, choose Select Columns from the View menu and select additional columns to be added, as shown here:


Select Columns

Select the columns that will appear on the Process page of the Task Manager.

- ☑ Image Name
- ☑ PID (Process Identifier)
- ☑ CPU Usage
- ☑ CPU Time
- ☑ Memory Usage
- ☐ Memory Usage Delta
- ☐ Peak Memory Usage
- ☐ Page Faults
- ☐ USER Objects
- ☐ I/O Reads
- ☐ I/O Read Bytes
- ☐ Page Faults Delta
- ☑ Virtual Memory Size
- ☐ Paged Pool
- ☐ Non-paged Pool
- ☑ Base Priority
- ☑ Handle Count
- ☑ Thread Count
- ☐ GDI Objects
- ☐ I/O Writes
- ☐ I/O Write Bytes
- ☐ I/O Other
- ☐ I/O Other Bytes

Although what you see in the Task Manager Processes tab is clearly a list of processes, what the Applications tab displays isn't as obvious. The Applications tab lists the top-level visible windows on all the desktops in the interactive window station. (By default, there are two desktop objects–you can create more by using the Windows *CreateDesktop* function.) The Status column indicates whether or not the thread that owns the window is in a Windows message wait state. "Running" means the thread is waiting for windowing input; "Not Responding" means the thread isn't waiting for windowing input (for example, the thread might be running or waiting for I/O or some Windows synchronization object).


Windows Task Manager

Task	Status
Inbox - Microsoft Outlook	Running
C:\Program Files	Running
untitled - Paint	Running
Command Prompt	Running
Performance	Running
Microsoft Platform SDK (Windows 2000)	Running
Terminal Services Client	Running

Processes: 25 | CPU Usage: 16% | Mem Usage: 92856K / 149764K

From the Applications tab, you can match a task to the process that owns the thread that owns the task window by right-clicking on the task name and choosing Go To Process.

Process Explorer, from *www.sysinternals.com*, shows more details about processes and threads than any other available tool, which is why you will see it used in a number of experiments throughout the book. The following are some of the unique things that Process Explorer shows or enables:

- Full path name for the image being executed
- Process security token (list of groups and privileges)
- Highlighting to show changes in the process and thread list
- List of services inside service-hosting processes, including display name and description
- Processes that are part of a job and job details
- Processes running .NET/WinFX applications and.NET-specific details (such as the list of appdomains and CLR performance counters)
- Start time for processes and threads
- Complete list of memory mapped files (not just DLLs)
- Ability to suspend a process
- Ability to kill an individual thread
- Easy identification of which processes were consuming the most CPU time over a period of time (The Performance Tool can display process CPU utilization for a given set of processes, but it won't automatically show processes created after the performance monitoring session has started.)

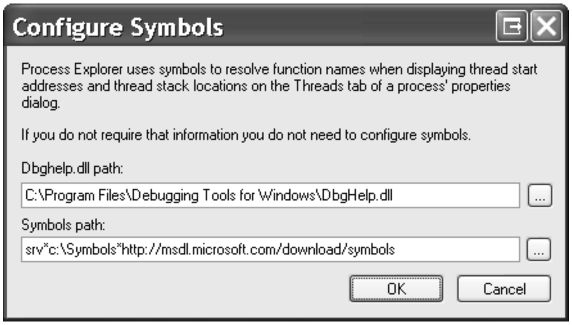
Process Explorer also provides easy access to information available through other tools from one central place, such as:

- Process tree (with ability to collapse parts of the tree)
- Open handles in a process without prior setup (The Microsoft tools to show open handles require the setting of a systemwide flag and a reboot before they can be used.)
- List of DLLs (and memory-mapped files) in a process
- Thread activity within a process
- User-mode thread stacks (including mapping of addresses to names using the debugging tools' symbol engine)
- Kernel-mode thread stacks for system threads (including mapping of addresses to names using the debugging tools' symbol engine)
- Context switch delta (a better representation of CPU activity, as explained in Chapter 6)
- Kernel memory (paged and nonpaged pool) limits (other tools show only current size)

An introductory experiment using Process Explorer follows.

EXPERIMENT: Viewing Process Details with Process Explorer

Download the latest version of Process Explorer from *www.sysinternals.com* and run it. The first time you run it, you will receive a message that symbols are not currently configured. If properly configured, Process Explorer can access symbol information to display the symbolic name of the thread start function and functions on its call stack (available by double-clicking on a process and clicking on the Threads tab). This is useful for identifying what threads are doing within a process. To access symbols, you must have the Debugging Tools installed (described later in this chapter). Then click on Options, choose Configure Symbols, and fill in the appropriate Symbols path. For example:



Configure Symbols

Process Explorer uses symbols to resolve function names when displaying thread start addresses and thread stack locations on the Threads tab of a process' properties dialog.

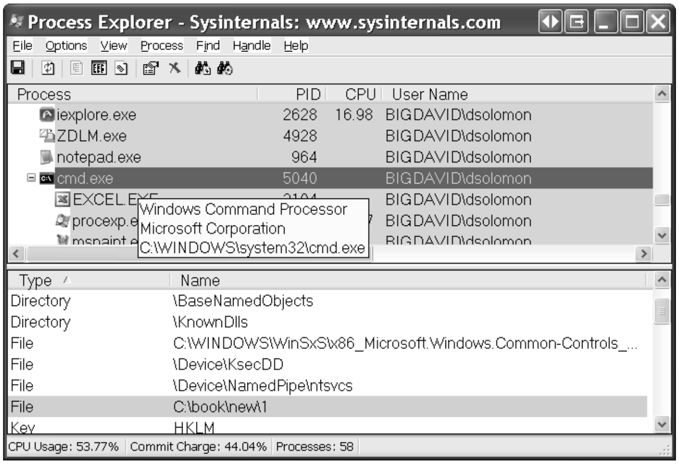
If you do not require that information you do not need to configure symbols.

Dbghelp.dll path: C:\Program Files\Debugging Tools for Windows\DbgHelp.dll

Symbols path: srv*c:\Symbols*http://msdl.microsoft.com/download/symbols

In the preceding example, the on-demand symbol server is being used to access symbols and a copy of the symbol files are being stored on the local machine in the c:\symbols folder. For more information on configuring use of the symbol server, see *http://www.microsoft.com/whdc/ddk/debugging/symbols.mspx*.

When Process Explorer starts, it shows by default the process list on the top half and the open handles for the currently selected process on the bottom half. It also shows the image description, company name, and full path if you hover the mouse pointer over the process name.



Process Explorer - Sysinternals: www.sysinternals.com

Process	PID	CPU	User Name
iexplore.exe	2628	16.98	BIGDAVID\dsolomon
ZDLM.exe	4928		BIGDAVID\dsolomon
notepad.exe	964		BIGDAVID\dsolomon
cmd.exe	5040		BIGDAVID\dsolomon
EXCEL.E			BIGDAVID\dsolomon
procexp.e			BIGDAVID\dsolomon
mspaint.e			BIGDAVID\dsolomon

Windows Command Processor
Microsoft Corporation
C:\WINDOWS\system32\cmd.exe

Type	Name
Directory	\BaseNamedObjects
Directory	\KnownDlls
File	C:\WINDOWS\WinSxS\x86_Microsoft.Windows.Common-Controls_...
File	\Device\KsecDD
File	\Device\NamedPipe\ntsvcs
File	C:\book\new\1
Key	HKLM

CPU Usage: 53.77% Commit Charge: 44.04% Processes: 58

Here are a few steps to walk you through some basic capabilities of Process Explorer:

1. Turn off the lower pane by deselecting View, Show Lower Pane. (The lower pane can show open handles or mapped DLLs and memory mapped files—these are explored in Chapters 3 and 7.)
2. Notice that processes hosting services are highlighted by default in pink. Your own processes are highlighted in blue. (These colors can be configured.)
3. Hover your mouse pointer over the image name for processes, and notice the full path displayed by the ToolTip.
4. Click on View, Select Columns, and add the image path.
5. Sort on the process column, and notice the tree view disappears. (You can either display tree view or sort by any of the columns shown.) Click again to sort from Z to A. Then click again and the display returns to tree view.
6. Deselect View, Show Processes From All Users to show only your processes.
7. Go to Options, Difference Highlight Duration, and change the value to 5 seconds. Then launch a new process (anything), and notice the new process highlighted in green for 5 seconds. Exit this new process, and notice the process is highlighted in red for 5 seconds before disappearing from the display. This can be useful to see processes being created and exiting on your system.
8. Finally, double-click on a process and explore the various tabs available from the process properties display. (These will be referenced in various experiments throughout the book where the information being shown is being explained.)

A *thread* is the entity within a process that Windows schedules for execution. Without it, the process's program can't run. A thread includes the following essential components:

- The contents of a set of CPU registers representing the state of the processor.
- Two stacks, one for the thread to use while executing in kernel mode and one for executing in user mode.
- A private storage area called thread-local storage (TLS) for use by subsystems, run-time libraries, and DLLs.
- A unique identifier called a *thread ID* (also internally called a *client ID*—process IDs and thread IDs are generated out of the same namespace, so they never overlap).
- Threads sometimes have their own security context that is often used by multithreaded server applications that impersonate the security context of the clients that they serve.

The volatile registers, stacks, and private storage area are called the thread's *context*. Because this information is different for each machine architecture that Windows runs on, this structure, by necessity, is architecture-specific. The Windows *GetThreadContext* function provides access to this architecture-specific information (called the CONTEXT block).

Fibers vs. Threads

Fibers allow an application to schedule its own "threads" of execution rather than rely on the priority-based scheduling mechanism built into Windows. Fibers are often called "lightweight" threads, and in terms of scheduling, they're invisible to the kernel because they're implemented in user mode in Kernel32.dll. To use fibers, a call is first made to the Windows *ConvertThreadToFiber* function. This function converts the thread to a running fiber. Afterward, the newly converted fiber can create additional fibers with the *CreateFiber* function. (Each fiber can have its own set of fibers.) Unlike a thread, however, a fiber doesn't begin execution until it's manually selected through a call to the *SwitchToFiber* function. The new fiber runs until it exits or until it calls *SwitchToFiber*, again selecting another fiber to run. For more information, see the Platform SDK documentation on fiber functions.

Although threads have their own execution context, every thread within a process shares the process's virtual address space (in addition to the rest of the resources belonging to the process), meaning that all the threads in a process can write to and read from each other's memory. Threads cannot accidentally reference the address space of another process, however, unless the other process makes available part of its private address space as a *shared memory section* (called a *file mapping object* in the Windows API) or unless one process has the right to open another process to use cross-process memory functions such as *ReadProcessMemory* and *WriteProcessMemory*.

In addition to a private address space and one or more threads, each process has a security identification and a list of open handles to objects such as files, shared memory sections, or one of the synchronization objects such as mutexes, events, or semaphores, as illustrated in Figure 1-2.

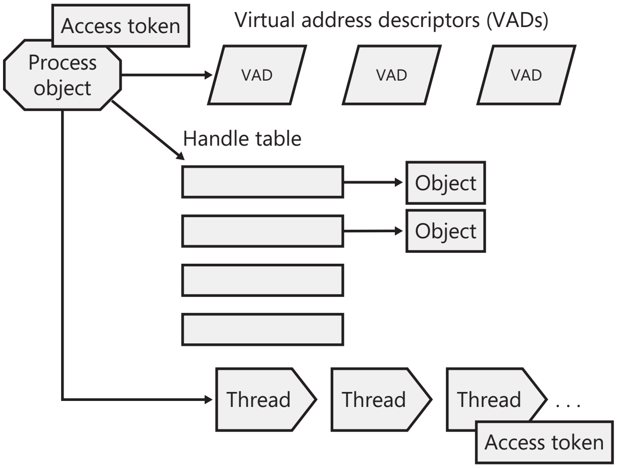


Figure 1-2 A process and its resources

Every process has a security context that is stored in an object called an *access token*. The process access token contains the security identification and credentials for the process. By default, threads don't have their own access token, but they can obtain one, thus allowing individual threads to impersonate the security context of another process—including processes running on a remote Windows system—without affecting other threads in the process. (See Chapter 8 for more details on process and thread security.)

The *virtual address descriptors* (VADs) are data structures that the memory manager uses to keep track of the virtual addresses the process is using. These data structures are described in more depth in Chapter 7.

Windows provides an extension to the process model called a *job*. A job object's main function is to allow groups of processes to be managed and manipulated as a unit. A job object allows control of certain attributes and provides limits for the process or processes associated with the job. It also records basic accounting information for all processes associated with the job and for all processes that were associated with the job but have since terminated. In some ways, the job object compensates for the lack of a structured process tree in Windows—yet in many ways it is more powerful than a UNIX-style process tree.

You'll find out much more about the internal structure of jobs, processes and threads, the mechanics of process and thread creation, and the thread-scheduling algorithms in Chapter 6.

Virtual Memory

Windows implements a virtual memory system based on a flat (linear) address space that provides each process with the illusion of having its own large, private address space. Virtual memory provides a logical view of memory that might not correspond to its physical layout. At run time, the memory manager, with assistance from hardware, translates, or *maps*, the virtual addresses into physical addresses, where the data is actually stored. By controlling the protection and mapping, the operating system can ensure that individual processes don't bump into one another or overwrite operating system data. Figure 1-3 illustrates three virtually contiguous pages mapped to three discontiguous pages in physical memory.

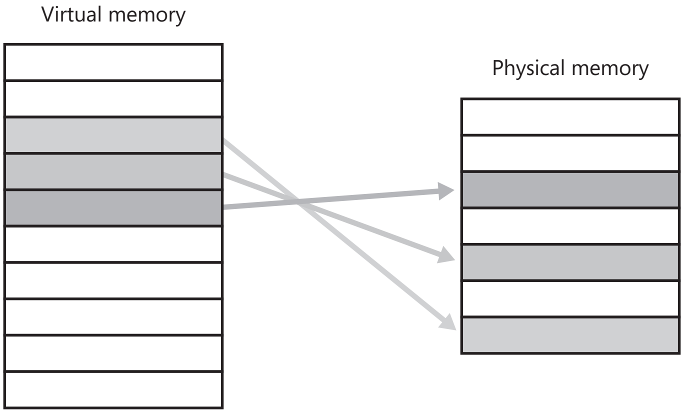


Figure 1-3 Mapping virtual memory to physical memory

Because most systems have much less physical memory than the total virtual memory in use by the running processes, the memory manager transfers, or *pages*, some of the memory contents to disk. Paging data to disk frees physical memory so that it can be used for other processes or for the operating system itself. When a thread accesses a virtual address that has been paged to disk, the virtual memory manager loads the information back into memory from disk. Applications don't have to be altered in any way to take advantage of paging because hardware support enables the memory manager to page without the knowledge or assistance of processes or threads.

The size of the virtual address space varies for each hardware platform. On 32-bit x86 systems, the total virtual address space has a theoretical maximum of 4 GB. By default, Windows allocates half this address space (the lower half of the 4 GB virtual address space, from x00000000 through x7FFFFFFF) to processes for their unique private storage and uses the other half (the upper half, addresses x80000000 through xFFFFFFFF) for its own protected operating system memory utilization. The mappings of the lower half change to reflect the virtual address space of the currently executing process, but the mappings of the upper half always consist of the operating system's virtual memory. Windows 2000 Advanced Server, Windows 2000 Datacenter Server, Windows XP (SP2 and later), and Windows Server 2003 support boot-time options (the /3GB and /USERVA qualifiers in Boot.ini, described in Chapter 5) that give processes running specially marked programs (the large address space aware flag must be set in the header of the executable image) the ability to use up to 3 GB of private address space (leaving 1 GB for the operating system). This option allows applications such as database servers to keep larger portions of a database in the process address space, thus reducing the need to map subset views of the database. Figure 1-4 shows the two virtual address space layouts supported by 32-bit Windows.

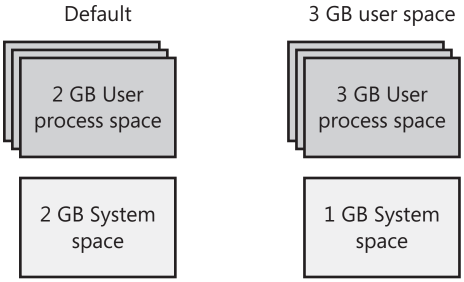


Figure 1-4 Address space layouts for 32-bit Windows

Although 3 GB is better than 2 GB, it's still not enough virtual address space to map very large (multigigabyte) databases. To address this need on 32-bit systems, Windows provides a mechanism called *Address Windowing Extension* (AWE), which allows a 32-bit application to allocate up to 64 GB of physical memory and then map views, or windows, into its 2-GB virtual address space. Although using AWE puts the burden of managing mappings of virtual to physical memory on the programmer, it does address the need of being able to directly access more physical memory than can be mapped at any one time in a 32-bit process address space.

64-bit Windows provides a much larger address space for processes: 7152 GB on Itanium systems and 8192 GB on x64 systems. Figure 1-5 shows a simplified view of the 64-bit system address space layouts. (For a detailed description, see Chapter 7.) Note that these sizes do not represent the architectural limits for these platforms, but rather implementation limits in the current versions of 64-bit Windows.

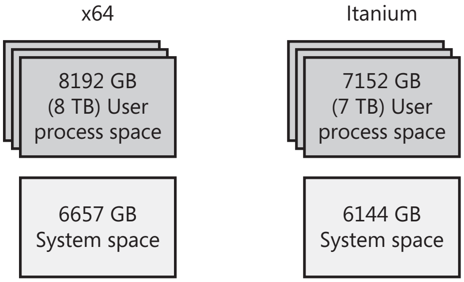

x64
Itanium
8192 GB (8 TB) User process space
7152 GB (7 TB) User process space
6657 GB System space
6144 GB System space

Figure 1-5 Address space layouts for 64-bit Windows

Details of the implementation of the memory manager, including how address translation works and how Windows manages physical memory, are described in Chapter 7.

Kernel Mode vs. User Mode

To protect user applications from accessing and/or modifying critical operating system data, Windows uses two *processor access modes* (even if the processor on which Windows is running supports more than two): *user mode* and *kernel mode*. User application code runs in user mode, whereas operating system code (such as system services and device drivers) runs in kernel mode. Kernel mode refers to a mode of execution in a processor that grants access to all system memory and all CPU instructions. By providing the operating system software with a higher privilege level than the application software has, the processor provides a necessary foundation for operating system designers to ensure that a misbehaving application can't disrupt the stability of the system as a whole.

Note The architecture of the Intel x86 processor defines four privilege levels, or rings, to protect system code and data from being overwritten either inadvertently or maliciously by code of lesser privilege. Windows uses privilege level 0 (or ring 0) for kernel mode and privilege level 3 (or ring 3) for user mode. The reason Windows uses only two levels is that some hardware architectures that were supported in the past (such as Compaq Alpha and Silicon Graphics MIPS) implemented only two privilege levels.

Although each Windows process has its own private memory space, the kernel-mode operating system and device driver code share a single virtual address space. Each page in virtual memory is tagged as to what access mode the processor must be in to read and/or write the page. Pages in system space can be accessed only from kernel mode, whereas all pages in the user address space are accessible from user mode. Read-only pages (such as those that contain executable code) are not writable from any mode.

Windows doesn't provide any protection to private read/write system memory being used by components running in kernel mode. In other words, once in kernel mode, operating system and device driver code has complete access to system space memory and can bypass Windows security to access objects. Because the bulk of the Windows operating system code runs in kernel mode, it is vital that components that run in kernel mode be carefully designed and tested to ensure that they don't violate system security.

This lack of protection also emphasizes the need to take care when loading a third-party device driver, because once in kernel mode the software has complete access to all operating system data. This vulnerability was one of the reasons behind the driver-signing mechanism introduced in Windows, which warns the user if an attempt is made to add an unauthorized (unsigned) driver. (See Chapter 9 for more information on driver signing.) Also, a mechanism called Driver Verifier helps device driver writers to find bugs (such as buffer overruns or memory leaks). Driver Verifier is also explained in Chapter 7.

As you'll see in Chapter 2, user applications switch from user mode to kernel mode when they make a system service call. For example, a Windows *ReadFile* function eventually needs to call the internal Windows routine that actually handles reading data from a file. That routine, because it accesses internal system data structures, must run in kernel mode. The transition from user mode to kernel mode is accomplished by the use of a special processor instruction that causes the processor to switch to kernel mode. The operating system traps this instruction, notices that a system service is being requested, validates the arguments the thread passed to the system function, and then executes the internal function. Before returning control to the user thread, the processor mode is switched back to user mode. In this way, the operating system protects itself and its data from perusal and modification by user processes.

> **Note** A transition from user mode to kernel mode (and back) does *not* affect thread scheduling per se—a mode transition is *not* a context switch. Further details on system service dispatching are included in Chapter 3.

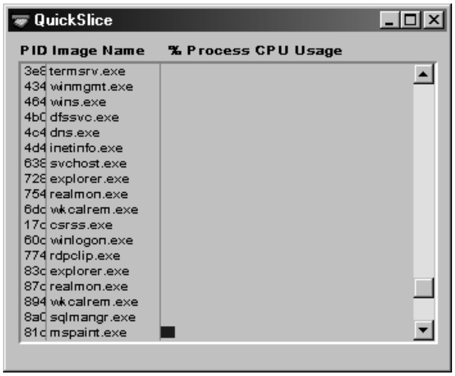
Thus, it's normal for a user thread to spend part of its time executing in user mode and part in kernel mode. In fact, because the bulk of the graphics and windowing system also runs in kernel mode, graphics-intensive applications spend more of their time in kernel mode than in user mode. An easy way to test this is to run a graphics-intensive application such as Microsoft Paint or Microsoft Pinball and watch the time split between user mode and kernel mode using one of the performance counters listed in Table 1-2.

Table 1-2 **Mode-Related Performance Counters**

Object: Counter	Function
Processor: % Privileged Time	Percentage of time that an individual CPU (or all CPUs) has run in kernel mode during a specified interval
Processor: % User Time	Percentage of time that an individual CPU (or all CPUs) has run in user mode during a specified interval
Process: % Privileged Time	Percentage of time that the threads in a process have run in kernel mode during a specified interval
Process: % User Time	Percentage of time that the threads in a process have run in user mode during a specified interval
Thread: % Privileged Time	Percentage of time that a thread has run in kernel mode during a specified interval
Thread: % User Time	Percentage of time that a thread has run in user mode during a specified interval

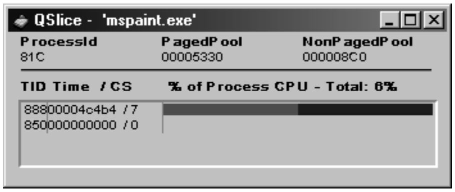
EXPERIMENT: Viewing Thread Activity with QuickSlice

QuickSlice gives a quick, dynamic view of the proportions of system and kernel time that each process currently running on your system is using. On line, the red part of the bar shows the amount of CPU time spent in kernel mode, and the blue part shows the user-mode time. (Although reproduced in the window below in black and white, the bars in the online display are always red and blue.) The total of all bars shown in the QuickSlice window should add up to 100 percent of CPU time. To run QuickSlice, click the Start button, choose Run, and enter **Qslice.exe** (assuming the Windows 2000 resource kit is in your path). For example, try running a graphics-intensive application such as Paint (Mspaint.exe). Open QuickSlice and Paint side by side, and draw squiggles in the Paint window. When you do so, you'll see Mspaint.exe running in the QuickSlice window, as shown here:



```
PID Image Name     % Process CPU Usage
3e8 termsrv.exe
434 winmgmt.exe
464 wins.exe
4b0 dfssvc.exe
4c4 dns.exe
4d4 inetinfo.exe
638 svchost.exe
728 explorer.exe
754 realmon.exe
6dc wkcalrem.exe
17c csrss.exe
60c winlogon.exe
774 rdpclip.exe
83c explorer.exe
87c realmon.exe
894 wkcalrem.exe
8a0 sqlmangr.exe
81c mspaint.exe
```

For additional information about the threads in a process, you can also double-click on a process (on either the process name or the colored bar). Here you can see the threads within the process and the relative CPU time each thread uses (not across the system):



```
ProcessId           PagedPool           NonPagedPool
81C                 00005330            000008C0

TID Time  / CS      % of Process CPU - Total: 6%
88800004c4b4 /7
850000000000 /0
```

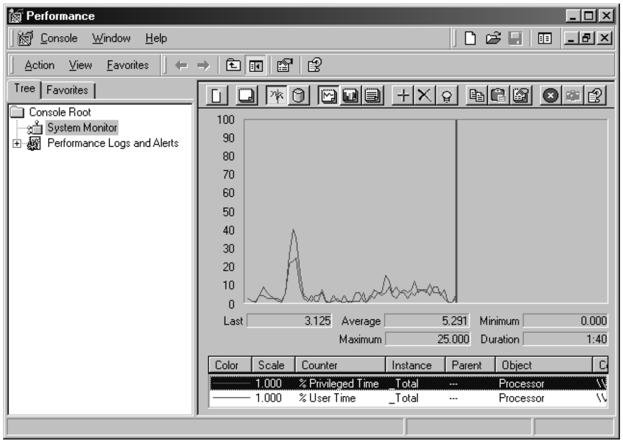
EXPERIMENT: Kernel Mode vs. User Mode

You can use the Performance tool to see how much time your system spends executing in kernel mode vs. in user mode. Follow these steps:

1. Run the Performance tool by opening the Start menu and selecting Programs/ Administrative Tools/Performance.
2. Click the Add button (+) on the toolbar.
3. With the Processor performance object selected, click the % Privileged Time counter and, while holding down the Ctrl key, click the % User Time counter.
4. Click Add, and then click Close.
5. Move the mouse rapidly back and forth. You should notice the % Privileged Time line going up when you move the mouse around, reflecting the time spent servicing the mouse interrupts and the time spent in the graphics part of the windowing system (which, as explained in Chapter 2, runs primarily as a device driver in kernel mode). (See Figure 1-6.)

6. When you're finished, click the New Counter Set button on the toolbar (or just close the tool).

You can also quickly see this activity by using Task Manager. Just click the Performance tab, and then select Show Kernel Times from the View menu. The CPU usage bar will show user-mode time in green and kernel-mode time in red.



Last 3.125 Average 5.291 Minimum 0.000
Maximum 25.000 Duration 1:40

Color	Scale	Counter	Instance	Parent	Object	C
—	1.000	% Privileged Time	_Total	---	Processor	\\
—	1.000	% User Time	_Total	---	Processor	\\

Figure 1-6 Performance tool showing time split between kernel mode and user mode

To see how the Performance tool itself uses kernel time and user time, run it again, but add the individual Process counters % User Time and % Privileged Time for every process in the system:

1. If it's not already running, run the Performance tool again. (If it is already running, start with a blank display by pressing the New Counter Set button on the toolbar.)
2. Click the Add button (+) on the toolbar.
3. Change the Performance Object to Process.
4. Select the % Privileged Time and % User Time counters.
5. Select all processes in the Instance box (except the _Total process).
6. Click Add, and then click Close.
7. Move the mouse rapidly back and forth.
8. Press Ctrl+H to turn on highlighting mode. This highlights the currently selected counter in white on Windows 2000 and black on Windows XP and Windows Server 2003.
9. Scroll through the counters at the bottom of the display to identify the processes whose threads were running when you moved the mouse, and note whether they were running in user mode or kernel mode.

You should see the Performance tool process (by looking in the Instance column for the mmc process) kernel-mode *and* user-mode time go up when you move the mouse because it is executing application code in user mode and calling Windows functions that run in kernel mode. You'll also notice kernel-mode thread activity in a process named csrss when you move the mouse. This activity occurs because the Windows subsystem's kernel-mode raw input thread, which handles keyboard and mouse input, is attached to this process. (See Chapter 2 for more information about system threads.) Finally, the process named Idle that you see spending nearly 100 percent of its time in kernel mode isn't really a process—it's a fake process used to account for idle CPU cycles. As you can observe from the mode in which the threads in the Idle process run, when Windows has nothing to do, it does it in kernel mode.

Terminal Services and Multiple Sessions

Terminal Services refers to the support in Windows for multiple interactive user sessions on a single system. With Windows Terminal Services, a remote user can establish a session on another machine, log in, and run applications on the server. The server transmits the graphical user interface to the client, and the client transmits the user's input back to the server. (This is different than X windows on UNIX systems, which permit running individual applications on a server system with the display remoted to the client, because the entire user session is remoted, not just a single application.)

The first login session at the physical console of the machine is considered the console session, or session zero. Additional sessions can be created through the use of the remote desktop connection program (Mstsc.exe) or on Windows XP systems through the use of fast user switching (described later).

The capability to create a remote session is supported on Windows 2000 Server systems but not Windows 2000 Professional. Windows XP Professional permits a single remote user to connect to the machine, but if someone is logged in at the console, the workstation is locked (that is, someone can be using the system either locally or remotely, but not at the same time).

Windows 2000 Server and Windows Server 2003 Standard Edition support two simultaneous remote connections. (This is to facilitate remote management—for example, use of management tools that require being logged in to the machine being managed.) Windows 2000 Advanced Server, Datacenter Server, Windows Server 2003 Enterprise Edition, and Datacenter Edition can support more than two sessions if appropriately licensed and configured as a terminal server.

Although Windows XP Home and Professional editions do not support multiple remote desktop connections, they do support multiple sessions created locally through a feature called fast user switching. (This feature is disabled on Windows XP Professional if the system joins a domain.) When a user chooses to disconnect their session instead of log off (for example, by clicking Start, clicking Log Off, and choosing Switch User or by holding down the Windows key and pressing "L"), the current session (that is, the processes running in that session and all the sessionwide data structures that describe the session) remains in the system and the system returns to the main logon screen. If a new user logs in, a new session is created.

For applications that want to be aware of running in a terminal server session, there are a set of Windows APIs for programmatically detecting that as well as for controlling various aspects of terminal services. (See the Platform SDK for details.)

Chapter 2 describes briefly how sessions are created and has some experiments showing how to view session information with various tools, including the kernel debugger. The "Object Manager" section in Chapter 3 describes how the system namespace for objects is instantiated on a per-session basis and how applications that need to be aware of other instances of themselves on the same system can accomplish that. Finally, Chapter 7 covers how the memory manager sets up and manages sessionwide data.

Objects and Handles

In the Windows operating system, an *object* is a single, run-time instance of a statically defined object type. An *object type* comprises a system-defined data type, functions that operate on instances of the data type, and a set of object attributes. If you write Windows applications, you might encounter process, thread, file, and event objects, to name just a few examples. These objects are based on lower-level objects that Windows creates and manages. In Windows, a process is an instance of the process object type, a file is an instance of the file object type, and so on.

An *object attribute* is a field of data in an object that partially defines the object's state. An object of type *process*, for example, would have attributes that include the process ID, a base scheduling priority, and a pointer to an access token object. *Object methods*, the means for manipulating objects, usually read or change the object attributes. For example, the *open* method for a process would accept a process identifier as input and return a pointer to the object as output.

Note Although there is a parameter named *ObjectAttributes* that a caller supplies when creating an object using either the Windows API or native object services, that parameter shouldn't be confused with the more general meaning of the term as used in this book.

The most fundamental difference between an object and an ordinary data structure is that the internal structure of an object is hidden. You must call an object service to get data out of an object or to put data into it. You can't directly read or change data inside an object. This difference separates the underlying implementation of the object from code that merely uses it, a technique that allows object implementations to be changed easily over time.

Objects provide a convenient means for accomplishing the following four important operating system tasks:

- Providing human-readable names for system resources
- Sharing resources and data among processes
- Protecting resources from unauthorized access
- Reference tracking, which allows the system to know when an object is no longer in use so that it can be automatically deallocated

Not all data structures in the Windows operating system are objects. Only data that needs to be shared, protected, named, or made visible to user-mode programs (via system services) is placed in objects. Structures used by only one component of the operating system to implement internal functions are not objects. Objects and handles (references to an instance of an object) are discussed in more detail in Chapter 3.

Security

Windows was designed from the start to be secure and to meet the requirements of various formal government and industry security ratings, such as the Common Criteria for Information Technology Security Evaluation (CCITSE) specification. Achieving a government-approved security rating allows an operating system to compete in that arena. Of course, many of these required capabilities are advantageous features for any multiuser system.

The core security capabilities of Windows include: discretionary (need-to-know) protection for all shareable system objects (such as files, directories, processes, threads, and so forth), security auditing (for accountability of subjects, or users and the actions they initiate), password authentication at logon, and the prevention of one user from accessing uninitialized resources (such as free memory or disk space) that another user has deallocated.

Windows has two forms of access control over objects. The first form—discretionary access control—is the protection mechanism that most people think of when they think of operating system security. It's the method by which owners of objects (such as files or printers) grant or deny access to others. When users log in, they are given a set of security credentials, or a security context. When they attempt to access objects, their security context is compared to the access control list on the object they are trying to access to determine whether they have permission to perform the requested operation.

Privileged access control is necessary for those times when discretionary access control isn't enough. It's a method of ensuring that someone can get to protected objects if the owner isn't available. For example, if an employee leaves a company, the administrator needs a way to gain access to files that might have been accessible only to that employee. In that case, under Windows, the administrator can take ownership of the file so that you can manage its rights as necessary.

Security pervades the interface of the Windows API. The Windows subsystem implements object-based security in the same way the operating system does; the Windows subsystem protects shared Windows objects from unauthorized access by placing Windows security descriptors on them. The first time an application tries to access a shared object, the Windows subsystem verifies the application's right to do so. If the security check succeeds, the Windows subsystem allows the application to proceed.

The Windows subsystem implements object security on a number of shared objects, some of which were built on top of native Windows objects. The Windows objects include desktop objects, window objects, menu objects, files, processes, threads, and several synchronization objects.

For a comprehensive description of Windows security, see Chapter 8.

Registry

If you've worked at all with Windows operating systems, you've probably heard about or looked at the registry. You can't talk much about Windows internals without referring to the registry because it's the system database that contains the information required to boot and configure the system, systemwide software settings that control the operation of Windows , the security database, and per-user configuration settings (such as which screen saver to use).

In addition, the registry is a window into in-memory volatile data, such as the current hardware state of the system (what device drivers are loaded, the resources they are using, and so on) as well as the Windows performance counters. The performance counters, which aren't actually "in" the registry, are accessed through the registry functions. See Chapter 4 for more on how performance counter information is accessed from the registry.

Although many Windows users and administrators will never need to look directly into the registry (because you can view or change most configuration settings with standard administrative utilities), it is still a useful source of Windows internals information because it contains many settings that affect system performance and behavior. (If you decide to directly change registry settings, you must exercise extreme caution; any changes might adversely affect system performance or, worse, cause the system to fail to boot successfully.) You'll find references to individual registry keys throughout this book as they pertain to the component being described. Most registry keys referred to in this book are under HKEY_LOCAL_MACHINE, which we'll abbreviate throughout as HKLM.

For further information on the registry and its internal structure, see Chapter 4.

Unicode

Windows differs from most other operating systems in that most internal text strings are stored and processed as 16-bit-wide Unicode characters. Unicode is an international character set standard that defines unique 16-bit values for most of the world's known character sets. (For more information about Unicode, see *www.unicode.org* as well as the programming documentation in the MSDN Library.)

Because many applications deal with 8-bit (single-byte) ANSI character strings, Windows functions that accept string parameters have two entry points: a Unicode (wide, 16-bit) and an ANSI (narrow, 8-bit) version. The Windows 95, Windows 98, and Windows Millennium Edition implementations of Windows don't implement all the Unicode interfaces to all the Windows functions, so applications designed to run on one of these operating systems as well as Windows typically use the narrow versions. If you call the narrow version of a Windows function, input string parameters are converted to Unicode before being processed by the system and output parameters are converted from Unicode to ANSI before being returned to the application. Thus, if you have an older service or piece of code that you need to run on Windows but this code is written using ANSI character text strings, Windows will convert the ANSI characters into Unicode for its own use. However, Windows never converts the *data* inside files—it's up to the application to decide whether to store data as Unicode or as ANSI.

In previous editions of Windows, Asian and Middle East editions were a superset of the core U.S. and European editions and contained additional Windows functions to handle more complex text input and layout requirements (such as right-to-left text input). As of Windows 2000, all language editions contain the same Windows functions. Instead of having separate language versions, Windows has a single worldwide binary so that a single installation can support multiple languages (by adding various language packs). Applications can also take advantage of Windows functions that allow single worldwide application binaries that can support multiple languages.

Digging into Windows Internals

Although much of the information in this book is based on reading the Windows source code and talking to the developers, you don't have to take *everything* on faith. Many details about the internals of Windows can be exposed and demonstrated by using a variety of available tools, such as those that come with Windows, the Windows Support Tools, the Windows resource kit tools, and the Windows debugging tools. These tool packages are briefly described later in this section.

To encourage your exploration of Windows internals, we've included "Experiment" sidebars throughout the book that describe steps you can take to examine a particular aspect of Windows internal behavior. (You already saw one of these sections earlier in this chapter.) We encourage you to try these experiments so that you can see in action many of the internals topics described in this book.

Table 1-3 shows a list of the tools used in this book and where they come from.

Table 1-3 Tools for Viewing Windows Internals

Tool	Image Name	Origin
Startup Programs Viewer	AUTORUNS	*www.sysinternals.com*
Dependency Walker	DEPENDS	Support Tools, Platform SDK
DLL List	LISTDLLS	*www.sysinternals.com*
EFS Information Dumper	EFSDUMP	*www.sysinternals.com**
File Monitor	FILEMON	*www.sysinternals.com*
Global Flags	GFLAGS	Support Tools
Handle Viewer	HANDLE	*www.sysinternals.com*
Junction tool	JUNCTION	*www.sysinternals.com*
Kernel debuggers	WINDBG, KD	Debugging tools, Platform SDK, Windows DDK
Live Kernel Debugging	LIVEKD	*www.sysinternals.com*
Logon Sessions	LOGINSESSIONS	*www.sysinternals.com*
Object Viewer	WINOBJ	*www.sysinternals.com*
Open Handles	OH	Resource kits
Page Fault Monitor	PFMON	Support Tools, Resource kits, Platform SDK
Pending File Moves	PENDMOVES	*www.sysinternals.com*
Performance tool	PERFMON.MSC	Windows built-in tool
PipeList tool	PIPELIST	*www.sysinternals.com*
Pool Monitor	POOLMON	Support Tools, Windows DDK
Process Explorer	PROCEXP	*www.sysinternals.com*
Get SID tool	PSGETSID	*www.sysinternals.com*
Process Statistics	PSTAT	Support Tools, Windows 2000 Resource kits, Platform SDK, *www.reskit.com*
Process Viewer	PVIEWER (in the Support Tools) or PVIEW (in the Platform SDK)	Platform SDK
Quick Slice	QSLICE	Windows 2000 resource kits
Registry Monitor	REGMON	*www.sysinternals.com*
Service Control	SC	Windows XP, Platform SDK, Windows 2000 resource kits
Task (Process) List	TLIST	Debugging tools
Task Manager	TASKMGR	Windows built-in tool
TDImon	TDIMON	*www.sysinternals.com*

Performance Tool

We'll refer to the Performance tool found in the Administrative Tools folder on the Start menu (or via Control Panel) throughout this book. The Performance tool has three functions: system monitoring, viewing performance counter logs, and setting alerts. For simplicity, when we refer to the Performance tool, we are referring to the System Monitor function within the tool.

The Performance tool can provide more information about how your system is operating than any other single utility. It includes hundreds of counters for various objects. For each major topic described in this book, a table of the relevant Windows performance counters is included.

The Performance tool contains a brief description for each counter. To see the descriptions, select a counter in the Add Counter window and click the Explain button. Or open the Performance Counter Reference help file in the resource kit. For information on how to interpret these counters to detect bottlenecks or plan capacity, see the section "Performance Monitoring" in the *Windows 2000 Server Operations Guide*, which is part of the Windows 2000 Server Resource Kit. These chapters provide an excellent description to anyone seriously interested in understanding Windows performance. For Windows XP and Windows Server 2003 , see the Windows Server 2003 Resource Kit Performance Counters Reference documentation (available online at www.microsoft.com).

Note that all the Windows performance counters are accessible programmatically. The section "HKEY_PERFORMANCE_DATA" in Chapter 4 has a brief description of the components involved in retrieving performance counters through the Windows API.

Windows Support Tools

The Windows Support Tools consist of about 40 tools useful in administering and troubleshooting Windows systems. Many of these tools were formerly part of the Windows NT 4 resource kits.

You can install the Support Tools by running Setup.exe in the \Support\Tools folder on any Windows product distribution media. For Windows 2000, the Support Tools are the same on Windows 2000 Professional, Server, Advanced Server, and Datacenter Server. Windows XP has its own version of the Support Tools, as does Windows Server 2003.

Windows Resource Kits

The Windows resource kits supplement the Support Tools, adding additional tools for system administration and support. The Windows 2003 Resource Kit tools are freely downloadable from *www.microsoft.com* (by searching for "resource kit tools"). They can be installed on Windows XP or Windows Server 2003.

There are two editions of the Windows 2000 resource kits: the Windows 2000 Professional Resource Kit and the Windows 2000 Server Resource Kit. (Supplement 1 is the most recent version.) Although the latter kit is a superset of the former and can be installed on Windows 2000 Professional systems, none of the experiments in this book use the tools that are included only with the Windows 2000 Server Resource Kit. Unlike the Windows Server 2003 Resource Kit, these tools are not freely downloadable. However, the Windows 2000 Server Resource Kit is included with the MSDN and TechNet subscriptions.

Kernel Debugging

Kernel debugging means examining internal kernel data structures and/or stepping through functions in the kernel. It is a useful way to investigate Windows internals because you can display internal system information not available through any other tools and get a clearer idea of code flows within the kernel.

Kernel debugging can be performed with a variety of tools: the Windows Debugging Tools from Microsoft, LiveKD from *www.sysinternals.com*, or SoftIce from Compuware NuMega. Before describing these tools, let's examine a file that you'll need to perform any type of kernel debugging.

Symbols for Kernel Debugging

Symbol files contain the names of functions and variables. They are generated by the linker and used debuggers to reference and display these names during a debug session. This information is not usually stored in the binary image because it is not needed to execute the code. This means that binaries are smaller and faster. However, this means that when debugging, you must make sure that the debugger can access the symbol files that are associated with the images you are referencing during a debugging session.

To use any of the kernel debugging tools to examine internal Windows kernel data structures (such as the process list, thread blocks, loaded driver list, memory usage information, and so on), you must have the correct symbol files for at least the kernel image, Ntoskrnl.exe. (The section "Architecture Overview" in Chapter 2 explains more about this file.) Symbol table files must match the version of the image they were taken from. For example, if you install a Windows Service Pack or hot fix, you must obtain the matching, updated symbol files for at least the kernel image; otherwise, you'll get a checksum error when you try to load them with the kernel debugger.

While it is possible to download and install symbols for various versions of Windows, updated symbols for hot fixes are not always available. The easiest solution to obtain the correct version of symbols for debugging is to use the Microsoft on-demand symbol server by using a special syntax for the symbol path that you specify in the debugger. For example, the following symbol path causes the debugging tools to load required symbols from the Internet symbol server and keep a local copy in the c:\symbols folder:

```
srv*c:\symbols*http://msdl.microsoft.com/download/symbols
```

For detailed instructions on how to use the symbol server, see the Debugging Tools help file or the Web page *www.microsoft.com/whdc/ddk/debugging/symbols.mspx*.

Windows Debugging Tools

The Windows Debugging Tools package contains advanced debugging tools used in this book to explore Windows internals. You can find the latest version at *www.microsoft.com/whdc/ddk/debugging*. These tools can be used to debug user-mode processes as well as the kernel. (See the following sidebar.)

Note The Windows Debugging Tools are updated frequently and released independently of Windows operating system versions, so check often for new versions.

User-Mode Debugging

The debugging tools can also be used to attach to a user-mode process and examine and/or change process memory. There are two options when attaching to a process:

- **Invasive** Unless specified otherwise, when you attach to a running process, the *DebugActiveProcess* Windows function is used to establish a connection between the debugger and the debugee. This permits examining and/or changing process memory, setting breakpoints, and performing other debugging functions. In Windows 2000, when the debugger exits, the debugee process is killed. However, as of Windows XP, you can detach a debugger without killing the target process.
- **Noninvasive** With this option, the debugger simply opens the process with the *OpenProcess* function. It does not attach to the process as a debugger. This allows you to examine and/or change memory in the target process, but you cannot set breakpoints. The advantage of this option is that you can exit the debugger on Windows 2000 without killing the target process.

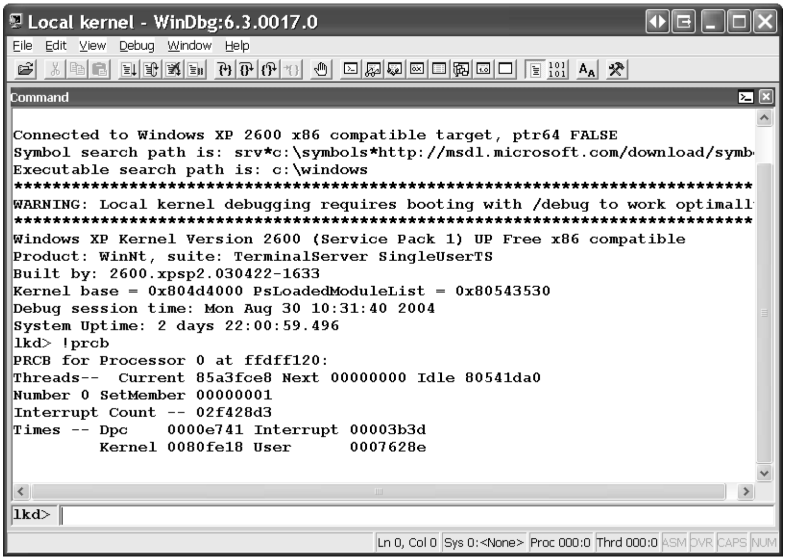
You can also open user-mode process dump files with the debugging tools. User mode dump files are explained in Chapter 3 in the section on exception dispatching.

There are two primary variants of the Microsoft debuggers that can be used for kernel debugging: a command-line version (Kd.exe) and a graphical user interface (GUI) version (Windbg.exe). Both provide the same set of commands, so which you choose is a matter of personal preference. You can perform three types of kernel debugging with these tools:

- Open a crash dump file created as a result of a Windows system crash. (See Chapter 14 for more information on crash dumps.)
- Connect to a live, running system and examine the system state (or set breakpoints if you're debugging device driver code). This operation requires two computers–a target

and a host. The target is the system being debugged, and the host is the system running the debugger. The target system can be either local (connected to the host via a null modem or IEEE 1394 cable) or remote (connected to the host via a modem). The target system must be booted with the /DEBUG qualifier (either by pressing F8 during the boot process and selecting Debug Mode or by adding a boot selection entry in Boot.ini).

- For Windows XP and Windows Server 2003 systems, connect to the local system and examine the system state. This is called *local kernel debugging*. To initiate local kernel debugging, select the menu item File, select Kernel Debug, click on the Local tab, and click OK. An example output screen is shown in Figure 1-7. Some kernel debugger commands do not work when used in local kernel debugging mode (such as viewing kernel stacks and creating a memory dump with the .dump command). However, you can use the free LiveKd tool from *www.sysinternals.com* in cases where the native local debugging support does not work. (See the next section.)



```
Connected to Windows XP 2600 x86 compatible target, ptr64 FALSE
Symbol search path is: srv*c:\symbols*http://msdl.microsoft.com/download/symb
Executable search path is: c:\windows
*****************************************************************************
WARNING: Local kernel debugging requires booting with /debug to work optimall
*****************************************************************************
Windows XP Kernel Version 2600 (Service Pack 1) UP Free x86 compatible
Product: WinNt, suite: TerminalServer SingleUserTS
Built by: 2600.xpsp2.030422-1633
Kernel base = 0x804d4000 PsLoadedModuleList = 0x80543530
Debug session time: Mon Aug 30 10:31:40 2004
System Uptime: 2 days 22:00:59.496
lkd> !prcb
PRCB for Processor 0 at ffdff120:
Threads--  Current 85a3fce8 Next 00000000 Idle 80541da0
Number 0 SetMember 00000001
Interrupt Count -- 02f428d3
Times -- Dpc    0000e741 Interrupt 00003b3d
         Kernel 0080fe18 User      0007628e
```

```
lkd>
```

Figure 1-7 Local kernel debugging

Once connected in kernel debugging mode, you can use one of the many *debugger extension commands* (commands that begin with "!") to display the contents of internal data structures such as threads, processes, I/O request packets, and memory management information. Throughout this book, the relevant kernel debugger commands and output are included as they apply to each topic being discussed. In addition, the *dt* (display type) command can format over 400 kernel structures because the kernel symbol files for Windows 2000 Service Pack 3, Windows XP, and Windows Server 2003 contain type information that the debugger can use to format structures.

EXPERIMENT: Displaying Type Information for Kernel Structures

To display the list of kernel structures whose type information is included in the kernel symbols, type **dt nt!_*** in the kernel debugger. A sample partial output is shown below:

```
lkd> dt nt!_*
          nt!_LIST_ENTRY
          nt!_LIST_ENTRY
          nt!_IMAGE_NT_HEADERS
          nt!_IMAGE_FILE_HEADER
          nt!_IMAGE_OPTIONAL_HEADER
          nt!_IMAGE_NT_HEADERS
          nt!_LARGE_INTEGER
```

You can also use the dt command to search for specific structures by using its wildcard lookup capability. For example, if you were looking for the structure name for an interrupt object, type **dt nt!_*interrupt***:

```
lkd> dt nt!_*interrupt*
          nt!_KINTERRUPT
          nt!_KINTERRUPT_MODE
```

Then, you can use dt to format a specific structure as shown below:

```
lkd> dt nt!_kinterrupt
nt!_KINTERRUPT
   +0x000 Type             : Int2B
   +0x002 Size             : Int2B
   +0x004 InterruptListEntry : _LIST_ENTRY
   +0x00c ServiceRoutine   : Ptr32
   +0x010 ServiceContext   : Ptr32 Void
   +0x014 SpinLock         : Uint4B
   +0x018 TickCount        : Uint4B
   +0x01c ActualLock       : Ptr32 Uint4B
   +0x020 DispatchAddress  : Ptr32
   +0x024 Vector           : Uint4B
   +0x028 Irql             : UChar
   +0x029 SynchronizeIrql  : UChar
   +0x02a FloatingSave     : UChar
   +0x02b Connected        : UChar
   +0x02c Number           : Char
   +0x02d ShareVector      : UChar
   +0x030 Mode             : _KINTERRUPT_MODE
   +0x034 ServiceCount     : Uint4B
   +0x038 DispatchCount    : Uint4B
   +0x03c DispatchCode     : [106] Uint4B
```

Note that dt does not show substructures (structures within structures) by default. To recurse through substructures, use the "-r" switch. For example, using this switch to display the kernel interrupt object shows the format of the _LIST_ENTRY structure stored at the InterruptListEntry field:

```
lkd> dt nt!_kinterrupt -r
nt!_KINTERRUPT
   +0x000 Type             : Int2B
   +0x002 Size             : Int2B
   +0x004 InterruptListEntry :
      +0x000 Flink            : Ptr32
         +0x000 Flink            : Ptr32 _LIST_ENTRY
         +0x004 Blink            : Ptr32 _LIST_ENTRY
      +0x004 Blink            : Ptr32
         +0x000 Flink            : Ptr32 _LIST_ENTRY
         +0x004 Blink            : Ptr32 _LIST_ENTRY
   +0x00c ServiceRoutine   : Ptr32
```

The Windows Debugging Tools help file explains how to set up and use the kernel debuggers. Additional details on using the kernel debuggers that are aimed primarily at device driver writers can be found in the Windows DDK documentation. There are also several useful Knowledge Base articles on the kernel debugger. Search for "debugref" in the Windows Knowledge Base (an online database of technical articles) on *support.microsoft.com.*

LiveKd Tool

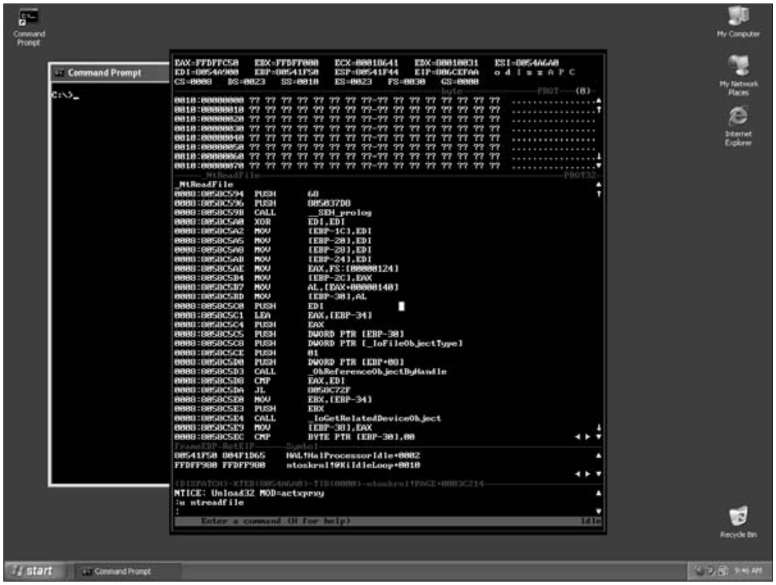
LiveKd is a free tool from *www.sysinternals.com* that allows you to use the standard Microsoft kernel debuggers just described to examine the running system without requiring a second computer to act as the host (via a null modem cable). While the built-in support for local kernel debugging works only on Windows XP and Windows Server 2003, LiveKd permits local kernel debugging on Windows NT 4.0, Windows 2000, Windows XP, and Windows Server 2003.

You run LiveKd just as you would Windbg or Kd. LiveKd passes any command-line options you specify to the debugger you select. By default, LiveKd runs the new command-line kernel debugger (Kd). To run the GUI debugger (Windbg), specify the –w switch. To see the help files on the switches for LiveKd, specify the –? switch.

LiveKd presents a simulated crash dump file to the debugger, so you can perform any operations in LiveKd that are supported on a crash dump. Because LiveKd is relying on physical memory to back the simulated dump, the kernel debugger might run into situations in which data structures are in the middle of being changed by the system and are inconsistent. Each time the debugger is launched, it gets a snapshot of the system state, so if you want to refresh the snapshot, quit the debugger (with the "q" command) and LiveKd will ask you whether you want to start it again. If the debugger gets in a loop in printing output, press Ctrl+C to interrupt the output, quit, and rerun it. If it hangs, press Ctrl+Break, which will terminate the debugger process and ask you whether you want to run the debugger again.

SoftICE

Another debugging tool that doesn't require two machines for live kernel debugging is a third-party kernel debugger called SoftICE, which you can buy from Compuware NuMega. (See *www.compuware.com* for details.) SoftICE has essentially the same capabilities as the Windows debugging tools, but it also supports stepping between user-mode and kernel-mode code. It also supports the Microsoft kernel extension DLLs, so most of the commands we describe in the book also work in SoftICE. Figure 1-8 shows the SoftICE user interface, which appears in response to the SoftICE activation key (by default, Ctrl+D) as a window on the desktop of the machine on which it's running.



```
C:\>_
```

```
EAX=FFDFFC50   EBX=FFDFF000   ECX=80010641   EDX=80010031   ESI=8054A6A0
EDI=8054A900   EBP=80541F50   ESP=80541F44   EIP=806CEFAA   o d I s z A P C
CS=0008   DS=0023   SS=0010   ES=0023   FS=0030   GS=0000
--------------------------------------------------byte----------------PROT---(0)-
0010:00000000 ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ................
0010:00000010 ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ................
0010:00000020 ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ................
0010:00000030 ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ................
0010:00000040 ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ................
0010:00000050 ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ................
0010:00000060 ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ................
0010:00000070 ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ................
-----_NtReadFile-----------------------------------------------------------PROT32-
_NtReadFile
0008:8058C594  PUSH      60
0008:8058C596  PUSH      805037D0
0008:8058C59B  CALL      __SEH_prolog
0008:8058C5A0  XOR       EDI,EDI
0008:8058C5A2  MOV       [EBP-1C],EDI
0008:8058C5A5  MOV       [EBP-20],EDI
0008:8058C5A8  MOV       [EBP-28],EDI
0008:8058C5AB  MOV       [EBP-24],EDI
0008:8058C5AE  MOV       EAX,FS:[00000124]
0008:8058C5B4  MOV       [EBP-2C],EAX
0008:8058C5B7  MOV       AL,[EAX+00000140]
0008:8058C5BD  MOV       [EBP-30],AL
0008:8058C5C0  PUSH      EDI
0008:8058C5C1  LEA       EAX,[EBP-34]
0008:8058C5C4  PUSH      EAX
0008:8058C5C5  PUSH      DWORD PTR [EBP-30]
0008:8058C5C8  PUSH      DWORD PTR [_IoFileObjectType]
0008:8058C5CE  PUSH      01
0008:8058C5D0  PUSH      DWORD PTR [EBP+08]
0008:8058C5D3  CALL      _ObReferenceObjectByHandle
0008:8058C5D8  CMP       EAX,EDI
0008:8058C5DA  JL        8058C72F
0008:8058C5E0  MOV       EBX,[EBP-34]
0008:8058C5E3  PUSH      EBX
0008:8058C5E4  CALL      _IoGetRelatedDeviceObject
0008:8058C5E9  MOV       [EBP-30],EAX
0008:8058C5EC  CMP       BYTE PTR [EBP-30],00
FrameEBP-RetEIP-------Symbol-------------------------------------------------------
80541F50 804F1D65    HAL!HalProcessorIdle+0002
FFDFF980 FFDFF980    ntoskrnl!@KiIdleLoop+0010

(DISPATCH)-KTEB(8054A6A0)-TID(0000)-ntoskrnl!PAGE+000C214-------------------------
NTICE: Unload32 MOD=actxprxy
:u ntreadfile
:
     Enter a command (H for help)                                              Idle
```

Figure 1-8 The SoftICE interface

Platform Software Development Kit (SDK)

The Platform SDK is part of the MSDN Professional and higher subscription levels, or it can be downloaded for free from *msdn.microsoft.com*. It contains the documentation, C header files, and libraries necessary to compile and link Windows applications. (Although Microsoft Visual C++ comes with a copy of these header files, the versions contained in the Platform SDK always match the latest version of the Windows operating systems, whereas the version that comes with Visual C++ might be an older version that was current when Visual C++ was released.) From an internals perspective, items of interest in the Platform SDK include the Windows API header files (\Program Files\Microsoft SDK\Include) as well as several utilities (Pfmon.exe, Pstat.exe, Pview.exe, Vadump.exe, and Winobj.exe). Some tools in the Platform SDK also come with the Support Tools and Resource Kits. Finally, a few of these tools are also shipped as example source code in both the Platform SDK and the MSDN Library.

Device Driver Kit (DDK)

The Windows DDK is also shipped as part of the MSDN Professional (and higher) subscription levels, but unlike the Platform SDK, it is not available for free download (although you can order the CD-ROM for a minimal cost). The Windows DDK documentation is included in the MSDN Library.

Although the DDK is aimed at device-driver developers, it is an abundant source of Windows-internals information. For example, while Chapter 9 describes the I/O system architecture, driver model, and basic device driver data structures, it does not describe the individual kernel support functions in detail. The DDK documentation contains a comprehensive description of all the Windows kernel support functions and mechanisms used by device drivers in both a tutorial and reference form.

Besides including the documentation, the DDK contains header files (in particular, Ntddk.h and Wdm.h) that define key internal data structures and constants as well as interfaces to many internal system routines. These files are useful when exploring Windows internal data structures with the kernel debugger because although the general layout and content of these structures are shown in this book, detailed field-level descriptions (such as size and data types) are not. A number of these data structures (such as object dispatcher headers, wait blocks, events, mutants, semaphores, and so on) are, however, fully described in the DDK.

So if you want to dig into the I/O system and driver model beyond what is presented in this book, read the DDK documentation (especially the Kernel-Mode Driver Architecture Design Guide and Reference manuals). Another excellent source is *Programming the Microsoft Windows Driver Model, Second Edition* (Microsoft Press) by Walt Oney.

Sysinternals Tools

Many experiments in this book use freeware tools that you can download from *www.sysinternals.com*. Mark Russinovich, coauthor of this book, wrote most of these tools. The most popular tools include Process Explorer, Filemon, and Regmon. Note that many of these utilities involve the installation and execution of kernel-mode device drivers and thus require administrator privileges.

Conclusion

In this chapter, you've been introduced to the key Windows technical concepts and terms that will be used throughout the book. You've also had a glimpse of the many useful tools available for digging into Windows internals. Now we're ready to begin our exploration of the internal design of the system, beginning with an overall view of the system architecture and its key components.

Chapter 2

System Architecture

Now that we've covered the terms, concepts, and tools you need to be familiar with, we're ready to start our exploration of the internal design goals and structure of the Microsoft Windows operating system. This chapter explains the overall architecture of the system—the key components, how they interact with each other, and the context in which they run. To provide a framework for understanding the internals of Windows, let's first review the requirements and goals that shaped the original design and specification of the system.

Requirements and Design Goals

The following requirements drove the specification of Windows NT back in 1989:

- Provide a true 32-bit, preemptive, reentrant, virtual memory operating system
- Run on multiple hardware architectures and platforms
- Run and scale well on symmetric multiprocessing systems
- Be a great distributed computing platform, both as a network client and as a server
- Run most existing 16-bit MS-DOS and Microsoft Windows 3.1 applications
- Meet government requirements for POSIX 1003.1 compliance
- Meet government and industry requirements for operating system security
- Be easily adaptable to the global market by supporting Unicode

To guide the thousands of decisions that had to be made to create a system that met these requirements, the Windows NT design team adopted the following design goals at the beginning of the project:

- **Extensibility** The code must be written to comfortably grow and change as market requirements change.
- **Portability** The system must be able to run on multiple hardware architectures and must be able to move with relative ease to new ones as market demands dictate.
- **Reliability and robustness** The system should protect itself from both internal malfunction and external tampering. Applications should not be able to harm the operating system or other applications.

- **Compatibility** Although Windows NT should extend existing technology, its user interface and APIs should be compatible with older versions of Windows and with MS-DOS. It should also interoperate well with other systems such as UNIX, OS/2, and NetWare.
- **Performance** Within the constraints of the other design goals, the system should be as fast and responsive as possible on each hardware platform.

As we explore the details of the internal structure and operation of Windows, you'll see how these original design goals and market requirements were woven successfully into the construction of the system. But before we start that exploration, let's examine the overall design model for Windows and compare it with other modern operating systems.

Operating System Model

In most multiuser operating systems, applications are separated from the operating system itself—the operating system kernel code runs in a privileged processor mode (referred to as *kernel mode* in this book), with access to system data and to the hardware; application code runs in a nonprivileged processor mode (called *user mode*), with a limited set of interfaces available, limited access to system data, and no direct access to hardware. When a user-mode program calls a system service, the processor traps the call and then switches the calling thread to kernel mode. When the system service completes, the operating system switches the thread context back to user mode and allows the caller to continue.

Windows is similar to most UNIX systems in that it's a monolithic operating system in the sense that the bulk of the operating system and device driver code shares the same kernel-mode protected memory space. This means that any operating system component or device driver can potentially corrupt data being used by other operating system components.

Is Windows a Microkernel-Based System?

Although some claim it as such, Windows isn't a microkernel-based operating system in the classic definition of microkernels, where the principal operating system components (such as the memory manager, process manager, and I/O manager) run as separate processes in their own private address spaces, layered on a primitive set of services the microkernel provides. For example, the Carnegie Mellon University Mach operating system, a contemporary example of a microkernel architecture, implements a minimal kernel that comprises thread scheduling, message passing, virtual memory, and device drivers. Everything else, including various APIs, file systems, and networking, runs in user mode. However, commercial implementations of the Mach microkernel operating system typically run at least all file system, networking, and memory management code in kernel mode. The reason is simple: the pure microkernel design is commercially impractical because it's too inefficient.

Does the fact that so much of Windows runs in kernel mode mean that it's more susceptible to crashes than a true microkernel operating system? Not at all. Consider the following scenario. Suppose the file system code of an operating system has a bug that causes it to crash from time to time. In a traditional operating system, a bug in kernel-mode code such as the memory manager or the file system would likely crash the entire operating system. In a pure microkernel operating system, such components run in user mode, so theoretically a bug would simply mean that the component's process exits. But in practical terms, the system would crash because recovering from the failure of such a critical process would likely be impossible.

All these operating system components are, of course, fully protected from errant applications because applications don't have direct access to the code and data of the privileged part of the operating system (although they can quickly call other kernel services). This protection is one of the reasons that Windows has the reputation for being both robust and stable as an application server and as a workstation platform yet fast and nimble from the perspective of core operating system services, such as virtual memory management, file I/O, networking, and file and print sharing.

The kernel-mode components of Windows also embody basic object-oriented design principles. For example, they don't in general reach into one another's data structures to access information maintained by individual components. Instead, they use formal interfaces to pass parameters and access and/or modify data structures.

Despite its pervasive use of objects to represent shared system resources, Windows is not an object-oriented system in the strict sense. Most of the operating system code is written in C for portability and because C development tools are widely available. C doesn't directly support object-oriented constructs, such as dynamic binding of data types, polymorphic functions, or class inheritance. Therefore, the C-based implementation of objects in Windows borrows from, but doesn't depend on, features of particular object-oriented languages.

Architecture Overview

With this brief overview of the design goals and packaging of Windows, let's take a look at the key system components that make up its architecture. A simplified version of this architecture is shown in Figure 2-1. Keep in mind that this diagram is basic—it doesn't show everything. (For example, the networking components and the various types of device driver layering are not shown.)

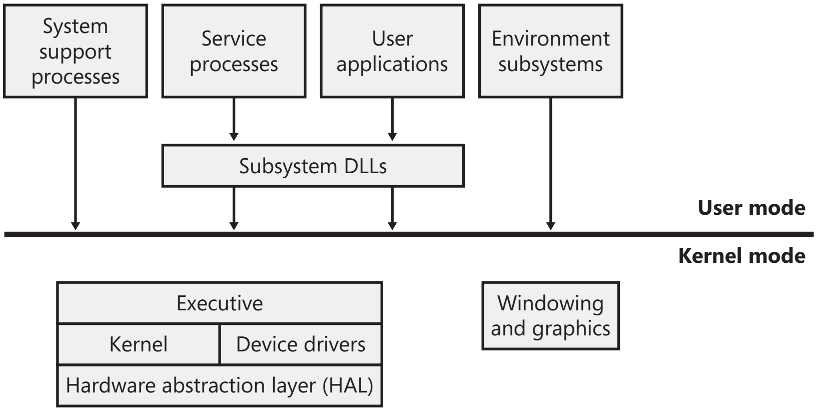


Figure 2-1 Simplified Windows architecture

In Figure 2-1, first notice the line dividing the user-mode and kernel-mode parts of the Windows operating system. The boxes above the line represent user-mode processes, and the components below the line are kernel-mode operating system services. As mentioned in Chapter 1, user-mode threads execute in a protected process address space (although while they are executing in kernel mode, they have access to system space). Thus, system support processes, service processes, user applications, and environment subsystems each have their own private process address space.

The four basic types of user-mode processes are described as follows:

- Fixed (or hardwired) *system support processes*, such as the logon process and the session manager, that are not Windows services. (That is, they are not started by the service control manager. Chapter 4 describes services in detail.)
- *Service processes* that host Windows services, such as the Task Scheduler and Spooler services. Services generally have the requirement that they run independently of user logons. Many Windows server applications, such as Microsoft SQL Server and Microsoft Exchange Server, also include components that run as services.
- *User applications*, which can be one of six types: Windows 32-bit, Windows 64-bit, Windows 3.1 16-bit, MS-DOS 16-bit, POSIX 32-bit, or OS/2 32-bit.
- *Environment subsystem server processes*, which implement part of the support for the operating system *environment*, or personality presented to the user and programmer. Windows NT originally shipped with three environment subsystems: Windows, POSIX, and OS/2. OS/2 was dropped as of Windows 2000. As of Windows XP, only the Windows subsystem is shipped in the base product—an enhanced POSIX subsystem is available as part of the free Services for Unix product.

In Figure 2-1, notice the "Subsystem DLLs" box below the "Service processes" and "User applications" boxes. Under Windows, user applications don't call the native Windows operating system services directly; rather, they go through one or more *subsystem dynamic-link libraries* (DLLs). The role of the subsystem DLLs is to translate a documented function into the appropriate internal (and generally undocumented) Windows system service calls. This translation might or might not involve sending a message to the environment subsystem process that is serving the user application.

The kernel-mode components of Windows include the following:

- The Windows *executive* contains the base operating system services, such as memory management, process and thread management, security, I/O, networking, and interprocess communication.
- The Windows *kernel* consists of low-level operating system functions, such as thread scheduling, interrupt and exception dispatching, and multiprocessor synchronization. It also provides a set of routines and basic objects that the rest of the executive uses to implement higher-level constructs.
- *Device drivers* include both hardware device drivers that translate user I/O function calls into specific hardware device I/O requests as well as file system and network drivers.
- The *hardware abstraction layer* (HAL) is a layer of code that isolates the kernel, device drivers, and the rest of the Windows executive from platform-specific hardware differences (such as differences between motherboards).
- The *windowing and graphics system* implements the graphical user interface (GUI) functions (better known as the Windows USER and GDI functions), such as dealing with windows, user interface controls, and drawing.

Table 2-1 lists the filenames of the core Windows operating system components. (You'll need to know these filenames because we'll be referring to some system files by name.) Each of these components is covered in greater detail both later in this chapter and in the chapters that follow.

Table 2-1 Core Windows System Files

Filename	Components
Ntoskrnl.exe	Executive and kernel
Ntkrnlpa.exe (32-bit systems only)	Executive and kernel with support for Physical Address Extension (PAE), which allows addressing of up to 64 GB of physical memory
Hal.dll	Hardware abstraction layer
Win32k.sys	Kernel-mode part of the Windows subsystem
Ntdll.dll	Internal support functions and system service dispatch stubs to executive functions
Kernel32.dll, Advapi32.dll, User32.dll, Gdi32.dll	Core Windows subsystem DLLs

Before we dig into the details of these system components, though, let's examine how Windows achieves portability across multiple hardware architectures.

Portability

Windows was designed to run on a variety of hardware architectures, including Intel-based CISC systems as well as RISC systems. The initial release of Windows NT supported the x86 and MIPS architecture. Support for the Digital Equipment Corporation (which was bought by Compaq, who later merged with Hewlett Packard) Alpha AXP was added shortly thereafter. (Although Alpha AXP was a 64-bit processor, Windows NT ran in 32-bit mode. During the development of Windows 2000, a native 64-bit version was running on Alpha AXP, but this never was released.) Support for a fourth processor architecture, the Motorola PowerPC, was added in Windows NT 3.51. Because of changing market demands, however, support for the MIPS and PowerPC architectures was dropped before development began on Windows 2000. Later, Compaq withdrew support for the Alpha AXP architecture, resulting in Windows 2000 being supported only on the x86 architecture. The most recent releases, Windows XP and Windows Server 2003, add support for three 64-bit processor families: the Intel Itanium IA-64 family, the AMD x86-64 family, and the Intel 64-bit Extension Technology (EM64T) for x86 (which is compatible with the AMD x86-64 architecture, although there are slight differences in instructions supported). The latter two processor families are called *64-bit extended systems* and in this book are referred to as *x64*. The most recent releases, Windows XP and Windows Server 2003, add support for three 64-bit processor families: the Intel Itanium IA-64 family, the AMD64 family, and the Intel 64-bit Extension Technology (EM64T) for x86 (which is compatible with the AMD64 architecture, although there are slight differences in instructions supported). (How Windows runs 32-bit applications on 64-bit Windows is explained in Chapter 3.)

Windows achieves portability across hardware architectures and platforms in two primary ways:

- Windows has a layered design, with low-level portions of the system that are processor-architecture-specific or platform-specific isolated into separate modules so that upper layers of the system can be shielded from the differences between architectures and among hardware platforms. The two key components that provide operating system portability are the kernel (contained in Ntoskrnl.exe) and the hardware abstraction layer (or HAL, contained in Hal.dll). Both these components are described in more detail later in this chapter. Functions that are architecture-specific (such as thread context switching and trap dispatching) are implemented in the kernel. Functions that can differ among systems within the same architecture (for example, different motherboards) are implemented in the HAL. The only other component with a significant amount of architecture-specific code is the memory manager, but even that is a small amount compared to the system as a whole.
- The vast majority of Windows is written in C, with some portions in C++. Assembly language is used only for those parts of the operating system that need to communicate directly with system hardware (such as the interrupt trap handler) or that are extremely performance-sensitive (such as context switching). Assembly language code exists not only in the kernel and the HAL but also in a few other places within the core operating system (such as the routines that implement interlocked instructions as well as one module in the local procedure call facility), in the kernel-mode part of the Windows

subsystem, and even in some user-mode libraries, such as the process startup code in Ntdll.dll (a system library explained later in this chapter).

Symmetric Multiprocessing

Multitasking is the operating system technique for sharing a single processor among multiple threads of execution. When a computer has more than one processor, however, it can execute two threads simultaneously. Thus, whereas a multitasking operating system only appears to execute multiple threads at the same time, a multiprocessing operating system actually does it, executing one thread on each of its processors.

As mentioned at the beginning of this chapter, one of the key design goals for Windows was that it had to run well on multiprocessor computer systems. Windows is a *symmetric multiprocessing* (SMP) operating system. There is no master processor—the operating system as well as user threads can be scheduled to run on any processor. Also, all the processors share just one memory space. This model contrasts with *asymmetric multiprocessing* (ASMP), in which the operating system typically selects one processor to execute operating system kernel code while other processors run only user code. The differences in the two multiprocessing models are illustrated in Figure 2-2.

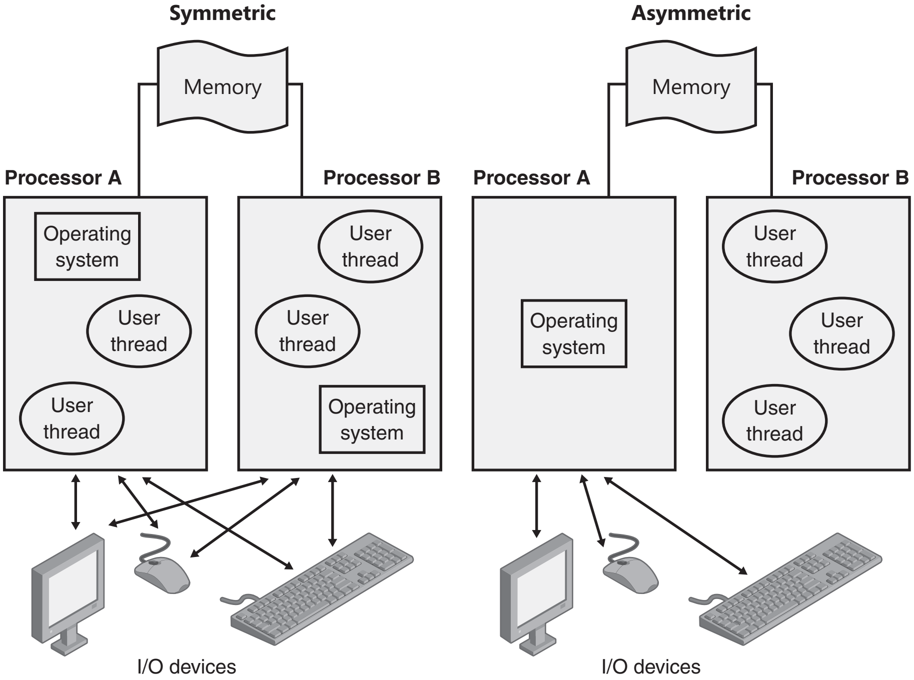


Figure 2-2 Symmetric vs. asymmetric multiprocessing

Windows XP and Windows Server 2003 support two new types of multiprocessor systems: hyperthreading and NUMA (non-uniform memory architecture). These are briefly mentioned in the following paragraphs. (For a complete detailed description of the scheduling support for these systems, see the thread scheduling section in Chapter 6.)

Hyperthreading is a technology introduced by Intel that provides many logical processors on one physical processor. Each logical processor has its CPU state, but the execution engine and onboard cache is shared. This permits one logical CPU to make progress while the other logical CPUs are busy (such as performing interrupt processing work, which prevents threads from running on that logical processor). The scheduling algorithms as of Windows XP have been enhanced to make optimal use of multiprocessor hyperthreaded machines, such as by scheduling threads on an idle physical processor versus choosing an idle logical processor on a physical processor whose other logical processors are busy.

In non-uniform memory architecture NUMA systems, processors are grouped in smaller units called nodes. Each node has its own processors and memory and is connected to the larger system through a cache-coherent interconnect bus. Windows on a NUMA system still runs as an SMP system, in that all processors have access to all memory—it's just that node-local memory is faster to reference than memory attached to other nodes. The system attempts to improve performance by scheduling threads on processors that are in the same node as the memory being used. It attempts to satisfy memory-allocation requests from within the node, but will allocate memory from other nodes if necessary.

Although Windows was originally designed to support up to 32 processors, nothing inherent in the multiprocessor design limits the number of processors to 32—that number is simply an obvious and convenient limit because 32 processors can easily be represented as a bit mask using a native 32-bit data type. In fact, the 64-bit versions of Windows support up to 64 processors, because the native size of a word on a 64-bit machine is 64 bits.

The actual number of supported processors depends on the edition of Windows being used. (See tables 2-3 and 2-4.) This number is stored in the registry value HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\LicensedProcessors. (Keep in mind that tampering with that data is a violation of the software license and modifying the registry to allow use of more processors involves more than just changing this value.)

For performance reasons, there are separate uniprocessor and multiprocessor versions of the kernel and HAL (and in the case of Windows 2000, a few other key system files). On Windows 2000, six system files (as explained in the following Note) are different on a multiprocessor system than on a uniprocessor system; on 32-bit Windows XP and Windows Server 2003 systems, only three are different. (See Table 2-2.) On 64-bit Windows systems, there is no PAE kernel, so only the kernel and HAL vary from uniprocessor to multiprocessor systems.

At installation time, the appropriate files are selected and copied to the local \Windows\System32 directory. To determine which files were copied, see the file \Windows\Repair\Setup.log, which itemizes all the files that were copied to the local system disk and where they came from off the distribution media.

Table 2-2 **Multiprocessor-Specific vs. Uniprocessor-Specific System Files**

Name of File on System Disk	Name of Uniprocessor Version on Distribution Media	Name of Multiprocessor Version on Distribution Media
Ntoskrnl.exe	Ntoskrnl.exe	Ntkrnlmp.exe
Ntkrnlpa.exe (PAE kernel; 32-bit systems only)	Ntkrnlpa.exe in \Windows\<arch>\Driver.cab	Ntkrpamp.exe in \Windows\<arch>\Driver.cab
Hal.dll	Depends on system type (See the list of HALs in Table 2-6.)	Depends on system type (See the list of HALs in Table 2-6.)
Windows 2000 only		
Win32k.sys	\I386\UNIPROC\Win32k.sys	Win32k.sys in \I386\Driver.cab
Ntdll.dll	\I386\UNIPROC\Ntdll.dll	\I386\Ntdll.dll
Kernel32.dll	\I386\UNIPROC\Kernel32.dll	\I386\Kernel32.dll

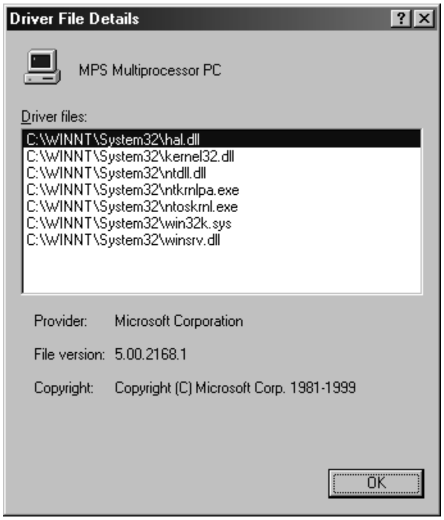
Note If you look in the \I386\UNIPROC folder in the Windows 2000 distribution tree, you'll see a file named Winsrv.dll. Although this file exists in a folder named UNIPROC, implying that there is a uniprocessor version, in fact there is only one version of this image for both multiprocessor and uniprocessor systems. This folder has been removed in Windows XP and Windows Server 2003.

EXPERIMENT: Looking at Multiprocessor-Specific Support Files on Windows 2000

You can see the files that are different for a 32-bit Windows 2000 multiprocessor system by looking at the driver details for the Computer in Device Manager:

1. Open the System properties (either by selecting System from Control Panel or by right-clicking the My Computer icon on your desktop and selecting Properties).
2. Click the Hardware tab.
3. Click Device Manager.
4. Expand the Computer object.
5. Double-click the child node underneath Computer.
6. Click the Driver tab.
7. Click Driver Details.

You should see the following dialog box for a multiprocessor system:



Driver files:

C:\WINNT\System32\hal.dll
C:\WINNT\System32\kernel32.dll
C:\WINNT\System32\ntdll.dll
C:\WINNT\System32\ntkrnlpa.exe
C:\WINNT\System32\ntoskrnl.exe
C:\WINNT\System32\win32k.sys
C:\WINNT\System32\winsrv.dll

Provider: Microsoft Corporation

File version: 5.00.2168.1

Copyright: Copyright (C) Microsoft Corp. 1981-1999

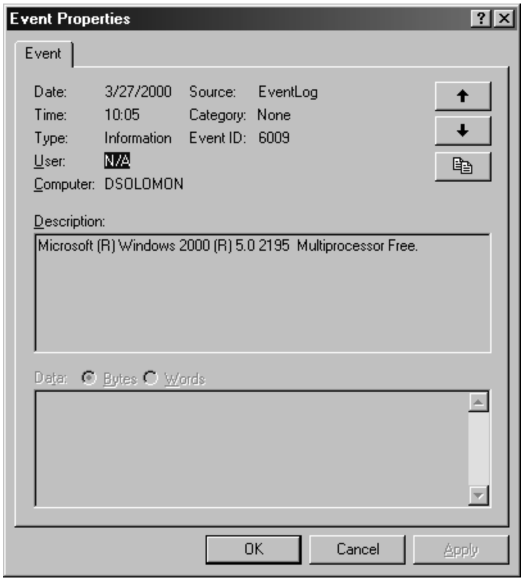
The reason for having uniprocessor versions of these key system files is performance—multiprocessor synchronization is inherently more complex and time consuming than the use of a single processor, so by having special uniprocessor versions of the key system files, this overhead is avoided on uniprocessor systems (which constitute the vast majority of systems running Windows).

Interestingly, although the uniprocessor and multiprocessor versions of Ntoskrnl are generated using conditionally compiled source code, the uniprocessor versions of Ntdll.dll and Kernel32.dll for Windows 2000 are created by patching the x86 LOCK and UNLOCK instructions, which are used to synchronize multiple threads with no-operation (NOP) instructions (which do nothing).

The rest of the system files that make up Windows (including all utilities, libraries, and device drivers) have the same version on both uniprocessor and multiprocessor systems (that is, they handle multiprocessor synchronization issues correctly). You should use this approach on any software you build, whether it is a Windows application or a device driver—keep multiprocessor synchronization issues in mind when you design your software, and test the software on both uniprocessor and multiprocessor systems.

EXPERIMENT: Checking Which Ntoskrnl Version You're Running

In Windows 2000 and later, there is no utility to show which version of Ntoskrnl you are running. However, an Event Log entry is written each time the system boots that does record the type of kernel image that loaded (uniprocessor vs. multiprocessor and free vs. checked), as shown in the following screen shot. (From the Start menu select Programs/Administrative Tools/Event Viewer, select System Log, and double-click an Event Log entry with an Event ID of 6009, indicating the entry was written at the system start.)

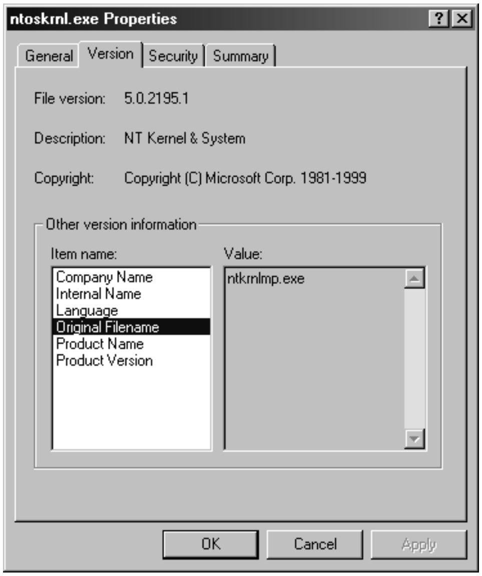


Date: 3/27/2000 Source: EventLog
Time: 10:05 Category: None
Type: Information Event ID: 6009
User: N/A
Computer: DSOLOMON

Description:
Microsoft (R) Windows 2000 (R) 5.0 2195 Multiprocessor Free.

This Event Log entry doesn't indicate whether you booted the PAE version of the kernel image that supports more than 4 GB of physical memory (Ntkrnlpa.exe). However, you can tell if you booted the PAE kernel by looking at the registry value HKLM\SYSTEM\CurrentControlSet\Control\SystemStartOptions. Also, if you boot the PAE kernel, the registry value HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management\PhysicalAddressExtension is set to 1.

You can also determine whether you installed the multiprocessor version of Ntoskrnl (or Ntkrnlpa) by examining the file properties: run Windows Explorer, right-click on Ntoskrnl.exe in your \Windows\System32 folder, and select Properties. Then click on the Version tab, and select the Original Filename property—if you're running the multiprocessor version, you'll see the following dialog box:



ntoskrnl.exe Properties

File version: 5.0.2195.1

Description: NT Kernel & System

Copyright: Copyright (C) Microsoft Corp. 1981-1999

Other version information

Item name:
- Company Name
- Internal Name
- Language
- Original Filename
- Product Name
- Product Version

Value: ntkrnlmp.exe

Finally, as mentioned earlier, you can see exactly which kernel image and HAL were selected at installation time by looking at the file \Windows\Repair\Setup.log.

Scalability

One of the key issues with multiprocessor systems is *scalability*. To run correctly on an SMP system, operating system code must adhere to strict guidelines and rules. Resource contention and other performance issues are more complicated in multiprocessing systems than in uniprocessor systems and must be accounted for in the system's design. Windows incorporates several features that are crucial to its success as a multiprocessor operating system:

- The ability to run operating system code on any available processor and on multiple processors at the same time
- Multiple threads of execution within a single process, each of which can execute simultaneously on different processors
- Fine-grained synchronization within the kernel (such as spinlocks, queued spinlocks, and pushlocks, described in Chapter 3) as well as within device drivers and server processes, which allows more components to run concurrently on multiple processors
- Programming mechanisms such as I/O completion ports (described in Chapter 9) that facilitate the efficient implementation of multithreaded server processes that can scale well on multiprocessor systems.

The scalability of the Windows kernel has evolved over time. For example, Windows Server 2003 has per-CPU scheduling queues, which permits thread scheduling decisions to occur in parallel on multiple machines. Multiprocessor thread scheduling details are covered in Chapter 6. Further details on multiprocessor synchronization can be found in Chapter 3.

Differences Between Client and Server Versions

Windows ships in both client and server retail packages. In Windows 2000, the client version is called Windows 2000 Professional. There are three Windows 2000 server versions: Windows 2000 Server, Advanced Server, and Datacenter Server.

There are six client versions of Windows XP: Windows XP Home Edition, Windows XP Professional, Windows XP Starter Edition, Windows XP Tablet PC Edition, Windows XP Media Center Edition, and Windows XP Embedded. The latter three are supersets of Windows XP Professional and are not described in detail in this book because they are all built on the same core operating system as Windows XP Professional.

There are six variants of Windows Server 2003: Windows Server 2003 Web Edition, Standard Edition, Small Business Server, Storage Server, Enterprise Edition, and Datacenter Edition.

These versions differ by:

- The number of processors supported
- The amount of physical memory supported
- The number of concurrent network connections supported (For example, a maximum of 10 concurrent connections are allowed to the file and print services in the client version.)
- Layered services that come with Server editions that don't come with the Professional edition (for example, directory services, clustering, and multiuser Terminal Services support)

Table 2-3 summarizes the differences in memory and processor support for Windows 2000. Table 2-4 lists the same information for Windows XP and Windows Server 2003. For a detailed comparison chart of the different editions of Windows Server 2003, see *http://www.microsoft.com/windowsserver2003/evaluation/features/compareeditions.mspx.*

Table 2-3 Differences Between Windows 2000 Professional and Server

Edition	Number of Processors Supported	Physical Memory Supported
Windows 2000 Professional	2	4 GB
Windows 2000 Server	4	4 GB
Windows 2000 Advanced Server	8	8 GB
Windows 2000 Datacenter Server	32	64 GB

Table 2-4 **Differences Between Windows XP and Windows Server 2003**

	Number of Processors Supported (32-bit edition)	Physical Memory Supported (32-bit edition)	Number of Processors Supported (64-bit edition)	Physical Memory Supported (Itanium editions)	Physical Memory Supported (x64 editions)
Windows XP Home Edition	1	4 GB	Not available	Not available	Not available
Windows XP Professional	2	4 GB	2	16 GB	16 GB
Windows Server 2003 Web Edition	2	2 GB	Not available	Not available	Not available
Windows Server 2003 Small Business Server	2	2 GB	Not available	Not available	Not available
Windows Server 2003 Standard Edition	4	4 GB	Not available	Not available	Not available
Windows Server 2003 Enterprise Edition	8	32 GB	8	64 GB	64 GB
Windows Server 2003 Datacenter Edition	32	128 GB on x64; 64 GB on x86	64	512 GB (1024 GB in SP1)	Not available

Although there are several client and server retail packages of the Windows operating system, they share a common set of core system files, including the kernel image, Ntoskrnl.exe (and the PAE version, Ntkrnlpa.exe); the HAL libraries; the device drivers; and the base system utilities and DLLs. These files are identical for all editions of Windows 2000.

> **Note** Windows XP was the first client release of the Windows NT code base to ship without corresponding server versions. Instead, development continued on what became Windows Server 2003 for over a year after the release of Windows XP. Therefore, the core system files are not identical for Windows XP and Windows Server 2003. Even so, the differences are not major (and in many cases, components were unchanged).

So if the kernel image for Windows 2000 Professional and Windows 2000 Server are identical (and similar for Windows XP and Windows Server 2003), how does the system know which edition is booted? By querying the registry values ProductType and ProductSuite under the HKLM\ SYSTEM\CurrentControlSet\Control\ProductOptions key. ProductType is used to distinguish whether the system is a client system or a server system (of any flavor). The valid values are listed in Table 2-5. The result is stored in the system global variable *MmProductType*, which can be queried from a device driver using the kernel-mode support function *MmIsThisAnNtAsSystem*, documented in the Windows DDK.

Table 2-5 ProductType Registry Values

Edition of Windows	Value of ProductType
Windows 2000 Professional, Windows XP Professional, Windows XP Home Edition	WinNT
Windows Server (domain controller)	LanmanNT
Windows Server (server only)	ServerNT

A different registry value, ProductSuite, distinguishes the various flavors of Windows Server systems (Standard, Enterprise, Datacenter, and so on) as well as distinguishing a Windows XP Home from a Windows XP Professional system.

If user programs need to determine which edition of Windows is running, they can call the Windows *VerifyVersionInfo* function, documented in the Platform SDK. Device drivers can call the kernel-mode function *RtlGetVersion*, documented in the Windows DDK.

So if the core files are essentially the same for the client and server versions, how do the systems differ in operation? In short, Server systems are by default optimized for system throughput as high-performance application servers, whereas the client version, although it has server capabilities, is optimized for response time for interactive desktop use. For example, based on the product type, several resource allocation decisions are made differently at system boot time, such as the size and number of operating system heaps (or pools), the number of internal system worker threads, and the size of the system data cache. Also, run-time policy decisions, such as the way the memory manager trades off system and process memory demands, differ between the server and client editions. Even some thread scheduling details have different default behavior in the two families (the default length of the time slice, or thread *quantum*—see Chapter 6 for details). Where there are significant operational differences in the two products, these are highlighted in the pertinent chapters throughout the rest of this book. Unless otherwise noted, everything in this book applies to both the client and server versions.

Checked Build

There is a special debug version of Windows 2000 Professional, Windows XP Professional, and Windows Server 2003 called the *checked build* (available only with the MSDN Professional or higher subscription). It is a recompilation of the Windows source code with a compile-time flag defined called "DBG" (to cause compile time conditional debugging and tracing code to be included). Also, to make it easier to understand the machine code, the post-processing of the Windows binaries to optimize code layout for faster execution is not performed. (See the section "Performance-Optimized Code" in the Debugging Tools help file.)

The checked build is provided primarily to aid device driver developers because it performs more stringent error checking on kernel-mode functions called by device drivers or other system code. For example, if a driver (or some other piece of kernel-mode code) makes an invalid call to a system function that is checking parameters (such as acquiring a spinlock at the wrong interrupt level), the system will stop execution when the problem is detected rather than allow some data structure to be corrupted and the system to possibly crash at a later time.

EXPERIMENT: Determining If You Are Running the Checked Build

There is no built-in tool to display whether you are running the checked build or the retail build (called the free build). However, this information is available through the "Debug" property of the Windows Management Instrumentation (WMI) Win32_OperatingSystem class. The following sample Visual Basic script displays this property:

```
strComputer = "."
Set objWMIService = GetObject("winmgmts:" _
 & "{impersonationLevel=impersonate}!\\" & strComputer & "\root\cimv2")
Set colOperatingSystems = objWMIService.ExecQuery _
 ("SELECT * FROM Win32_OperatingSystem"S)
For Each objOperatingSystem in colOperatingSystems
 Wscript.Echo "Caption: " & objOperatingSystem.Caption
 Wscript.Echo "Debug: " & objOperatingSystem.Debug
 Wscript.Echo "Version: " & objOperatingSystem.Version
Next
```

To try this, type in the preceding script and save it as file. The following is the output from running the script:

```
C:\>cscript osversion.vbs
Microsoft (R) Windows Script Host Version 5.6
Copyright (C) Microsoft Corporation 1996-2001. All rights reserved.

Caption: Microsoft Windows XP Professional
Debug: False
Version: 5.1.2600
```

This system is not running the checked build, as the Debug flag shown here says False.

Much of the additional code in the checked-build binaries is a result of using the ASSERT macro, which is defined in the DDK header file Ntddk.h and documented in the DDK documentation. This macro tests a condition (such as the validity of a data structure or parameter), and if the expression evaluates to FALSE, the macro calls the kernel-mode function *RtlAssert*, which calls *DbgPrint* to send the text of the debug message to a debug message buffer. If a kernel debugger is attached, this message is displayed automatically followed by a prompt asking the user what to do about the assertion failure (breakpoint, ignore, terminate process, or terminate thread). If the system wasn't booted with the kernel debugger (using the /DEBUG switch in Boot.ini) and no kernel debugger is currently attached, failure of an ASSERT test will crash the system. For a list of ASSERT checks made by some of the kernel support routines, see the section "Checked Build ASSERTs" in the Windows DDK documentation.

> **Note** On the checked build, if you compare Ntoskrnl.exe and Ntkrnlmp.exe or Ntkrnlpa.exe and Ntkrpamp.exe, you'll find that they are identical—they are all multiprocessor versions of the same files. In other words, there is no debug uniprocessor version of the kernel images provided with the checked build.

The checked build is also useful for system administrators because of the additional detailed informational tracing that can be enabled for certain components. (For detailed instructions, see the Microsoft Knowledge Base Article number 314743 entitled *HOWTO: Enable Verbose Debug Tracing in Various Drivers and Subsystems*.) This information output is sent to an internal debug message buffer using the *DbgPrint* function referred to earlier. To view the debug messages, you can either attach a kernel debugger to the target system (which requires booting the target system in debugging mode), use the *!dbgprint* command while performing local kernel debugging, or use the Dbgview.exe tool from *www.sysinternals.com*.

You don't have to install the entire checked build to take advantage of the debug version of the operating system. You can just copy the checked version of the kernel image (Ntoskrnl.exe) and the appropriate HAL (Hal.dll) to a normal retail installation. The advantage of this approach is that device drivers and other kernel code get the rigorous checking of the checked build without having to run the slower debug versions of all components in the system. For detailed instructions on how to do this, see the section "Installing Just the Checked Operating System and HAL" in the Windows DDK documentation. Because Microsoft doesn't supply a checked build version of Windows 2000 Server, you can also apply this technique to run the checked version of the kernel on a Windows 2000 Server system.

Finally, the checked build can also be useful for testing user-mode code only because the timing of the system is different. (This is because of the additional checking taking place within the kernel and the fact that the components are compiled without optimizations.) Often, multithreaded synchronization bugs are related to specific timing conditions. By running your tests on a system running the checked build (or at least the checked kernel and HAL), the fact that the timing of the whole system is different might cause latent timing bugs to surface that do not occur on a normal retail system.

Key System Components

Now that we've looked at the high-level architecture of Windows, let's delve deeper into the internal structure and the role each key operating system component plays. Figure 2-3 is a more detailed and complete diagram of the core Windows system architecture and components than was shown earlier in the chapter (in Figure 2-2). Note that it still does not show all components (networking in particular, which is explained in Chapter 13).

The following sections elaborate on each major element of this diagram. Chapter 3 explains the primary control mechanisms the system uses (such as the object manager, interrupts, and so forth). Chapter 5 describes the process of starting and shutting down Windows, and Chapter 4 details management mechanisms such as the registry, service processes, and Windows Management Instrumentation (WMI). Then the remaining chapters explore in even more detail the internal structure and operation of key areas such as processes and threads, memory management, security, the I/O manager, storage management, the cache manager, the Windows file system (NTFS), and networking.

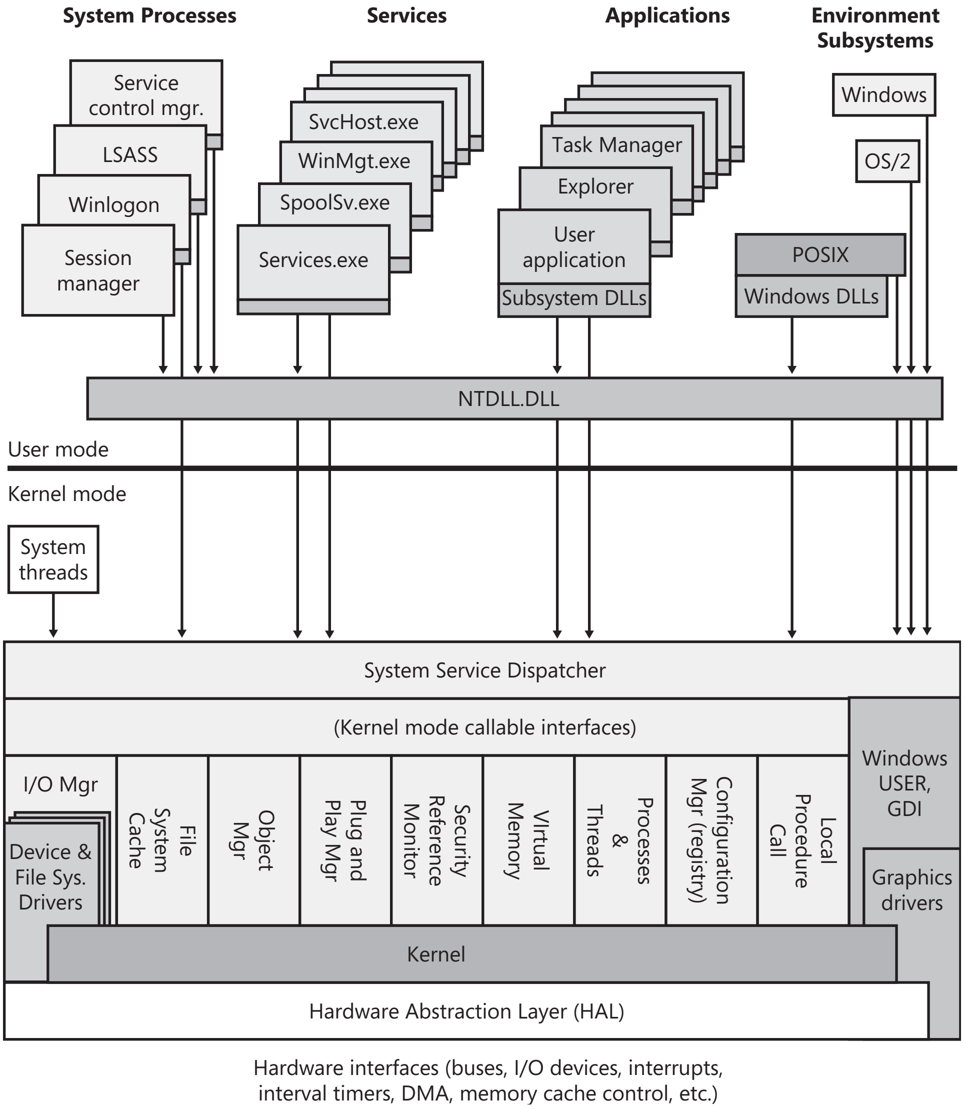


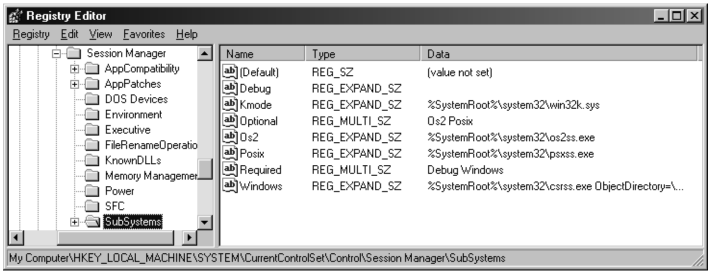
Figure 2-3 Windows architecture

Environment Subsystems and Subsystem DLLs

As shown in Figure 2-3, Windows originally had three environment subsystems: OS/2, POSIX, and Windows. As stated earlier, the OS/2 subsystem was removed in Windows 2000. Although the basic POSIX subsystem that originally shipped with Windows no longer ships with the system as of Windows XP, a greatly enhanced version is available for free as part of the Services for UNIX product.

As we'll explain shortly, of the three, the Windows subsystem is special in that Windows can't run without it. (It owns the keyboard, mouse, and display, and it is required to be present even on server systems with no interactive users logged in.) In fact, the other two subsystems are configured to start on demand, whereas the Windows subsystem must always be running.

The subsystem startup information is stored under the registry key HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\SubSystems. Figure 2-4 shows the values under this key.



Name	Type	Data
(Default)	REG_SZ	(value not set)
Debug	REG_EXPAND_SZ	
Kmode	REG_EXPAND_SZ	%SystemRoot%\system32\win32k.sys
Optional	REG_MULTI_SZ	Os2 Posix
Os2	REG_EXPAND_SZ	%SystemRoot%\system32\os2ss.exe
Posix	REG_EXPAND_SZ	%SystemRoot%\system32\psxss.exe
Required	REG_MULTI_SZ	Debug Windows
Windows	REG_EXPAND_SZ	%SystemRoot%\system32\csrss.exe ObjectDirectory=\...

My Computer\HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\SubSystems

Figure 2-4 Registry Editor showing Windows startup information

The Required value lists the subsystems that load when the system boots. The value has two strings: Windows and Debug. The Windows value contains the file specification of the Windows subsystem, Csrss.exe, which stands for Client/Server Run-Time Subsystem. (See the Note later in this section.) Debug is blank (because it's used for internal testing) and therefore does nothing. The Optional value indicates that the OS/2 and POSIX subsystems will be started on demand. The registry value Kmode contains the filename of the kernel-mode portion of the Windows subsystem, Win32k.sys (explained later in this chapter).

The role of an environment subsystem is to expose some subset of the base Windows executive system services to application programs. Each subsystem can provide access to different subsets of the native services in Windows. That means that some things can be done from an application built on one subsystem that can't be done by an application built on another subsystem. For example, a Windows application can't use the POSIX *fork* function.

Each executable image (.exe) is bound to one and only one subsystem. When an image is run, the process creation code examines the subsystem type code in the image header so that it can notify the proper subsystem of the new process. This type code is specified with the /SUBSYSTEM qualifier of the *link* command in Microsoft Visual C++ and can be viewed with the Exetype tool in the Windows resource kits.

Note As a historical note, the reason the Windows subsystem process is called Csrss.exe is that in the original design of Windows NT, all the subsystems were going to execute as threads inside a single systemwide environment subsystem process. When the POSIX and OS/2 subsystems were removed and put in their own processes, the filename for the Windows subsystem process wasn't changed.

Function calls can't be mixed between subsystems. In other words, a POSIX application can call only services exported by the POSIX subsystem, and a Windows application can call only services exported by the Windows subsystem. As you'll see later, this restriction is one reason why the original POSIX subsystem, which implements a very limited set of functions (only POSIX 1003.1), wasn't a useful environment for porting UNIX applications.

As mentioned earlier, user applications don't call Windows system services directly. Instead, they go through one or more subsystem DLLs. These libraries export the documented interface that the programs linked to that subsystem can call. For example, the Windows subsystem DLLs (such as Kernel32.dll, Advapi32.dll, User32.dll, and Gdi32.dll) implement the Windows API functions. The POSIX subsystem DLL (Psxdll.dll) implements the POSIX API functions.

EXPERIMENT: Viewing the Image Subsystem Type

You can see the image subsystem type by using either the Exetype tool in the Windows resource kits or the Dependency Walker tool (Depends.exe) in the Windows Support Tools and Platform SDK. For example, notice the image types for two different Windows images, Notepad.exe (the simple text editor) and Cmd.exe (the Windows command prompt):

```
C:\>exetype \Windows\system32\notepad.exe
File "\Windows\system32\notepad.exe" is of the following type:
    Windows NT
    32 bit machine
    Built for the Intel 80386 processor
    Runs under the Windows GUI subsystem

C:\>exetype \Windows\system32\cmd.exe
File "\Windows\system32\cmd.exe" is of the following type:
    Windows NT
    32 bit machine
    Built for the Intel 80386 processor
    Runs under the Windows character-based subsystem
```

This shows that Notepad is a GUI program while Cmd is a console or character-based program. And although the output of the Exetype tool implies there are two different subsystems for GUI and character-based programs, there is just one Windows subsystem. Also, Windows isn't supported on the Intel 386 processor (or the 486 for that matter)—the text output by the Exetype program hasn't been updated.

When an application calls a function in a subsystem DLL, one of three things can occur:

- The function is entirely implemented in user mode inside the subsystem DLL. In other words, no message is sent to the environment subsystem process, and no Windows executive system services are called. The function is performed in user mode, and the results are returned to the caller. Examples of such functions include *GetCurrentProcess* (which always returns -1, a value that is defined to refer to the current process in all process-related functions) and *GetCurrentProcessId*. (The process ID doesn't change for a running process, so this ID is retrieved from a cached location, thus avoiding the need to call into the kernel.)
- The function requires one or more calls to the Windows executive. For example, the Windows *ReadFile* and *WriteFile* functions involve calling the underlying internal (and undocumented) Windows I/O system services *NtReadFile* and *NtWriteFile*, respectively.
- The function requires some work to be done in the environment subsystem process. (The environment subsystem processes, running in user mode, are responsible for maintaining the state of the client applications running under their control.) In this case, a client/server request is made to the environment subsystem via a message sent to the subsystem to perform some operation. The subsystem DLL then waits for a reply before returning to the caller.

Some functions can be a combination of the second and third items just listed, such as the Windows *CreateProcess* and *CreateThread* functions.

Although Windows was designed to support multiple, independent environment subsystems, from a practical perspective, having each subsystem implement all the code to handle windowing and display I/O would result in a large amount of duplication of system functions that, ultimately, would have negatively affected both system size and performance. Because Windows was the primary subsystem, the Windows designers decided to locate these basic functions there and have the other subsystems call on the Windows subsystem to perform display I/O. Thus, the POSIX and OS/2 subsystems call services in the Windows subsystem to perform display I/O. (In fact, if you examine the subsystem type for these images, you'll see that they are Windows executables.)

Let's take a closer look at each of the environment subsystems.

Windows Subsystem

The Windows subsystem consists of the following major components:

- The environment subsystem process (Csrss.exe) contains support for:
  - ❑ Console (text) windows
  - ❑ Creating and deleting processes and threads
  - ❑ Portions of the support for 16-bit virtual DOS machine (VDM) processes

    - Other miscellaneous functions, such as *GetTempFile*, *DefineDosDevice*, *ExitWindowsEx*, and several natural language support functions
- The kernel-mode device driver (Win32k.sys) contains:
    - The window manager, which controls window displays; manages screen output; collects input from keyboard, mouse, and other devices; and passes user messages to applications.
    - The Graphics Device Interface (GDI), which is a library of functions for graphics output devices. It includes functions for line, text, and figure drawing and for graphics manipulation.
- Subsystem DLLs (such as Kernel32.dll, Advapi32.dll, User32.dll, and Gdi32.dll) translate documented Windows API functions into the appropriate and mostly undocumented kernel-mode system service calls to Ntoskrnl.exe and Win32k.sys.
- Graphics device drivers are hardware-dependent graphics display drivers, printer drivers, and video miniport drivers.

Applications call the standard USER functions to create user interface controls, such as windows and buttons, on the display. The window manager communicates these requests to the GDI, which passes them to the graphics device drivers, where they are formatted for the display device. A display driver is paired with a video miniport driver to complete video display support.

The GDI provides a set of standard two-dimensional functions that let applications communicate with graphics devices without knowing anything about the devices. GDI functions mediate between applications and graphics devices such as display drivers and printer drivers. The GDI interprets application requests for graphic output and sends the requests to graphics display drivers. It also provides a standard interface for applications to use varying graphics output devices. This interface enables application code to be independent of the hardware devices and their drivers. The GDI tailors its messages to the capabilities of the device, often dividing the request into manageable parts. For example, some devices can understand directions to draw an ellipse; others require the GDI to interpret the command as a series of pixels placed at certain coordinates. For more information about the graphics and video driver architecture, see the "Design Guide" section of the book *Graphics Drivers* in the Windows DDK.

Prior to Windows NT 4, the window manager and graphics services were part of the user-mode Windows subsystem process. In Windows NT 4, the bulk of the windowing and graphics code was moved from running in the context of the Windows subsystem process to a set of callable services running in kernel mode (in the file Win32k.sys). The primary reason for this shift was to improve overall system performance. Having a separate server process that contains the Windows graphics subsystem required multiple thread and process context switches, which consumed considerable CPU cycles and memory resources even though the original design was highly optimized.

For example, for each thread on the client side there was a dedicated, paired server thread in the Windows subsystem process waiting on the client thread for requests. A special interprocess communication facility called *fast LPC* was used to send messages between these threads. Unlike normal thread context switches, transitions between paired threads via fast LPC don't cause a rescheduling event in the kernel, thereby enabling the server thread to run for the remaining time slice of the client thread before having to take its turn in the kernel's preemptive thread scheduler. Moreover, shared memory buffers were used to allow fast passing of large data structures, such as bitmaps, and clients had direct but read-only access to key server data structures to minimize the need for thread/process transitions between clients and the Windows server. Also, GDI operations were (and still are) batched. *Batching* means that a series of graphics calls by a Windows application aren't "pushed" over to the server and drawn on the output device until a GDI batching queue is filled. You can set the size of the queue by using the Windows *GdiSetBatchLimit* function, and you can flush the queue at any time with *GdiFlush*. Conversely, read-only properties and data structures of GDI, once they were obtained from the Windows subsystem process, were cached on the client side for fast subsequent access.

Despite these optimizations, however, the overall system performance was still not adequate for graphics-intensive applications. The obvious solution was to eliminate the need for the additional threads and resulting context switches by moving the windowing and graphics system into kernel mode. Also, once applications have called into the window manager and the GDI, those subsystems can access other Windows executive components directly without the cost of user-mode or kernel-mode transitions. This direct access is especially important in the case of the GDI calling through video drivers, a process that involves interaction with video hardware at high frequencies and high bandwidths.

So, what remains in the user-mode process part of the Windows subsystem? All the drawing and updating for console or text windows are handled by it because console applications have no notion of repainting a window. It's easy to see this activity—simply open a command prompt and drag another window over it, and you'll see the Windows subsystem consuming CPU time as it repaints the console window. But other than console window support, only a few Windows functions result in sending a message to the Windows subsystem process anymore: process and thread creation and termination, network drive letter mapping, and creation of temporary files. In general, a running Windows application won't be causing many, if any, context switches to the Windows subsystem process.

Is Windows Less Stable with USER and GDI in Kernel Mode?

Some people wondered whether moving this much code into kernel mode would substantially affect system stability. The reason the impact on system stability has been minimal is that prior to Windows NT 4 (and this is still true today), a bug (such as an access violation) in the user-mode Windows subsystem process (Csrss.exe) results in a system crash because the Windows subsystem process was (and still is) a vital process to the running of the system. Because it was the process that contained the data structures that described the windows on the display, the death of that process would kill the user interface. However, even a Windows system operating as a server, with no interactive processes, can't run without this process, because server processes might be using window messaging to drive the internal state of the application. With Windows, an access violation in the same code now running in kernel mode simply crashes the system more quickly, because exceptions in kernel mode result in a system crash.

There is, however, one additional theoretical danger that didn't exist prior to moving the windowing and graphics system into kernel mode. Because this body of code is now running in kernel mode, a bug (such as the use of a bad pointer) could result in corrupting kernel-mode protected data structures. Prior to Windows NT 4, such references would have caused an access violation because kernel-mode pages aren't writable from user mode. But a system crash would have then resulted, as described earlier. With the code now running in kernel mode, a bad pointer reference that caused a write operation to some kernel-mode page might not immediately cause a system crash, but if it corrupted some data structure, a crash would likely result soon after. There is a small chance, however, that such a reference could corrupt a memory buffer (rather than a data structure), possibly resulting in returning corrupt data to a user program or writing bad data to the disk.

Another area of possible impact can come from the move of the graphics drivers into kernel mode. Previously, some portions of a graphics driver ran within Csrss and others ran in kernel mode. Now, the entire driver runs in kernel mode. Although Microsoft doesn't develop all the graphics device drivers supported in Windows, it does work directly with hardware manufacturers to help ensure that they are able to produce reliable and efficient drivers. All drivers shipped with the system are submitted to the same rigorous testing as other executive components.

Finally, it's important to understand that this design (running the windowing and graphics subsystem in kernel mode) is not fundamentally risky. It is identical to the approaches many other device drivers use (for example, network card drivers and hard disk drivers). All these drivers have been operating in kernel mode since the inception of Windows NT with a high degree of reliability.

Some people speculated that the move of the window manager and the GDI into kernel mode would hurt the preemptive multitasking capability of Windows. The theory was that with all the additional Windows processing time spent in kernel mode, other threads would have less opportunity to be run preemptively. This view was based on a misunderstanding of the Windows architecture. It is true that in many other nominally preemptive operating systems, executing in kernel mode is never preempted by the operating system scheduler—or is preempted only at a certain limited number of predefined points of kernel reentrancy. In Windows, however, threads running anywhere in the executive are preempted and scheduled alongside threads running in user mode, and all code within the executive is fully reentrant. Among other reasons, this capability is necessary to achieve a high degree of system scalability on SMP hardware.

Another line of speculation was that SMP scaling would be hurt by this change. The theory went like this: Previously, an interaction between an application and the window manager or the GDI involved two threads, one in the application and one in Csrss.exe. Therefore, on an SMP system, the two threads could run in parallel, thus improving throughput. This analysis shows a misunderstanding of how Windows NT technology worked prior to Windows NT 4. In most cases, calls from a client application to the Windows subsystem process run synchronously; that is, the client thread entirely blocks waiting on the server thread and begins to run again only when the server thread has completed the call. Therefore, no parallelism on SMP hardware can ever be achieved. This phenomenon is easily observable with a busy graphics application using the Performance tool on an SMP system. The observer will discover that on a two-processor system each processor is approximately 50 percent loaded, and it's relatively easy to find the single Csrss thread that is paired off with the busy application thread. Indeed, because the two threads are fairly intimate with each other and sharing state, the processors' caches must be flushed constantly to maintain coherency. This constant flushing is the reason that with Windows NT 3.51 a single-threaded graphics application typically runs slightly slower on an SMP machine than on a single processor system.

As a result, the changes in Windows NT 4 increased SMP throughput of applications that make heavy use of the window manager and the GDI, especially when more than one application thread is busy. When two application threads are busy on a two-processor Windows NT 3.51–based machine, a total of four threads (two in the application plus two in Csrss) are battling for time on the two processors. Although only two are typically ready to run at any given time, the lack of a consistent pattern in which threads run results in a loss of locality of reference and cache coherency. This loss occurs because the busy threads are likely to get shuffled from one processor to another. In the Windows NT 4 design, each of the two application threads essentially has its own processor, and the automatic thread affinity of Windows tends to run the same thread on the same processor indefinitely, thus maximizing locality of reference and minimizing the need to synchronize the private per-processor memory caches.

So in summary, moving the window manager and the GDI from user mode to kernel mode has provided improved performance without any significant decrease in system stability or reliability, even in the case of multiple sessions being created in a Terminal Service enabled configuration.

POSIX Subsystem

POSIX, an acronym loosely defined as "a portable operating system interface based on UNIX," refers to a collection of international standards for UNIX-style operating system interfaces. The POSIX standards encourage vendors implementing UNIX-style interfaces to make them compatible so that programmers can move their applications easily from one system to another.

Windows implements only one of the many POSIX standards, POSIX.1, formally known as ISO/IEC 9945-1:1990 or IEEE POSIX standard 1003.1-1990. This standard was included primarily to meet U.S. government procurement requirements set in the mid-to-late 1980s that mandated POSIX.1 compliance as specified in Federal Information Processing Standard (FIPS) 151-2, developed by the National Institute of Standards and Technology. Windows NT 3.5, 3.51, and 4 have been formally tested and certified according to FIPS 151-2.

Because POSIX.1 compliance was a mandatory goal for Windows, the operating system was designed to ensure that the required base system support was present to allow for the implementation of a POSIX.1 subsystem (such as the *fork* function, which is implemented in the Windows executive, and the support for hard file links in the Windows file system). However, because POSIX.1 defines a limited set of services (such as process control, interprocess communication, simple character cell I/O, and so on), the POSIX subsystem that comes with Windows 2000 isn't a complete programming environment. And because applications can't mix calls between subsystems on Windows, by default, POSIX applications are limited to the strict set of services defined in POSIX.1. This restriction means that a POSIX executable on Windows can't create a thread or a window or use remote procedure calls (RPCs) or sockets.

To address this limitation, Microsoft provides a product called Windows Services for Unix, which includes (as of version 3.5) an enhanced POSIX subsystem environment that provides nearly 2000 UNIX functions and 300 UNIX-like tools and utilities. (See *http://www.microsoft.com/windows/sfu/default.asp* for more information on Windows Services for Unix.)

This enhanced POSIX subsystem assists in porting UNIX applications to Windows. However, because the programs are still linked as POSIX executables, they cannot call Windows functions. To port UNIX applications to Windows and allow the use of Windows functions, you can purchase UNIX-to-Windows porting packages, such as the MKS Toolkit products available from Mortice Kern Systems Inc. (*www.mkssoftware.com*). With this approach, a UNIX application can be recompiled and relinked as a Windows executable and can slowly start to integrate calls to native Windows functions.

EXPERIMENT: Watching the POSIX Subsystem Start

The POSIX subsystem is configured by default to start the first time a POSIX executable is run, so you can watch it start by running a POSIX program, such as one of the POSIX utilities that comes with the Windows Services for Unix. (You can also find a small set of POSIX utilities in the \Apps\POSIX folder on the Windows 2000 resource kit tools media—they are not installed as part of the resource kit tools installation.) Follow these steps to watch the POSIX subsystem start:

1. Start a command prompt.
2. Run Process Explorer and check that the POSIX subsystem isn't already running (that is, that there's no Psxss.exe process on the system). Make sure Process Explorer is displaying the process list in tree view (by pressing Ctrl+T).
3. Run a POSIX program, such as the C Shell or Korn Shell included with Windows Services for Unix (or a POSIX tool from the Windows 2000 resource kit, such as \Apps\POSIX\Ls.exe).
4. Go back to Process Explorer and notice the new Psxss.exe process that is a child of Smss.exe (which, depending on your different highlight duration, might still be highlighted as a new process on the display).

To compile and link a POSIX application in Windows requires the POSIX headers and libraries from the Platform SDK. POSIX executables are linked against the POSIX subsystem library, Psxdll.dll. Because by default Windows is configured to start the POSIX subsystem on demand, the first time you run a POSIX application, the POSIX subsystem process (Psxss.exe) must be started. It remains running until the system reboots. (If you kill the POSIX subsystem process, you won't be able to run more POSIX applications until you reboot.) The POSIX image itself isn't run directly—instead, a special support image called Posix.exe is launched, which in turn creates a child process to run the POSIX application.

OS/2 Subsystem

The OS/2 environment subsystem, like the built-in POSIX subsystem, is fairly limited in usefulness in that it supports only OS/2 1.2 16-bit character-based or video I/O (VIO) applications. Although Microsoft did sell a replacement OS/2 1.2 Presentation Manager subsystem for Windows NT 4, it didn't support OS/2 2.*x* (or later) applications (and it isn't available for Windows 2000 or later).

Also, because Windows doesn't allow direct hardware access by user applications, OS/2 programs that contain I/O privilege segments that attempt to perform IN/OUT instructions (to access some hardware device) as well as advanced video I/O (AVIO) aren't supported. Applications that use the CLI/STI instructions are supported—but all the other OS/2 applications in the system and all the other threads in the OS/2 process issuing the CLI instructions are suspended until an STI instruction is executed.

The 16-MB memory limitation on native OS/2 1.2 doesn't apply to Windows—the OS/2 subsystem uses the 32-bit virtual address space of Windows to provide up to 512 MB of memory to OS/2 1.2 applications, as illustrated in Figure 2-5.

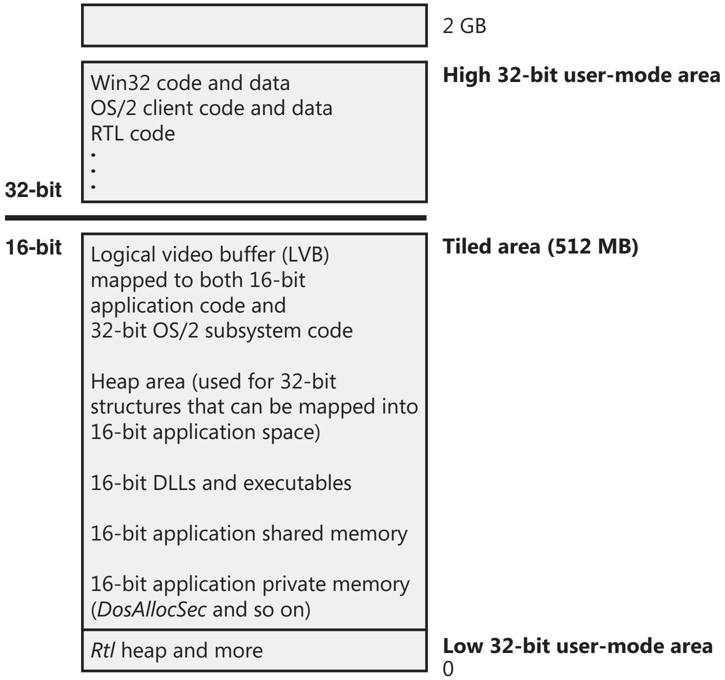

2 GB
High 32-bit user-mode area
Win32 code and data
OS/2 client code and data
RTL code
•
•
•
32-bit
16-bit
Tiled area (512 MB)
Logical video buffer (LVB) mapped to both 16-bit application code and 32-bit OS/2 subsystem code
Heap area (used for 32-bit structures that can be mapped into 16-bit application space)
16-bit DLLs and executables
16-bit application shared memory
16-bit application private memory (*DosAllocSec* and so on)
Rtl heap and more
Low 32-bit user-mode area
0

Figure 2-5 OS/2 subsystem virtual memory layout

The tiled area is 512 MB of virtual address space that is reserved up front and then committed or decommitted when 16-bit applications need segments. The OS/2 subsystem maintains a local descriptor table (LDT) for each process, with shared memory segments at the same LDT slot for all OS/2 processes.

As we'll discuss in detail in Chapter 6, threads are the elements of a program that execute, and as such they must be scheduled for processor time. Although Windows priority levels range from 0 through 31, the 64 OS/2 priority levels (0 through 63) are mapped to Windows dynamic priorities 1 through 15. OS/2 threads never receive Windows real-time priorities 16 through 31.

As with the POSIX subsystem, the OS/2 subsystem starts automatically the first time you activate a compatible OS/2 image. It remains running until the system is rebooted.

For more information on how Windows handles running POSIX and OS/2 applications, see the section "Flow of *CreateProcess*" in Chapter 6.

Ntdll.dll

Ntdll.dll is a special system support library primarily for the use of subsystem DLLs. It contains two types of functions:

- System service dispatch stubs to Windows executive system services
- Internal support functions used by subsystems, subsystem DLLs, and other native images

The first group of functions provides the interface to the Windows executive system services that can be called from user mode. There are more than 200 such functions, such as *NtCreateFile*, *NtSetEvent*, and so on. As noted earlier, most of the capabilities of these functions are accessible through the Windows API. (A number are not, however, and are for use within the operating system.)

For each of these functions, Ntdll contains an entry point with the same name. The code inside the function contains the architecture-specific instruction that causes a transition into kernel mode to invoke the system service dispatcher (explained in more detail in Chapter 3), which after verifying some parameters, calls the actual kernel-mode system service that contains the real code inside Ntoskrnl.exe.

Ntdll also contains many support functions, such as the image loader (functions that start with *Ldr*), the heap manager, and Windows subsystem process communication functions (functions that start with *Csr*), as well as general run-time library routines (functions that start with *Rtl*). It also contains the user-mode asynchronous procedure call (APC) dispatcher and exception dispatcher. (APCs and exceptions are explained in Chapter 3.)

Executive

The Windows executive is the upper layer of Ntoskrnl.exe. (The kernel is the lower layer.) The executive includes the following types of functions:

- Functions that are exported and callable from user mode. These functions are called *system services* and are exported via Ntdll. Most of the services are accessible through the Windows API or the APIs of another environment subsystem. A few services, however, aren't available through any documented subsystem function. (Examples include LPCs and various query functions such as *NtQueryInformationProcess*, specialized functions such as *NtCreatePagingFile*, and so on.)
- Device driver functions that are called through the use of the *DeviceIoControl* function. This provides a general interface from user mode to kernel mode to call functions in device drivers that are not associated with a read or write.
- Functions that can be called only from kernel mode that are exported and documented in the Windows DDK or Windows Installable File System (IFS) Kit. (For information on the Windows IFS Kit, go to *http://www.microsoft.com/whdc/ddk/ifskit/default.mspx*.)

- Functions that are exported and callable from kernel mode but are not documented in the Windows DDK or IFS Kit (such as the functions called by the boot video driver, which start with *Inbv*).
- Functions that are defined as global symbols but are not exported. These include internal support functions called within Ntoskrnl, such as those that start with *Iop* (internal I/O manager support functions) or *Mi* (internal memory management support functions).
- Functions that are internal to a module that are not defined as global symbols.

The executive contains the following major components, each of which is covered in detail in a subsequent chapter of this book:

- The *configuration manager* (explained in Chapter 4) is responsible for implementing and managing the system registry.
- The *process and thread manager* (explained in Chapter 6) creates and terminates processes and threads. The underlying support for processes and threads is implemented in the Windows kernel; the executive adds additional semantics and functions to these lower-level objects.
- The *security reference monitor* (or SRM, described in Chapter 8) enforces security policies on the local computer. It guards operating system resources, performing run-time object protection and auditing.
- The *I/O manager* (explained in Chapter 9) implements device-independent I/O and is responsible for dispatching to the appropriate device drivers for further processing.
- The *Plug and Play (PnP) manager* (explained in Chapter 9) determines which drivers are required to support a particular device and loads those drivers. It retrieves the hardware resource requirements for each device during enumeration. Based on the resource requirements of each device, the PnP manager assigns the appropriate hardware resources such as I/O ports, IRQs, DMA channels, and memory locations. It is also responsible for sending proper event notification for device changes (addition or removal of a device) on the system.
- The *power manager* (explained in Chapter 9) coordinates power events and generates power management I/O notifications to device drivers. When the system is idle, the power manager can be configured to reduce power consumption by putting the CPU to sleep. Changes in power consumption by individual devices are handled by device drivers but are coordinated by the power manager.
- The *WDM Windows Management Instrumentation routines* (explained in Chapter 4) enable device drivers to publish performance and configuration information and receive commands from the user-mode WMI service. Consumers of WMI information can be on the local machine or remote across the network.

- The *cache manager* (explained in Chapter 11) improves the performance of file-based I/O by causing recently referenced disk data to reside in main memory for quick access (and by deferring disk writes by holding the updates in memory for a short time before sending them to the disk). As you'll see, it does this by using the memory manager's support for mapped files.
- The *memory manager* (explained in Chapter 7) implements *virtual memory*, a memory management scheme that provides a large, private address space for each process that can exceed available physical memory. The memory manager also provides the underlying support for the cache manager.
- The *logical prefetcher* (explained in Chapter 7) accelerates system and process startup by optimizing the loading of data referenced during the startup of the system or a process.

In addition, the executive contains four main groups of support functions that are used by the executive components just listed. About a third of these support functions are documented in the DDK because device drivers also use them. These are the four categories of support functions:

- The *object manager*, which creates, manages, and deletes Windows executive objects and abstract data types that are used to represent operating system resources such as processes, threads, and the various synchronization objects. The object manager is explained in Chapter 3.
- The *LPC facility* (explained in Chapter 3) passes messages between a client process and a server process on the same computer. LPC is a flexible, optimized version of *remote procedure call* (RPC), an industry-standard communication facility for client and server processes across a network.
- A broad set of common *run-time library* functions, such as string processing, arithmetic operations, data type conversion, and security structure processing.
- Executive support routines, such as system memory allocation (paged and nonpaged pool), interlocked memory access, as well as two special types of synchronization objects: resources and fast mutexes.

Kernel

The kernel consists of a set of functions in Ntoskrnl.exe that provide fundamental mechanisms (such as thread scheduling and synchronization services) used by the executive components, as well as low-level hardware architecture-dependent support (such as interrupt and exception dispatching), that are different on each processor architecture. The kernel code is written primarily in C, with assembly code reserved for those tasks that require access to specialized processor instructions and registers not easily accessible from C.

Like the various executive support functions mentioned in the preceding section, a number of functions in the kernel are documented in the DDK (and can be found by searching for functions beginning with *Ke*) because they are needed to implement device drivers.

Kernel Objects

The kernel provides a low-level base of well-defined, predictable operating system primitives and mechanisms that allow higher-level components of the executive to do what they need to do. The kernel separates itself from the rest of the executive by implementing operating system mechanisms and avoiding policy making. It leaves nearly all policy decisions to the executive, with the exception of thread scheduling and dispatching, which the kernel implements.

Outside the kernel, the executive represents threads and other shareable resources as objects. These objects require some policy overhead, such as object handles to manipulate them, security checks to protect them, and resource quotas to be deducted when they are created. This overhead is eliminated in the kernel, which implements a set of simpler objects, called *kernel objects*, that help the kernel control central processing and support the creation of executive objects. Most executive-level objects encapsulate one or more kernel objects, incorporating their kernel-defined attributes.

One set of kernel objects, called *control objects*, establishes semantics for controlling various operating system functions. This set includes the APC object, the *deferred procedure call* (DPC) object, and several objects the I/O manager uses, such as the interrupt object.

Another set of kernel objects, known as *dispatcher objects*, incorporates synchronization capabilities that alter or affect thread scheduling. The dispatcher objects include the kernel thread, mutex (called *mutant* internally), event, kernel event pair, semaphore, timer, and waitable timer. The executive uses kernel functions to create instances of kernel objects, to manipulate them, and to construct the more complex objects it provides to user mode. Objects are explained in more detail in Chapter 3, and processes and threads are described in Chapter 6.

Hardware Support

The other major job of the kernel is to abstract or isolate the executive and device drivers from variations between the hardware architectures supported by Windows. This job includes handling variations in functions such as interrupt handling, exception dispatching, and multiprocessor synchronization.

Even for these hardware-related functions, the design of the kernel attempts to maximize the amount of common code. The kernel supports a set of interfaces that are portable and semantically identical across architectures. Most of the code that implements this portable interface is also identical across architectures.

Some of these interfaces are implemented differently on different architectures, however, or some of the interfaces are partially implemented with architecture-specific code. These architecturally independent interfaces can be called on any machine, and the semantics of the interface will be the same whether or not the code varies by architecture. Some kernel interfaces (such as spinlock routines, which are described in Chapter 3) are actually implemented in the HAL (described in the next section) because their implementation can vary for systems within the same architecture family.

The kernel also contains a small amount of code with x86-specific interfaces needed to support old MS-DOS programs. These x86 interfaces aren't portable in the sense that they can't be called on a machine based on any other architecture; they won't be present. This x86-specific code, for example, supports calls to manipulate global descriptor tables (GDTs) and LDTs, hardware features of the x86.

Other examples of architecture-specific code in the kernel include the interface to provide translation buffer and CPU cache support. This support requires different code for the different architectures because of the way caches are implemented.

Another example is context switching. Although at a high level the same algorithm is used for thread selection and context switching (the context of the previous thread is saved, the context of the new thread is loaded, and the new thread is started), there are architectural differences among the implementations on different processors. Because the context is described by the processor state (registers and so on), what is saved and loaded varies depending on the architecture.

Hardware Abstraction Layer

As mentioned at the beginning of this chapter, one of the crucial elements of the Windows design is its portability across a variety of hardware platforms. The hardware abstraction layer (HAL) is a key part of making this portability possible. The HAL is a loadable kernel-mode module (Hal.dll) that provides the low-level interface to the hardware platform on which Windows is running. It hides hardware-dependent details such as I/O interfaces, interrupt controllers, and multiprocessor communication mechanisms—any functions that are both architecture-specific and machine-dependent.

So rather than access hardware directly, Windows internal components as well as user-written device drivers maintain portability by calling the HAL routines when they need platform-dependent information. For this reason, the HAL routines are documented in the Windows DDK. To find out more about the HAL and its use by device drivers, refer to the DDK.

Although several HALs are included with Windows (as shown in Table 2-6), only one is chosen at installation time and copied to the system disk with the filename Hal.dll. (Other operating systems, such as VMS, select the equivalent of the HAL at system boot time.) Therefore, you can't assume that a system disk from one x86 installation will boot on a different processor if the HAL that supports the other processor is different.

Table 2-6 List of x86 HALs in \Windows\Driver Cache\i386\Driver.cab

HAL File Name	Systems Supported
Hal.dll	Standard PCs
Halacpi.dll	Advanced Configuration and Power Interface (ACPI) PCs
Halapic.dll	Advanced Programmable Interrupt Controller (APIC) PCs
Halaacpi.dll	APIC ACPI PCs
Halmps.dll	Multiprocessor PCs
Halmacpi.dll	Multiprocessor ACPI PCs
Halborg.dll	Silicon Graphics Workstation (Windows 2000 only; platform no longer marketed)
Halsp.dll	Compaq SystemPro (Windows XP only)

Note As of Windows Server 2003, no vendor-specific HALs are shipped with the base system.

EXPERIMENT: Viewing the Base HALs Included with Windows

To view the HALs included with Windows, open the file Driver.cab in the appropriate architecture-specific folder underneath \Windows\Driver Cache. (For example, for x86 systems, the file name is \Windows\Driver Cache\i386\Driver.cab.) Scroll down to the files beginning with “Hal” and you should see the files listed in Table 2-6.

EXPERIMENT: Determining Which HAL You’re Running

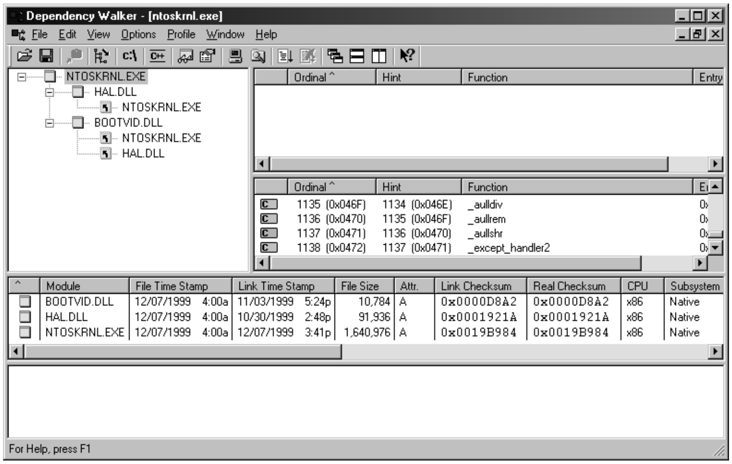
There are two ways to determine which HAL you’re running:

1. Open the file \Windows\Repair\Setup.log, search for *Hal.dll*, and look at the filename after the equals sign. This is the name of the HAL on the distribution media extracted from Driver.cab.
2. In Device Manager (right-click on the My Computer icon on your desktop, select Properties, click on the Hardware tab, and then click Device Manager), look at the name of the “driver” under the Computer device type. For example, the following screen shot is from a system running the ACPI HAL:

EXPERIMENT: Viewing NTOSKRNL and HAL Image Dependencies

You can view the relationship of the kernel and HAL images by examining their export and import tables using the Dependency Walker tool (Depends.exe), which is contained in the Windows Support Tools and the Platform SDK. To examine an image in the Dependency Walker, select Open from the File menu to open the desired image file.

Here is a sample of output you can see by viewing the dependencies of Ntoskrnl using this tool:


Dependency Walker - [ntoskrnl.exe]
File Edit View Options Profile Window Help
For Help, press F1

```
NTOSKRNL.EXE
    HAL.DLL
        NTOSKRNL.EXE
    BOOTVID.DLL
        NTOSKRNL.EXE
        HAL.DLL
```

	Ordinal ^	Hint	Function	Entry

	Ordinal ^	Hint	Function	E
C	1135 (0x046F)	1134 (0x046E)	_aulldiv	0x
C	1136 (0x0470)	1135 (0x046F)	_aullrem	0x
C	1137 (0x0471)	1136 (0x0470)	_aullshr	0x
C	1138 (0x0472)	1137 (0x0471)	_except_handler2	0x

^	Module	File Time Stamp	Link Time Stamp	File Size	Attr.	Link Checksum	Real Checksum	CPU	Subsystem
□	BOOTVID.DLL	12/07/1999 4:00a	11/03/1999 5:24p	10,784	A	0x0000D8A2	0x0000D8A2	x86	Native
□	HAL.DLL	12/07/1999 4:00a	10/30/1999 2:48p	91,936	A	0x0001921A	0x0001921A	x86	Native
□	NTOSKRNL.EXE	12/07/1999 4:00a	12/07/1999 3:41p	1,640,976	A	0x0019B984	0x0019B984	x86	Native

Notice that Ntoskrnl is linked against the HAL, which is in turn linked against Ntoskrnl. (They both use functions in each other.) Ntoskrnl is also linked against Bootvid.dll, the boot video driver that is used to implement the GUI startup screen. On Windows XP and later, you will see an additional DLL, Kdcom.dll, in the list. This contains kernel debugger infrastructure code that used to be part of Ntoskrnl.exe.

For a detailed description of the information displayed by this tool, see the Dependency Walker help file (Depends.hlp).

Device Drivers

Although device drivers are explained in detail in Chapter 9, this section provides a brief overview of the types of drivers and explains how to list the drivers installed and loaded on your system.

Device drivers are loadable kernel-mode modules (typically ending in .sys) that interface between the I/O manager and the relevant hardware. They run in kernel mode in one of three contexts:

- In the context of the user thread that initiated an I/O function
- In the context of a kernel-mode system thread
- As a result of an interrupt (and therefore not in the context of any particular process or thread—whichever process or thread was current when the interrupt occurred)

As stated in the preceding section, device drivers in Windows don't manipulate hardware directly, but rather they call functions in the HAL to interface with the hardware. Drivers are typically written in C (sometimes C++) and therefore, with proper use of HAL routines, can be source code portable across the CPU architectures supported by Windows and binary portable within an architecture family.

There are several types of device drivers:

- *Hardware device drivers* manipulate hardware (using the HAL) to write output to or retrieve input from a physical device or network. There are many types of hardware device drivers, such as bus drivers, human interface drivers, mass storage drivers, and so on.
- *File system drivers* are Windows drivers that accept file-oriented I/O requests and translate them into I/O requests bound for a particular device.
- *File system filter drivers*, such as those that perform disk mirroring and encryption, intercept I/Os and perform some added-value processing before passing the I/O to the next layer.
- *Network redirectors and servers* are file system drivers that transmit file system I/O requests to a machine on the network and receive such requests, respectively.
- *Protocol drivers* implement a networking protocol such as TCP/IP, NetBEUI, and IPX/SPX.
- *Kernel streaming filter drivers* are chained together to perform signal processing on data streams, such as recording or displaying audio and video.

Because installing a device driver is the only way to add user-written kernel-mode code to the system, some programmers have written device drivers simply as a way to access internal operating system functions or data structures that are not accessible from user mode (but that are documented and supported in the DDK). For example, many of the utilities from *www.sysinternals.com* combine a Windows GUI application and a device driver that is used to gather internal system state and call kernel-mode-only accessible functions not accessible from the user-mode Windows API.

Windows Driver Model (WDM)

Windows 2000 added support for Plug and Play, Power Options, and an extension to the Windows NT driver model called the Windows Driver Model (WDM). Windows 2000 and later can run legacy Windows NT 4 drivers, but because these don't support Plug and Play and Power Options, systems running these drivers will have reduced capabilities in these two areas.

From the WDM perspective, there are three kinds of drivers:

- A *bus driver* services a bus controller, adapter, bridge, or any device that has child devices. Bus drivers are required drivers, and Microsoft generally provides them; each type of bus (such as PCI, PCMCIA, and USB) on a system has one bus driver. Third parties can write bus drivers to provide support for new buses, such as VMEbus, Multibus, and Futurebus.
- A *function driver* is the main device driver and provides the operational interface for its device. It is a required driver unless the device is used raw (an implementation in which I/O is done by the bus driver and any bus filter drivers, such as SCSI PassThru). A function driver is by definition the driver that knows the most about a particular device, and it is usually the only driver that accesses device-specific registers.
- A *filter driver* is used to add functionality to a device (or existing driver) or to modify I/O requests or responses from other drivers (and is often used to fix hardware that provides incorrect information about its hardware resource requirements). Filter drivers are optional and can exist in any number, placed above or below a function driver and above a bus driver. Usually, system original equipment manufacturers (OEMs) or independent hardware vendors (IHVs) supply filter drivers.

In the WDM driver environment, no single driver controls all aspects of a device: a bus driver is concerned with reporting the devices on its bus to the PnP manager, while a function driver manipulates the device.

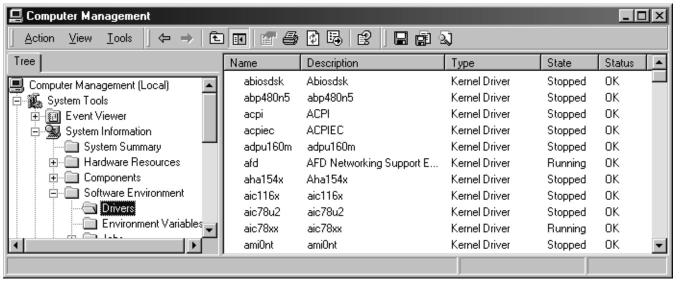
In most cases, lower-level filter drivers modify the behavior of device hardware. For example, if a device reports to its bus driver that it requires four I/O ports when it actually requires 16 I/O ports, a lower-level device-specific function filter driver could intercept the list of hardware resources reported by the bus driver to the PnP manager, and update the count of I/O ports.

Upper-level filter drivers usually provide added-value features for a device. For example, an upper-level device filter driver for a keyboard can enforce additional security checks.

Interrupt processing is explained in Chapter 3. Further details about the I/O manager, WDM, Plug and Play, and Power Options are included in Chapter 9.

EXPERIMENT: Viewing the Installed Device Drivers

You can list the installed drivers by running Computer Management. (From the Start menu, select Programs, Administrative Tools, and then Computer Management; or from Control Panel, open Administrative Tools and select Computer Management.) From within Computer Management, expand System Information and then Software Environment, and open Drivers. Here's an example output of the list of installed drivers:



Name	Description	Type	State	Status
abiosdsk	Abiosdsk	Kernel Driver	Stopped	OK
abp480n5	abp480n5	Kernel Driver	Stopped	OK
acpi	ACPI	Kernel Driver	Stopped	OK
acpiec	ACPIEC	Kernel Driver	Stopped	OK
adpu160m	adpu160m	Kernel Driver	Stopped	OK
afd	AFD Networking Support E...	Kernel Driver	Running	OK
aha154x	Aha154x	Kernel Driver	Stopped	OK
aic116x	aic116x	Kernel Driver	Stopped	OK
aic78u2	aic78u2	Kernel Driver	Stopped	OK
aic78xx	aic78xx	Kernel Driver	Running	OK
ami0nt	ami0nt	Kernel Driver	Stopped	OK

This window displays the list of device drivers defined in the registry, their type, and their state (Running or Stopped). Device drivers and Windows service processes are both defined in the same place: HKLM\ SYSTEM\CurrentControlSet\Services. However, they are distinguished by a type code—for example, type 1 is a kernel-mode device driver. (For a complete list of the information stored in the registry for device drivers, see Table 4-7.)

Alternatively, you list the currently loaded device drivers with the Drivers utility (Drivers.exe in the Windows 2000 resource kits) or the Pstat utility (Pstat.exe in the Windows XP Support Tools, Windows Server 2003 Support Tools, Windows 2000 resource kits, and the Platform SDK). Here is a partial output from the Drivers utility:

```
C:\>drivers
  ModuleName   Code   Data  Bss   Paged    Init         LinkDate
----------------------------------------------------------------
ntoskrnl.exe 429184  96896   0  775360  138880  Tue Dec 07 18:41:11 1999
     hal.dll  25856   6016   0   16160   10240  Tue Nov 02 20:14:22 1999
 BOOTVID.DLL   5664   2464   0       0     320  Wed Nov 03 20:24:33 1999
    ACPI.sys  92096   8960   0   43488    4448  Wed Nov 10 20:06:04 1999
  WMILIB.SYS    512      0   0    1152     192  Sat Sep 25 14:36:47 1999
     pci.sys  12704   1536   0   31264    4608  Wed Oct 27 19:11:08 1999
  isapnp.sys  14368    832   0   22944    2048  Sat Oct 02 16:00:35 1999
compbatt.sys   2496      0   0    2880    1216  Fri Oct 22 18:32:49 1999
   BATTC.SYS    800      0   0    2976     704  Sun Oct 10 19:45:37 1999
intelide.sys   1760     32   0       0     128  Thu Oct 28 19:20:03 1999
 PCIIDEX.SYS   4544    480   0   10944    1632  Wed Oct 27 19:02:19 1999
  pcmcia.sys  32800   8864   0   23680    6240  Fri Oct 29 19:20:08 1999
  ftdisk.sys   4640     32   0   95072    3392  Mon Nov 22 14:36:23 1999
----------------------------------------------------------------
      Total 4363360  580320  0 3251424  432992
```

Each loaded kernel-mode component (Ntoskrnl, the HAL, as well as device drivers) is shown, along with the sizes of the sections in each image.

The Pstat utility also shows the loaded driver list, but only after it first displays the process list and the threads in each process. Pstat includes one important piece of information that the Drivers utility doesn't: the load address of the module in system space. As we'll explain later, this address is needed to map running system threads to the device driver in which they exist.

Peering into Undocumented Interfaces

Examining the names of the exported or global symbols in key system images (such as Ntoskrnl.exe, Hal.dll, or Ntdll.dll) can be enlightening—you can get an idea of the kinds of things Windows can do versus what happens to be documented and supported today. Of course, just because you know the names of these functions doesn't mean that you can or should call them—the interfaces are undocumented and are subject to change. We suggest that you look at these functions purely to gain more insight into the kinds of internal functions Windows performs, not to bypass supported interfaces.

For example, looking at the list of functions in Ntdll.dll gives you the list of all the system services that Windows provides to user-mode subsystem DLLs versus the subset that each subsystem exposes. Although many of these functions map clearly to documented and supported Windows functions, several are not exposed via the Windows API. (See the article "Inside the Native API" from *www.sysinternals.com*.)

Conversely, it's also interesting to examine the imports of Windows subsystem DLLs (such as Kernel32.dll or Advapi32.dll) and which functions they call in Ntdll.

Another interesting image to dump is Ntoskrnl.exe—although many of the exported routines that kernel-mode device drivers use are documented in the Windows DDK, quite a few are not. You might also find it interesting to take a look at the import table for Ntoskrnl and the HAL; this table shows the list of functions in the HAL that Ntoskrnl uses and vice versa.

Table 2-7 lists most of the commonly used function name prefixes for the executive components. Each of these major executive components also uses a variation of the prefix to denote internal functions—either the first letter of the prefix followed by an *i* (for *internal*) or the full prefix followed by a *p* (for *private*). For example, *Ki* represents internal kernel functions, and *Psp* refers to internal process support functions.

Table 2-7 Commonly Used Prefixes

Prefix	Component
Cc	Cache manager
Cm	Configuration manager
Ex	Executive support routines
FsRtl	File system driver run-time library
Hal	Hardware abstraction layer
Io	I/O manager
Ke	Kernel
Lpc	Local procedure call
Lsa	Local security authentication
Mm	Memory manager
Nt	Windows system services (most of which are exported as Windows functions)
Ob	Object manager
Po	Power manager
Pp	PnP manager
Ps	Process support
Rtl	Run-time library
Se	Security
Wmi	Windows Management Instrumentation
Zw	Mirror entry point for system services (beginning with *Nt*) that sets previous access mode to kernel, which eliminates parameter validation, because *Nt* system services validate parameters only if previous access mode is user

You can decipher the names of these exported functions more easily if you understand the naming convention for Windows system routines. The general format is:

<Prefix><Operation><Object>

In this format, *Prefix* is the internal component that exports the routine, *Operation* tells what is being done to the object or resource, and *Object* identifies what is being operated on.

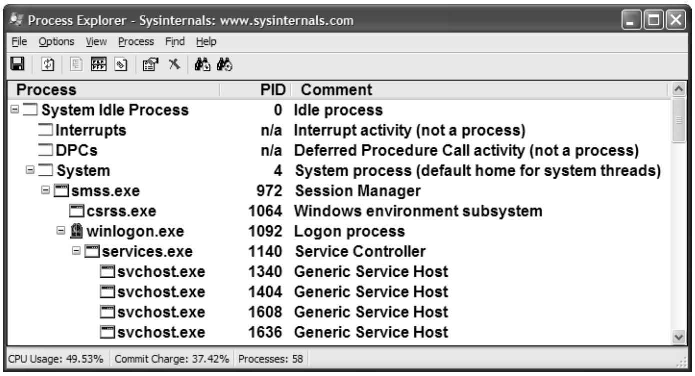
For example, *ExAllocatePoolWithTag* is the executive support routine to allocate from a paged or nonpaged pool. *KeInitializeThread* is the routine that allocates and sets up a kernel thread object.

System Processes

The following system processes appear on every Windows system. (Two of these—Idle and System—are not full processes, as they are not running a user-mode executable.)

- Idle process (contains one thread per CPU to account for idle CPU time)
- System process (contains the majority of the kernel-mode system threads)
- Session manager (Smss.exe)
- Windows subsystem (Csrss.exe)
- Logon process (Winlogon.exe)
- Service control manager (Services.exe) and the child service processes it creates (such as the system-supplied generic service host process, Svchost.exe)
- Local security authentication server (Lsass.exe)

To understand the relationship of these processes, it is helpful to view the process "tree"—that is, the parent/child relationship between processes. Seeing which process created each process helps to understand where each process comes from. Figure 2-6 is a partial screen snapshot of the process tree with comments put on the first few system processes. (Process Explorer allows you to add a comment for individual processes and optionally display that as a column on the display.)



```
Process                        PID   Comment
⊟ System Idle Process            0   Idle process
    Interrupts                 n/a   Interrupt activity (not a process)
    DPCs                       n/a   Deferred Procedure Call activity (not a process)
  ⊟ System                       4   System process (default home for system threads)
    ⊟ smss.exe                 972   Session Manager
        csrss.exe             1064   Windows environment subsystem
      ⊟ winlogon.exe          1092   Logon process
        ⊟ services.exe        1140   Service Controller
            svchost.exe       1340   Generic Service Host
            svchost.exe       1404   Generic Service Host
            svchost.exe       1608   Generic Service Host
            svchost.exe       1636   Generic Service Host
CPU Usage: 49.53% | Commit Charge: 37.42% | Processes: 58
```

Figure 2-6 Initial System Process Tree

The next sections explain the key system processes shown in Figure 2-6. Although these sections briefly indicate the order of process startup, Chapter 5 contains a detailed description of the steps involved in booting and starting Windows.

Idle Process

The first process listed in Figure 2-6 is the system idle process. As we'll explain in Chapter 6, processes are identified by their image name. However, this process (as well as the process named System) isn't running a real user-mode image (in that there is no "System Idle Process.exe" in the \Windows directory). In addition, the name shown for this process differs from utility to utility (because of implementation details). Table 2-8 lists several of the names given to the Idle process (process ID 0). The Idle process is explained in detail in Chapter 6.

Table 2-8 Names for Process ID 0 in Various Utilities

Utility	Name for Process ID 0
Task Manager	System Idle Process
Process Viewer (Pviewer.exe)	Idle
Process Status (Pstat.exe)	Idle Process
Process Explode (Pview.exe)	System Process
Task List (Tlist.exe)	System Process
QuickSlice (Qslice.exe)	Systemprocess

Now let's look at system threads and the purpose of each of the system processes that are running real images.

Interrupts and DPCs

The two lines labeled Interrupts and DPCs represent time spent servicing interrupts and deferred procedure calls. These mechanisms are explained in Chapter 3. Note that while Process Explorer displays these as entries in the process list, they are not processes. They are shown because they account for CPU time not charged to any process. (For example, a system with heavy interrupt activity will not appear as a process consuming CPU time.) Note that Task Manager includes interrupt and DPC time in the system idle time. Thus a system with heavy interrupt activity will appear to be idle when using Task Manager.

System Process and System Threads

The System process (process ID 8 in Windows 2000 and process ID 4 in Windows XP and Windows Server 2003) is the home for a special kind of thread that runs only in kernel mode: a *kernel-mode system thread*. System threads have all the attributes and contexts of regular user-mode threads (such as a hardware context, priority, and so on) but are different in that they run only in kernel-mode executing code loaded in system space, whether that is in Ntoskrnl.exe or in any other loaded device driver. In addition, system threads don't have a user process address space and hence must allocate any dynamic storage from operating system memory heaps, such as a paged or nonpaged pool.

System threads are created by the *PsCreateSystemThread* function (documented in the DDK), which can be called only from kernel mode. Windows as well as various device drivers create system threads during system initialization to perform operations that require thread context, such as issuing and waiting for I/Os or other objects or polling a device. For example, the memory manager uses system threads to implement such functions as writing dirty pages to the page file or mapped files, swapping processes in and out of memory, and so forth. The kernel creates a system thread called the *balance set manager* that wakes up once per second to possibly initiate various scheduling and memory management–related events. The cache manager also uses system threads to implement both read-ahead and write-behind I/Os. The file server device driver (Srv.sys) uses system threads to respond to network I/O requests for file data on disk partitions shared to the network. Even the floppy driver has a system thread to poll the floppy device. (Polling is more efficient in this case because an interrupt-driven floppy driver consumes a large amount of system resources.) Further information on specific system threads is included in the chapters in which the component is described.

By default, system threads are owned by the System process, but a device driver can create a system thread in any process. For example, the Windows subsystem device driver (Win32k.sys) creates system threads in the Windows subsystem process (Csrss.exe) so that they can easily access data in the user-mode address space of that process.

When you're troubleshooting or going through a system analysis, it's useful to be able to map the execution of individual system threads back to the driver or even to the subroutine that contains the code. For example, on a heavily loaded file server, the System process will likely be consuming considerable CPU time. But the knowledge that when the System process is running "some system thread" is running isn't enough to determine which device driver or operating system component is running.

So if threads in the System process are running, first determine which ones are running (for example, with the Performance tool). Once you find the thread (or threads) that is running, look up in which driver the system thread began execution (which at least tells you which driver likely created the thread) or examine the call stack (or at least the current address) of the thread in question, which would indicate where the thread is currently executing.

Both of these techniques are illustrated in the following experiments.

EXPERIMENT: Identifying System Threads in the System Process

You can see that the threads inside the System process must be kernel-mode system threads because the start address for each thread is greater than the start address of system space (which by default begins at 0x80000000, unless the system was booted with the /3GB Boot.ini switch). Also, if you look at the CPU time for these threads, you'll see that those that have accumulated any CPU time have run only in kernel mode.

To find out which driver created the system thread, look up the start address of the thread (which you can display with Pviewer.exe) and look for the driver whose base address is closest to (but before) the start address of the thread. Both the Pstat utility (at the end of its output) as well as the *!drivers* kernel debugger command list the base address of each loaded device driver.

To quickly find the current address of the thread, use the *!stacks 0* command in the kernel debugger. Here is sample output from a live system (using LiveKd):

```
kd> !stacks 0
Proc.Thread  Thread  ThreadState  Blocker
                                    [System]
 8.000004  8146edb0  BLOCKED  ntoskrnl!MmZeroPageThread+0x5f
 8.00000c  8146e730  BLOCKED  ?? Kernel stack not resident ??
 8.000010  8146e4b0  BLOCKED  ntoskrnl!ExpWorkerThread+0x73
 8.000014  8146d030  BLOCKED  ?? Kernel stack not resident ??
 8.000018  8146ddb0  BLOCKED  ntoskrnl!ExpWorkerThread+0x73
 8.00001c  8146db30  BLOCKED  ntoskrnl!ExpWorkerThread+0x73
 8.000020  8146d8b0  BLOCKED  ntoskrnl!ExpWorkerThread+0x73
 8.000024  8146d630  BLOCKED  ntoskrnl!ExpWorkerThread+0x73
 8.000028  8146d3b0  BLOCKED  ntoskrnl!ExpWorkerThread+0x73
 8.00002c  8146c030  BLOCKED  ntoskrnl!ExpWorkerThread+0x73
 8.000030  8146cdb0  BLOCKED  ntoskrnl!ExpWorkerThreadBalanceManager+0x55
 8.000034  8146b470  BLOCKED  ntoskrnl!MiDereferenceSegmentThread+0x44
 8.000038  8146b1f0  BLOCKED  ntoskrnl!MiModifiedPageWriterWorker+0x31
 8.00003c  8146a030  BLOCKED  ntoskrnl!KeBalanceSetManager+0x7e
 8.000040  8146adb0  BLOCKED  ntoskrnl!KeSwapProcessOrStack+0x24
 8.000044  8146a5b0  BLOCKED  ntoskrnl!FsRtlWorkerThread+0x33
 8.000048  8146a330  BLOCKED  ntoskrnl!FsRtlWorkerThread+0x33
 8.00004c  81461030  BLOCKED  ACPI!ACPIWorker+0x46
 8.000050  8143a770  BLOCKED  ntoskrnl!MiMappedPageWriter+0x4d
 8.000054  81439730  BLOCKED  dmio!voliod_loop+0x399
 8.000058  81436c90  BLOCKED  NDIS!ndisWorkerThread+0x22
 8.00005c  813d9170  BLOCKED  ltmdmntt!WakeupTimerThread+0x27
 8.000060  813d8030  BLOCKED  ltmdmntt!WriteRegistryThread+0x1c
 8.000070  8139c850  BLOCKED  raspptp!MainPassiveLevelThread+0x78
 8.000074  8139c5d0  BLOCKED  raspptp!PacketWorkingThread+0xc0
 8.00006c  81384030  BLOCKED  rasacd!AcdNotificationRequestThread+0xd8
 8.000080  81333330  BLOCKED  rdbss!RxpWorkerThreadDispatcher+0x6f
 8.000084  813330b0  BLOCKED  rdbss!RxSpinUpRequestsDispatcher+0x58
 8.00008c  81321db0  BLOCKED  ?? Kernel stack not resident ??
 8.00015c  81205570  BLOCKED  INO_FLTR+0x68bd
 8.000160  81204570  BLOCKED  INO_FLTR+0x80e9
 8.000178  811fcdb0  BLOCKED  irda!RxThread+0xfa
 8.0002d0  811694f0  BLOCKED  ?? Kernel stack not resident ??
 8.0002d4  81168030  BLOCKED  ?? Kernel stack not resident ??
 8.000404  811002b0  BLOCKED  rdbss!RxpWorkerThreadDispatcher+0x6f
 8.000430  810f4990  READY    parallel!ParallelThread+0x3e
 8.00069c  80993030  READY    rdbss!RxpWorkerThreadDispatcher+0x6f
```

The first column is the process ID and thread ID (in the form "process ID.thread ID"). The second column is the current address of the thread. The third column indicates whether the thread is in a wait state, ready state, or running state. (See Chapter 6 for a description of thread states.) The last column is the top-most address on the thread's stack. The information in this last column makes it easy to see which driver each thread started in. For the threads in Ntoskrnl, the name of the function gives a further indication of what the thread is doing.

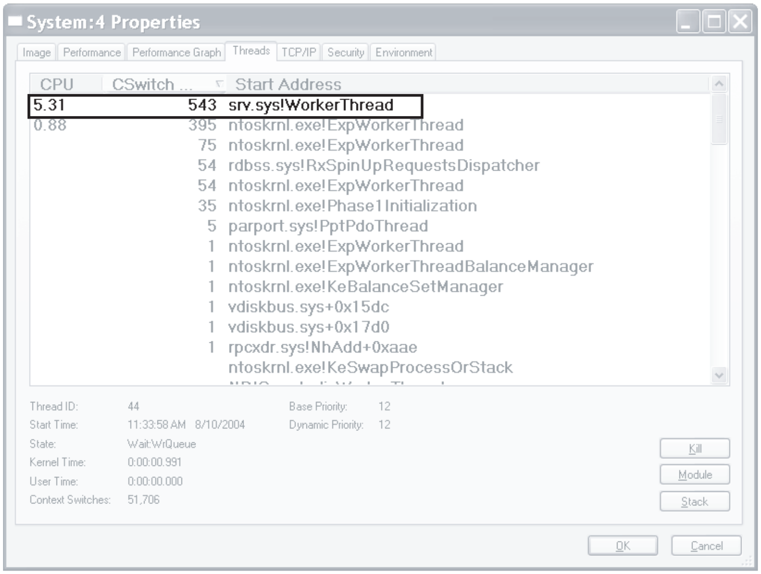
However, if the thread running is one of the system worker threads (ExpWorkerThread), you still don't really know what the thread is doing because any device driver can submit work to a system worker thread. Therefore, the only way to trace back worker thread activity is to set a breakpoint at ExQueueWorkItem. When you reach the breakpoint, type **!dso**

work_queue_item esp+4. This command will dump the first argument to ExQueueWorkItem (a work queue structure), which in turn contains the address of the worker routine to be called in the context of the worker thread. Alternatively, you can look at the caller by using the *k* command in the kernel debugger, which displays the current call stack. The current call stack will show the driver that is queuing the work to the worker thread (as opposed to the routine to be called from the worker thread).

EXPERIMENT: Mapping a System Thread to a Device Driver

In this experiment, we'll see how to map CPU activity in the System process to the responsible system thread (and the driver it falls in) generating the activity. This is important because when the System process is running, you must go to the thread granularity to really understand what's going on. For this experiment, we will generate system thread activity by generating file server activity on your machine. (The file server driver, Srv.sys, creates system threads to handle inbound requests for file I/O. See Chapter 13 for more information on this component.)

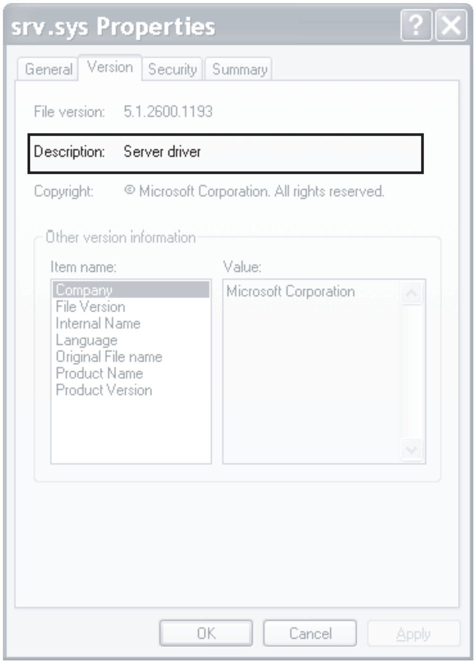
1. Open a command prompt.
2. Do a directory listing of your entire C drive using a network path to access your C drive. For example, if your computer name is COMPUTER1, type "**dir \\computer1\c$ /s**". (The /s switch lists all subdirectories.)
3. Run Process Explorer, and double-click on the System process.
4. Click on the Threads tab.
5. Sort by the CSwitch Delta (context switch delta) column. You should see one or more threads in Srv.sys running such as the following:



CPU	CSwitch ...	Start Address
5.31	543	srv.sys!WorkerThread
0.88	395	ntoskrnl.exe!ExpWorkerThread
	75	ntoskrnl.exe!ExpWorkerThread
	54	rdbss.sys!RxSpinUpRequestsDispatcher
	54	ntoskrnl.exe!ExpWorkerThread
	35	ntoskrnl.exe!Phase1Initialization
	5	parport.sys!PptPdoThread
	1	ntoskrnl.exe!ExpWorkerThread
	1	ntoskrnl.exe!ExpWorkerThreadBalanceManager
	1	ntoskrnl.exe!KeBalanceSetManager
	1	vdiskbus.sys+0x15dc
	1	vdiskbus.sys+0x17d0
	1	rpcxdr.sys!NhAdd+0xaae
		ntoskrnl.exe!KeSwapProcessOrStack

Thread ID: 44
Start Time: 11:33:58 AM 8/10/2004
State: Wait:WrQueue
Kernel Time: 0:00:00.991
User Time: 0:00:00.000
Context Switches: 51,706
Base Priority: 12
Dynamic Priority: 12

If you see a system thread running and you are not sure what the driver is, press the Module button, which will bring up the file properties. Pressing the Module button while highlighting the thread in Srv.sys previously shown results in the following display:



srv.sys Properties

File version: 5.1.2600.1193

Description: Server driver

Copyright: © Microsoft Corporation. All rights reserved.

Other version information

Item name: Company

Value: Microsoft Corporation

Session Manager (Smss)

The Session Manager (\Windows\System32\Smss.exe) is the first user-mode process created in the system. The kernel-mode system thread that performs the final phase of the initialization of the executive and kernel creates the actual Smss process.

The Session Manager is responsible for a number of important steps in starting Windows, such as opening additional page files, performing delayed file rename and delete operations, and creating system environment variables. It also launches the subsystem processes (normally just Csrss.exe) and the Winlogon process, which in turn creates the rest of the system processes.

Much of the configuration information in the registry that drives the initialization steps of Smss can be found under HKLM\SYSTEM\CurrentControlSet\Control\Session Manager. Some of these are explained in Chapter 5 in the section on Smss. (For a more complete description of the keys and values, see the Registry Entries help file, Regentry.chm, in the Windows 2000 resource kits.)

After performing these initialization steps, the main thread in Smss waits forever on the process handles to Csrss and Winlogon. If either of these processes terminates unexpectedly, Smss crashes the system (using the crash code STATUS_SYSTEM_PROCESS_TERMINATED, or 0xC000021A), because Windows relies on their existence. Meanwhile, Smss waits for requests to load subsystems, debug events, and requests to create new terminal server sessions. (For a description of terminal services, see the section "Terminal Services and Multiple Sessions" in Chapter 1.)

Terminal Services session creation is performed by Smss. When a request comes in to Smss to create a session, it first calls NtSetSystemInformation with a request to set up kernel-mode session data structures. This in turn calls the internal memory manager function **MmSessionCreate**, which sets up the session virtual address space that will contain the session paged pool and the per-session data structures allocated by the kernel-mode part of the Win32 subsystem (Win32k.sys) and other session-space device drivers. (See Chapter 7 for more details.) Smss then creates an instance of Winlogon and Csrss for the session.

Winlogon, LSASS and Userinit

The Windows logon process (\Windows\System32\Winlogon.exe) handles interactive user logons and logoffs. Winlogon is notified of a user logon request when the *secure attention sequence* (SAS) keystroke combination is entered. The default SAS on Windows is the combination Ctrl+Alt+Delete. The reason for the SAS is to protect users from password-capture programs that simulate the logon process, because this keyboard sequence cannot be intercepted by a user mode application.

The identification and authentication aspects of the logon process are implemented in a replaceable DLL named GINA (Graphical Identification and Authentication). The standard Windows GINA, Msgina.dll, implements the default Windows logon interface. However, developers can provide their own GINA DLL to implement other identification and authentication mechanisms in place of the standard Windows username/password method (such as one based on a voice print). In addition, Winlogon can load additional network provider DLLs that need to perform secondary authentication. This capability allows multiple network providers to gather identification and authentication information all at one time during normal logon.

Once the username and password have been captured, they are sent to the local security authentication server process (\Windows\System32\Lsass.exe, described in Chapter 8) to be authenticated. LSASS calls the appropriate authentication package (implemented as a DLL) to perform the actual verification, such as checking whether a password matches what is stored in the active directory or the SAM (the part of the registry that contains the definition of the users and groups).

Upon a successful authentication, LSASS calls a function in the security reference monitor (for example, *NtCreateToken*) to generate an access token object that contains the user's security profile. This access token is then used by Winlogon to create the initial process(es) in the user's session. The initial process(es) are stored in the registry value *Userinit* under the registry key HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon. (The default is Userinit.exe, but there can be more than one image in the list.)

Userinit performs some initialization of the user environment (such as running the login script and applying group policies) and then looks in the registry at the *Shell* value (under the same Winlogon key referred to previously) and creates a process to run the system-defined shell (by default, Explorer.exe). Then Userinit exits. This is the reason Explorer.exe is shown with no parent—its parent has exited, and as explained earlier, tlist left-justifies processes whose parent isn't running. (Another way of looking at it is that Explorer is the grandchild of Winlogon.)

Winlogon is active not only during user logon and logoff but also whenever it intercepts the SAS from the keyboard. For example, when you press Ctrl+Alt+Delete while logged in, the Windows Security dialog box comes up, providing the options to log off, start the Task Manager, lock the workstation, shut down the system, and so forth. Winlogon is the process that handles this interaction.

For a complete description of the steps involved in the logon process, see the section "Smss, Csrss, and Winlogon" in Chapter 5. For more details on security authentication, see Chapter 8. For details on the callable functions that interface with LSASS (the functions that start with **Lsa**), see the documentation in the Platform SDK.

Service Control Manager (SCM)

Recall from earlier in the chapter that "services" on Windows can refer either to a server process or to a device driver. This section deals with services that are user-mode processes. Services are like UNIX "daemon processes" or VMS "detached processes" in that they can be configured to start automatically at system boot time without requiring an interactive logon. They can also be started manually (such as by running the Services administrative tool or by calling the Windows *StartService* function). Typically, services do not interact with the logged-on user, although there are special conditions when this is possible. (See Chapter 4.)

The service control manager is a special system process running the image \Windows\System32\Services.exe that is responsible for starting, stopping, and interacting with service processes. Service programs are really just Windows images that call special Windows functions to interact with the service control manager to perform such actions as registering the service's successful startup, responding to status requests, or pausing or shutting down the service. Services are defined in the registry under HKLM\SYSTEM\CurrentControlSet \Services. The resource kit Registry Entries help file (Regentry.chm) documents the subkeys and values for services.

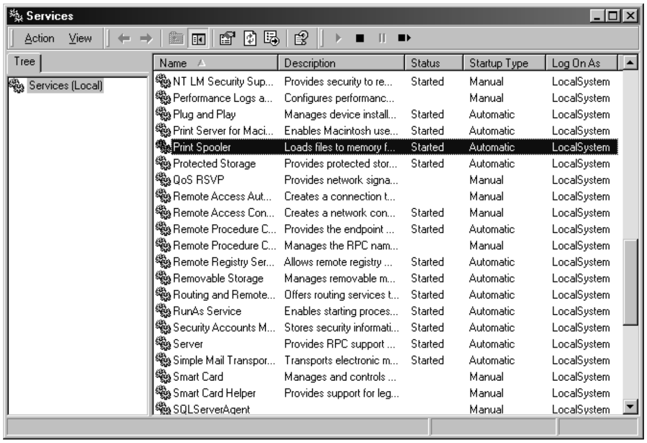
Keep in mind that services have three names: the process name you see running on the system, the internal name in the registry, and the display name shown in the Services administrative tool. (Not all services have a display name—if a service doesn't have a display name, the internal name is shown.) With Windows, services can also have a description field that further details what the service does.

To map a service process to the services contained in that process, use the *tlist /s* command. Note that there isn't always one-to-one mapping between service process and running services, however, because some services share a process with other services. In the registry, the type code indicates whether the service runs in its own process or shares a process with other services in the image.

A number of Windows components are implemented as services, such as the Spooler, Event Log, Task Scheduler, and various networking components.

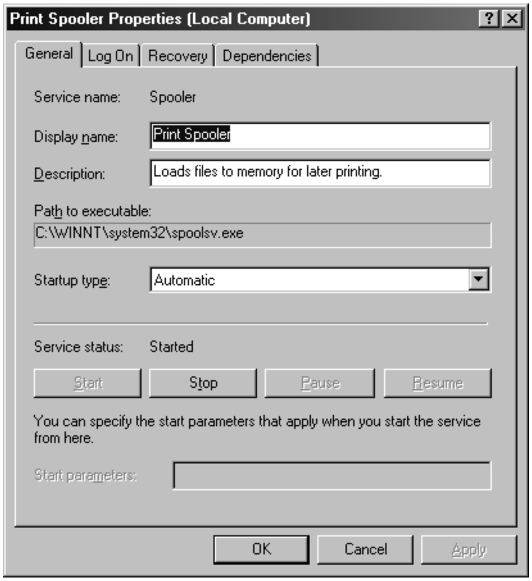
EXPERIMENT: Listing Installed Services

To list the installed services, select Administrative Tools from Control Panel, and then select Services. You should see output like this:



Name	Description	Status	Startup Type	Log On As
NT LM Security Sup...	Provides security to re...	Started	Manual	LocalSystem
Performance Logs a...	Configures performanc...		Manual	LocalSystem
Plug and Play	Manages device install...	Started	Automatic	LocalSystem
Print Server for Maci...	Enables Macintosh use...	Started	Automatic	LocalSystem
Print Spooler	Loads files to memory f...	Started	Automatic	LocalSystem
Protected Storage	Provides protected stor...	Started	Automatic	LocalSystem
QoS RSVP	Provides network signa...		Manual	LocalSystem
Remote Access Aut...	Creates a connection t...		Manual	LocalSystem
Remote Access Con...	Creates a network con...	Started	Manual	LocalSystem
Remote Procedure C...	Provides the endpoint ...	Started	Automatic	LocalSystem
Remote Procedure C...	Manages the RPC nam...		Manual	LocalSystem
Remote Registry Ser...	Allows remote registry ...	Started	Automatic	LocalSystem
Removable Storage	Manages removable m...	Started	Automatic	LocalSystem
Routing and Remote...	Offers routing services t...	Started	Automatic	LocalSystem
RunAs Service	Enables starting proces...	Started	Automatic	LocalSystem
Security Accounts M...	Stores security informati...	Started	Automatic	LocalSystem
Server	Provides RPC support ...	Started	Automatic	LocalSystem
Simple Mail Transpor...	Transports electronic m...	Started	Automatic	LocalSystem
Smart Card	Manages and controls ...		Manual	LocalSystem
Smart Card Helper	Provides support for leg...		Manual	LocalSystem
SQLServerAgent			Manual	LocalSystem

To see the detailed properties about a service, right-click on a service and select Properties. For example, here are the properties for the Print Spooler service (highlighted in the previous figure):



Service name: Spooler

Display name: Print Spooler

Description: Loads files to memory for later printing.

Path to executable:
C:\WINNT\system32\spoolsv.exe

Startup type: Automatic

Service status: Started

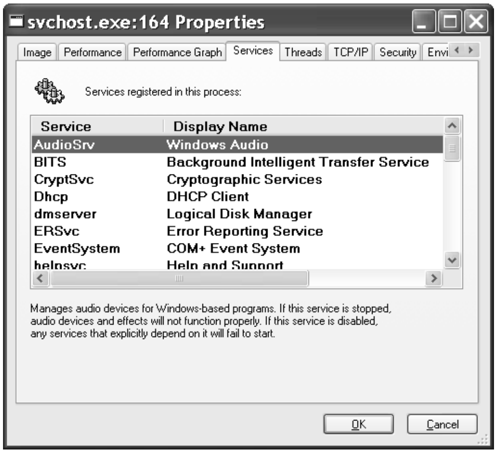
You can specify the start parameters that apply when you start the service from here.

Start parameters:

Notice that the Path To Executable field identifies the program that contains this service. Remember that some services share a process with other services—mapping isn't always one to one.

EXPERIMENT: Viewing Service Details Inside Service Processes

Process Explorer highlights processes hosting one service or more. (You can configure this by selecting the Configure Highlighting entry in the Options menu.) If you double-click on a service-hosting process, you will see a Services tab that lists the services inside the process: the name of the registry key that defines the service, the display name seen by the administrator, and the description text for that service (if present). For example, listing the services in a Svchost.exe process on Windows XP running under the System account looks like this:



svchost.exe:164 Properties

Services registered in this process:

Service	Display Name
AudioSrv	Windows Audio
BITS	Background Intelligent Transfer Service
CryptSvc	Cryptographic Services
Dhcp	DHCP Client
dmserver	Logical Disk Manager
ERSvc	Error Reporting Service
EventSystem	COM+ Event System
helpsvc	Help and Support

Manages audio devices for Windows-based programs. If this service is stopped, audio devices and effects will not function properly. If this service is disabled, any services that explicitly depend on it will fail to start.

For more details on services, see Chapter 4.

Conclusion

In this chapter, we've taken a broad look at the overall system architecture of Windows. We've examined the key components of Windows and seen how they interrelate. In the next chapter, we'll look in more detail at the core system mechanisms that these components are built on, such as the object manager and synchronization.

Chapter 3
System Mechanisms

Microsoft Windows provides several base mechanisms that kernel-mode components such as the executive, the kernel, and device drivers use. This chapter explains the following system mechanisms and describes how they are used:

- Trap dispatching, including interrupts, deferred procedure calls (DPCs), asynchronous procedure calls (APCs), exception dispatching, and system service dispatching
- The executive object manager
- Synchronization, including spinlocks, kernel dispatcher objects, and how waits are implemented
- System worker threads
- Miscellaneous mechanisms such as Windows global flags
- Local procedure calls (LPCs)
- Kernel Event Tracing
- Wow64

Trap Dispatching

Interrupts and exceptions are operating system conditions that divert the processor to code outside the normal flow of control. Either hardware or software can detect them. The term *trap* refers to a processor's mechanism for capturing an executing thread when an exception or an interrupt occurs and transferring control to a fixed location in the operating system. In Windows, the processor transfers control to a *trap handler*, a function specific to a particular interrupt or exception. Figure 3-1 illustrates some of the conditions that activate trap handlers.

The kernel distinguishes between interrupts and exceptions in the following way. An *interrupt* is an asynchronous event (one that can occur at any time) that is unrelated to what the processor is executing. Interrupts are generated primarily by I/O devices, processor clocks, or timers, and they can be enabled (turned on) or disabled (turned off). An *exception*, in contrast, is a synchronous condition that results from the execution of a particular instruction. Running a program a second time with the same data under the same conditions can reproduce exceptions. Examples of exceptions include memory access violations, certain debugger instructions, and divide-by-zero errors. The kernel also regards system service calls as exceptions (although technically they're system traps).

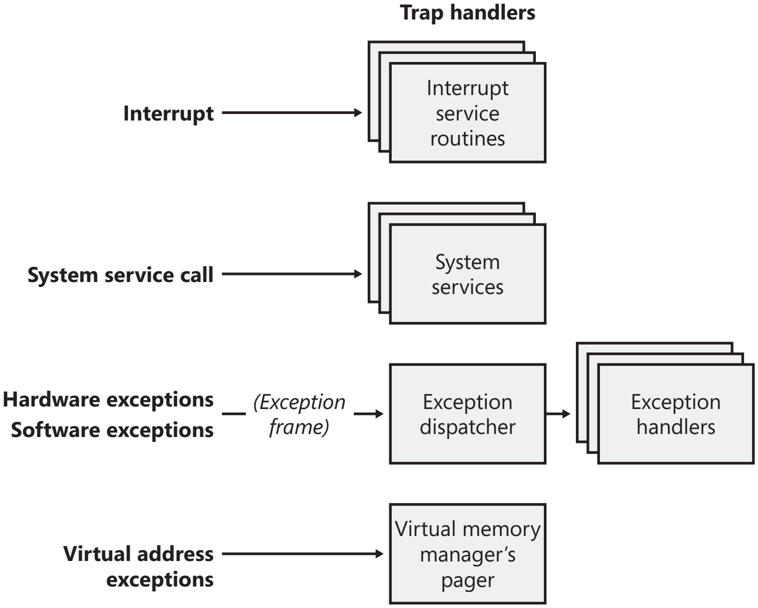


Figure 3-1 Trap dispatching

Either hardware or software can generate exceptions and interrupts. For example, a bus error exception is caused by a hardware problem, whereas a divide-by-zero exception is the result of a software bug. Likewise, an I/O device can generate an interrupt, or the kernel itself can issue a software interrupt (such as an APC or DPC, described later in this chapter).

When a hardware exception or interrupt is generated, the processor records enough machine state on the kernel stack of the thread that's interrupted so that it can return to that point in the control flow and continue execution as if nothing had happened. If the thread was executing in user mode, Windows switches to the thread's kernel-mode stack. Windows then creates a *trap frame* on the kernel stack of the interrupted thread into which it stores the execution state of the thread. The trap frame is a subset of a thread's complete context, and you can view its definition by typing **dt nt!_ktrap_frame** in the kernel debugger. (Thread context is described in Chapter 6.) The kernel handles software interrupts either as part of hardware interrupt handling or synchronously when a thread invokes kernel functions related to the software interrupt.

In most cases, the kernel installs front-end trap handling functions that perform general trap handling tasks before and after transferring control to other functions that field the trap. For example, if the condition was a device interrupt, a kernel hardware interrupt trap handler transfers control to the *interrupt service routine* (ISR) that the device driver provided for the interrupting device. If the condition was caused by a call to a system service, the general system service trap handler transfers control to the specified system service function in the executive. The kernel also installs trap handlers for traps that it doesn't expect to see or doesn't handle. These trap handlers typically execute the system function *KeBugCheckEx*, which halts the computer when the kernel detects problematic or incorrect behavior that, if left unchecked, could result in data corruption. (For more information on bug checks, see Chapter 14.) The following sections describe interrupt, exception, and system service dispatching in greater detail.

Interrupt Dispatching

Hardware-generated interrupts typically originate from I/O devices that must notify the processor when they need service. Interrupt-driven devices allow the operating system to get the maximum use out of the processor by overlapping central processing with I/O operations. A thread starts an I/O transfer to or from a device and then can execute other useful work while the device completes the transfer. When the device is finished, it interrupts the processor for service. Pointing devices, printers, keyboards, disk drives, and network cards are generally interrupt driven.

System software can also generate interrupts. For example, the kernel can issue a software interrupt to initiate thread dispatching and to asynchronously break into the execution of a thread. The kernel can also disable interrupts so that the processor isn't interrupted, but it does so only infrequently—at critical moments while it's processing an interrupt or dispatching an exception, for example.

The kernel installs interrupt trap handlers to respond to device interrupts. Interrupt trap handlers transfer control either to an external routine (the ISR) that handles the interrupt or to an internal kernel routine that responds to the interrupt. Device drivers supply ISRs to service device interrupts, and the kernel provides interrupt handling routines for other types of interrupts.

In the following subsections, you'll find out how the hardware notifies the processor of device interrupts, the types of interrupts the kernel supports, the way device drivers interact with the kernel (as a part of interrupt processing), and the software interrupts the kernel recognizes (plus the kernel objects that are used to implement them).

Hardware Interrupt Processing

On the hardware platforms supported by Windows, external I/O interrupts come into one of the lines on an interrupt controller. The controller in turn interrupts the processor on a single line. Once the processor is interrupted, it queries the controller to get the *interrupt request* (IRQ). The interrupt controller translates the IRQ to an interrupt number, uses this number as an index into a structure called the *interrupt dispatch table* (IDT), and transfers control to the appropriate interrupt dispatch routine. At system boot time, Windows fills in the IDT with pointers to the kernel routines that handle each interrupt and exception.

EXPERIMENT: Viewing the IDT

You can view the contents of the IDT, including information on what trap handlers Windows has assigned to interrupts (including exceptions and IRQs), using the *!idt* kernel debugger command. The *!idt* command with no flags shows vectors that map to addresses in modules other than Ntoskrnl.exe.

The following example shows what the output of the *!idt* command looks like:

```
kd> !idt

Dumping IDT:

30:     806b14c0 hal!HalpClockInterrupt
31:     8a39dc3c i8042prt!I8042KeyboardInterruptService (KINTERRUPT 8a39dc00)
34:     8a436dd4 serial!SerialCIsrSw (KINTERRUPT 8a436d98)
35:     8a44ed74 NDIS!ndisMIsr (KINTERRUPT 8a44ed38)
                 portcls!CInterruptSync::Release+0x10 (KINTERRUPT 899c44a0)
38:     806abe80 hal!HalpProfileInterrupt
39:     8a4a8abc ACPI!ACPIInterruptServiceRoutine (KINTERRUPT 8a4a8a80)
3b:     8a48d8c4 pcmcia!PcmciaInterrupt (KINTERRUPT 8a48d888)
                 ohci1394!OhciIsr (KINTERRUPT 8a41da18)
                 VIDEOPRT!pVideoPortInterrupt (KINTERRUPT 8a1bc2c0)
                 USBPORT!USBPORT_InterruptService (KINTERRUPT 8a2302b8)
                 USBPORT!USBPORT_InterruptService (KINTERRUPT 8a0b8008)
                 USBPORT!USBPORT_InterruptService (KINTERRUPT 8a170008)
                 USBPORT!USBPORT_InterruptService (KINTERRUPT 8a258380)
                 NDIS!ndisMIsr (KINTERRUPT 8a0e0430)
3c:     8a39d3ec i8042prt!I8042MouseInterruptService (KINTERRUPT 8a39d3b0)
3e:     8a47264c atapi!IdePortInterrupt (KINTERRUPT 8a472610)
3f:     8a489b3c atapi!IdePortInterrupt (KINTERRUPT 8a489b00)
```

On the system used to provide the output for this experiment, the keyboard device driver's (I8042prt.sys) keyboard ISR is at interrupt number 0x3C and several devices—including the video adapter, PCMCIA bus, USB and IEEE 1394 ports, and network adapter—share interrupt 0x3B.

Windows maps hardware IRQs to interrupt numbers in the IDT, and the system also uses the IDT to configure trap handlers for exceptions. For example, the x86 and x64 exception number for a page fault (an exception that occurs when a thread attempts to access a page of virtual memory that isn't defined or present) is 0xe. Thus, entry 0xe in the IDT points to the system's page fault handler. Although the architectures supported by Windows allow up to 256 IDT entries, the number of IRQs a particular machine can support is determined by the design of the interrupt controller the machine uses.

Each processor has a separate IDT so that different processors can run different ISRs, if appropriate. For example, in a multiprocessor system, each processor receives the clock interrupt, but only one processor updates the system clock in response to this interrupt. All the processors, however, use the interrupt to measure thread quantum and to initiate rescheduling when a thread's quantum ends. Similarly, some system configurations might require that a particular processor handle certain device interrupts.

x86 Interrupt Controllers

Most x86 systems rely on either the i8259A Programmable Interrupt Controller (PIC) or a variant of the i82489 Advanced Programmable Interrupt Controller (APIC); the majority of new computers include an APIC. The PIC standard originates with the original IBM PC. PICs work only with uniprocessor systems and have 15 interrupt lines. APICs and SAPICs (discussed shortly) work with multiprocessor systems and have 256 interrupt lines. Intel and other companies have defined the Multiprocessor Specification (MP Specification), a design standard for x86 multiprocessor systems that centers on the use of APIC. To provide compatibility with uniprocessor operating systems and boot code that starts a multiprocessor system in uniprocessor mode, APICs support a PIC compatibility mode with 15 interrupts and delivery of interrupts to only the primary processor. Figure 3-2 depicts the APIC architecture. The APIC actually consists of several components: an I/O APIC that receives interrupts from devices, local APICs that receive interrupts from the I/O APIC on a private APIC bus and that interrupt the CPU they are associated with, and an i8259A-compatible interrupt controller that translates APIC input into PIC-equivalent signals. The I/O APIC is responsible for implementing interrupt routing algorithms—which are software-selectable (the hardware abstraction layer, or HAL, makes the selection on Windows)—that both balance the device interrupt load across processors and attempt to take advantage of locality, delivering device interrupts to the same processor that has just fielded a previous interrupt of the same type.

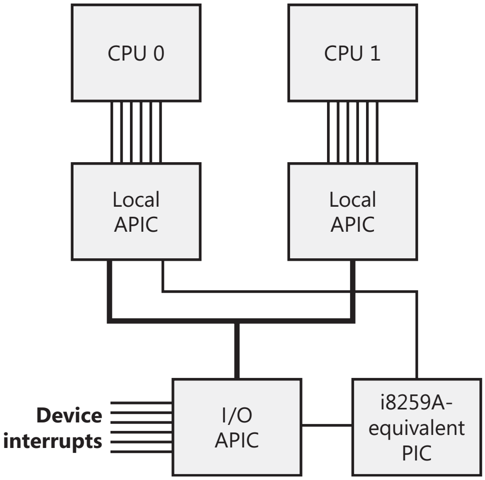


Figure 3-2 x86 APIC architecture

x64 Interrupt Controllers

Because the x64 architecture is compatible with x86 operating systems, x64 systems must provide the same interrupt controllers as does the x86. A significant difference, however, is that the x64 versions of Windows will not run on systems that do not have an APIC and they use the APIC for interrupt control.

IA64 Interrupt Controllers

The IA64 architecture relies on the Streamlined Advanced Programmable Interrupt Controller (SAPIC), which is an evolution of the APIC. A major difference between the APIC and SAPIC architectures is that the I/O APICs on an APIC system deliver interrupts to local APICs over a private APIC bus, whereas on a SAPIC system interrupts traverse the I/O and system bus for faster delivery. Another difference is that interrupt routing and load balancing is handled by the APIC bus on an APIC system, but a SAPIC system, which doesn't have a private APIC bus, requires that the support be programmed into the firmware. Even if load balancing and routing are present in the firmware, Windows does not take advantage of it; instead, it statically assigns interrupts to processors in a round-robin manner.

EXPERIMENT: Viewing the PIC and APIC

You can view the configuration of the PIC on a uniprocessor and the APIC on a multiprocessor by using the *!pic* and *!apic* kernel debugger commands, respectively. (You can't use LiveKd for this experiment because LiveKd can't access hardware.) Here's the output of the *!pic* command on a uniprocessor. (Note that the *!pic* command doesn't work if your system is using an APIC HAL.)

```
lkd> !pic
----- IRQ Number ----- 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
Physically in service:  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
Physically masked:      .  .  .  Y  .  .  Y  Y  .  .  Y  .  .  Y  .  .
Physically requested:   .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
Level Triggered:        .  .  .  .  .  Y  .  .  .  Y  .  Y  .  .  .  .
```

Here's the output of the *!apic* command on a system running with the MPS HAL. The "0:" prefix for the debugger prompt indicates that commands are running on processor 0, so this is the I/O APIC for processor 0:

```
lkd> !apic
Apic @ fffe0000  ID:0 (40010)  LogDesc:01000000  DestFmt:ffffffff  TPR 20
TimeCnt: 0bebc200clk  SpurVec:3f  FaultVec:e3  error:0
Ipi Cmd: 0004001f  Vec:1F  FixedDel    Dest=Self      edg  high
Timer..: 000300fd  Vec:FD  FixedDel    Dest=Self      edg  high       masked
Linti0.: 0001003f  Vec:3F  FixedDel    Dest=Self      edg  high       masked
Linti1.: 000184ff  Vec:FF  NMI         Dest=Self      lvl  high       masked
TMR: 61, 82, 91-92, B1
IRR:
ISR:
```

The following output is for the *!ioapic* command, which displays the configuration of the I/O APIC, the interrupt controller component connected to devices:

```
0: kd> !ioapic
IoApic @ ffd02000  ID:8 (11)  Arb:0
Inti00.: 000100ff  Vec:FF  FixedDel  PhysDest:00  edg  masked
Inti01.: 00000962  Vec:62  LowestDl  Lg:03000000  edg
Inti02.: 000100ff  Vec:FF  FixedDel  PhysDest:00  edg  masked
Inti03.: 00000971  Vec:71  LowestDl  Lg:03000000  edg
Inti04.: 000100ff  Vec:FF  FixedDel  PhysDest:00  edg  masked
Inti05.: 00000961  Vec:61  LowestDl  Lg:03000000  edg
Inti06.: 00010982  Vec:82  LowestDl  Lg:02000000  edg  masked
Inti07.: 000100ff  Vec:FF  FixedDel  PhysDest:00  edg  masked
Inti08.: 000008d1  Vec:D1  FixedDel  Lg:01000000  edg
Inti09.: 000100ff  Vec:FF  FixedDel  PhysDest:00  edg  masked
Inti0A.: 000100ff  Vec:FF  FixedDel  PhysDest:00  edg  masked
Inti0B.: 000100ff  Vec:FF  FixedDel  PhysDest:00  edg  masked
Inti0C.: 00000972  Vec:72  LowestDl  Lg:03000000  edg
Inti0D.: 000100ff  Vec:FF  FixedDel  PhysDest:00  edg  masked
Inti0E.: 00000992  Vec:92  LowestDl  Lg:03000000  edg
Inti0F.: 000100ff  Vec:FF  FixedDel  PhysDest:00  edg  masked
Inti10.: 000100ff  Vec:FF  FixedDel  PhysDest:00  edg  masked
Inti11.: 000100ff  Vec:FF  FixedDel  PhysDest:00  edg  masked
Inti12.: 000100ff  Vec:FF  FixedDel  PhysDest:00  edg  masked
Inti13.: 000100ff  Vec:FF  FixedDel  PhysDest:00  edg  masked
Inti14.: 0000a9a3  Vec:A3  LowestDl  Lg:03000000  lvl
Inti15.: 0000a993  Vec:93  LowestDl  Lg:03000000  lvl
Inti16.: 000100ff  Vec:FF  FixedDel  PhysDest:00  edg  masked
Inti17.: 000100ff  Vec:FF  FixedDel  PhysDest:00  edg  masked
```

Software Interrupt Request Levels (IRQLs)

Although interrupt controllers perform a level of interrupt prioritization, Windows imposes its own interrupt priority scheme known as *interrupt request levels* (IRQLs). The kernel represents IRQLs internally as a number from 0 through 31 on x86 and from 0 to 15 on x64 and IA64, with higher numbers representing higher-priority interrupts. Although the kernel defines the standard set of IRQLs for software interrupts, the HAL maps hardware-interrupt numbers to the IRQLs. Figure 3-3 shows IRQLs defined for the x86 architecture, and Figure 3-4 shows IRQLs for the x64 and IA64 architectures.

Note SYNCH_LEVEL, which multiprocessor versions of the kernel use to protect access to per-processor processor control blocks (PRCB), is not shown in the charts because its value varies across different versions of Windows. See Chapter 6 for a description of SYNCH_LEVEL and its possible values.

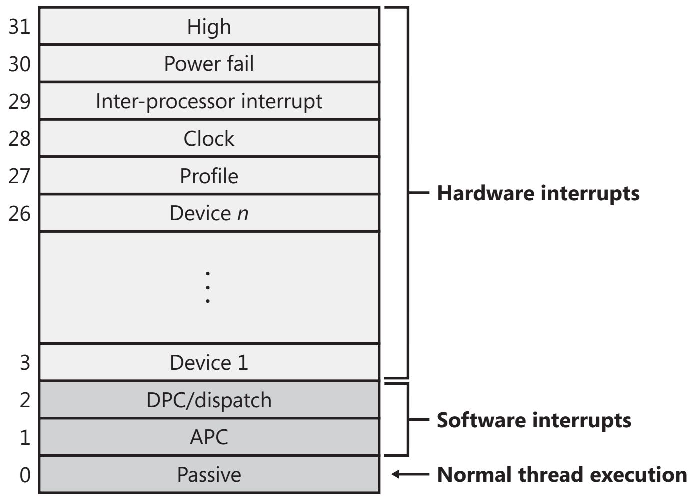
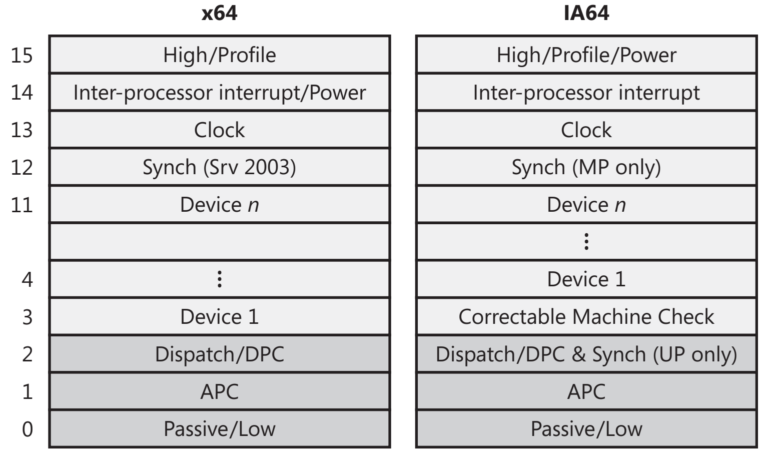


Figure 3-3 x86 interrupt request levels (IRQLs)



	x64	IA64
15	High/Profile	High/Profile/Power
14	Inter-processor interrupt/Power	Inter-processor interrupt
13	Clock	Clock
12	Synch (Srv 2003)	Synch (MP only)
11	Device *n*	Device *n*
		⋮
4	⋮	Device 1
3	Device 1	Correctable Machine Check
2	Dispatch/DPC	Dispatch/DPC & Synch (UP only)
1	APC	APC
0	Passive/Low	Passive/Low

Figure 3-4 x64 and IA64 interrupt request levels (IRQLs)

Interrupts are serviced in priority order, and a higher-priority interrupt preempts the servicing of a lower-priority interrupt. When a high-priority interrupt occurs, the processor saves the interrupted thread's state and invokes the trap dispatchers associated with the interrupt. The trap dispatcher raises the IRQL and calls the interrupt's service routine. After the service routine executes, the interrupt dispatcher lowers the processor's IRQL to where it was before the interrupt occurred and then loads the saved machine state. The interrupted thread resumes executing where it left off. When the kernel lowers the IRQL, lower-priority interrupts that were masked might materialize. If this happens, the kernel repeats the process to handle the new interrupts.

IRQL priority levels have a completely different meaning than thread-scheduling priorities (which are described in Chapter 6). A scheduling priority is an attribute of a thread, whereas an IRQL is an attribute of an interrupt source, such as a keyboard or a mouse. In addition, each processor has an IRQL setting that changes as operating system code executes.

Each processor's IRQL setting determines which interrupts that processor can receive. IRQLs are also used to synchronize access to kernel-mode data structures. (You'll find out more about synchronization later in this chapter.) As a kernel-mode thread runs, it raises or lowers the processor's IRQL either directly by calling *KeRaiseIrql* and *KeLowerIrql* or, more commonly, indirectly via calls to functions that acquire kernel synchronization objects. As Figure 3-5 illustrates, interrupts from a source with an IRQL above the current level interrupt the processor, whereas interrupts from sources with IRQLs equal to or below the current level are *masked* until an executing thread lowers the IRQL.

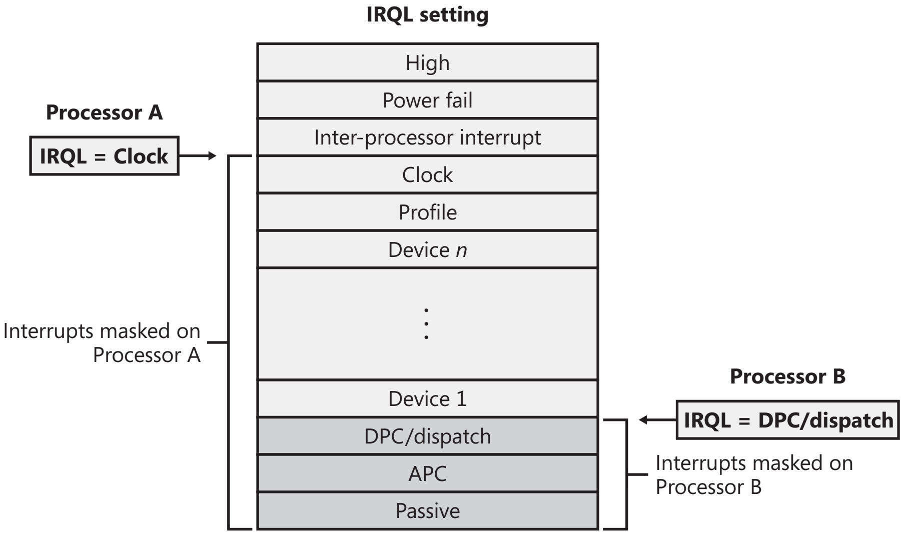


Figure 3-5 Masking interrupts

Because accessing a PIC is a relatively slow operation, HALs that use a PIC implement a performance optimization, called *lazy IRQL*, that avoids PIC accesses. When the IRQL is raised, the HAL notes the new IRQL internally instead of changing the interrupt mask. If a lower-priority interrupt subsequently occurs, the HAL sets the interrupt mask to the settings appropriate for the first interrupt and postpones the lower-priority interrupt until the IRQL is lowered. Thus, if no lower-priority interrupts occur while the IRQL is raised, the HAL doesn't need to modify the PIC.

A kernel-mode thread raises and lowers the IRQL of the processor on which it's running, depending on what it's trying to do. For example, when an interrupt occurs, the trap handler (or perhaps the processor) raises the processor's IRQL to the assigned IRQL of the interrupt source. This elevation masks all interrupts at and below that IRQL (on that processor only), which ensures that the processor servicing the interrupt isn't waylaid by an interrupt at the same or a lower level. The masked interrupts are either handled by another processor or held back until the IRQL drops. Therefore, all components of the system, including the kernel and device drivers, attempt to keep the IRQL at passive level (sometimes called low level). They do this because device drivers can respond to hardware interrupts in a timelier manner if the IRQL isn't kept unnecessarily elevated for long periods.

Note An exception to the rule that raising the IRQL blocks interrupts of that level and lower relates to APC_LEVEL interrupts. If a thread raises the IRQL to APC_LEVEL and then is rescheduled because of a DISPATCH_LEVEL interrupt, the system might deliver an APC_LEVEL interrupt to the newly scheduled thread. Thus, APC_LEVEL can be considered a thread-local rather than processor-wide IRQL.

EXPERIMENT: Viewing the IRQL

If you are running the kernel debugger on Windows Server 2003, you can view a processor's IRQL with the *!irql* debugger command:

```
kd> !irql
Debugger saved IRQL for processor 0x0 -- 0 (LOW_LEVEL)
```

Note that there is a field called IRQL in a data structure called the *processor control region* (PCR) and its extension the *processor control block* (PRCB), which contain information about the state of each processor in the system, such as the current IRQL, a pointer to the hardware IDT, the currently running thread, and the next thread selected to run. The kernel and the HAL use this information to perform architecture-specific and machine-specific actions. Portions of the PCR and PRCB structures are defined publicly in the Windows Device Driver Kit (DDK) header file Ntddk.h, so examine that file if you want a complete definition of these structures.

You can view the contents of the PCR with the kernel debugger by using the *!pcr* command:

```
kd> !pcr
PCR Processor 0 @ffdff000
        NtTib.ExceptionList: f8effc68
            NtTib.StackBase: f8effdf0
           NtTib.StackLimit: f8efd000
         NtTib.SubSystemTib: 00000000
              NtTib.Version: 00000000
          NtTib.UserPointer: 00000000
              NtTib.SelfTib: 7ffde000
```

```
       SelfPcr: ffdff000
          Prcb: ffdff120
          Irql: 00000000
           IRR: 00000000
           IDR: ffff28e8
 InterruptMode: 00000000
           IDT: 80036400
           GDT: 80036000
           TSS: 802b5000

 CurrentThread: 81638020
    NextThread: 00000000
    IdleThread: 8046bdf0
```

Unfortunately, Windows does not maintain the Irql field on systems that do not use lazy IRQL, so on most systems the field will always be 0.

Because changing a processor's IRQL has such a significant effect on system operation, the change can be made only in kernel mode—user-mode threads can't change the processor's IRQL. This means that a processor's IRQL is always at passive level when it's executing user-mode code. Only when the processor is executing kernel-mode code can the IRQL be higher.

Each interrupt level has a specific purpose. For example, the kernel issues an *interprocessor interrupt* (IPI) to request that another processor perform an action, such as dispatching a particular thread for execution or updating its translation look-aside buffer cache. The system clock generates an interrupt at regular intervals, and the kernel responds by updating the clock and measuring thread execution time. If a hardware platform supports two clocks, the kernel adds another clock interrupt level to measure performance. The HAL provides a number of interrupt levels for use by interrupt-driven devices; the exact number varies with the processor and system configuration. The kernel uses software interrupts (described later in this chapter) to initiate thread scheduling and to asynchronously break into a thread's execution.

Mapping Interrupts to IRQLs IRQL levels aren't the same as the interrupt requests (IRQs) defined by interrupt controllers—the architectures on which Windows runs don't implement the concept of IRQLs in hardware. So how does Windows determine what IRQL to assign to an interrupt? The answer lies in the HAL. In Windows, a type of device driver called a bus driver determines the presence of devices on its bus (PCI, USB, and so on) and what interrupts can be assigned to a device. The bus driver reports this information to the Plug and Play manager, which decides, after taking into account the acceptable interrupt assignments for all other devices, which interrupt will be assigned to each device. Then it calls the HAL function *HalpGetSystemInterruptVector*, which maps interrupts to IRQLs.

The algorithm for assignment differs for the various HALs that Windows includes. On uniprocessor x86 systems, the HAL performs a straightforward translation: the IRQL of a given interrupt vector is calculated by subtracting the interrupt vector from 27. Thus, if a device uses interrupt vector 5, its ISR executes at IRQL 22. On an x86 multiprocessor system, the mapping isn't as simple. APICs support over 200 interrupt vectors, so there aren't enough IRQLs for a one-to-one correspondence. The multiprocessor HAL therefore assigns IRQLs to interrupt vectors in a round-robin manner, cycling through the device IRQL (DIRQL) range. As a result, on an x86 multiprocessor system there's no easy way for you to predict or to know what IRQL Windows assigns to APIC IRQs. Finally, on x64 and IA64 systems, the HAL computes the IRQL for a given IRQ by dividing the interrupt vector assigned to the IRQ by 16.

Predefined IRQLs Let's take a closer look at the use of the predefined IRQLs, starting from the highest level shown in Figure 3-5:

- The kernel uses *high* level only when it's halting the system in *KeBugCheckEx* and masking out all interrupts.
- *Power fail level* originated in the original Microsoft Windows NT design documents, which specified the behavior of system power failure code, but this IRQL has never been used.
- *Inter-processor interrupt* level is used to request another processor to perform an action, such as queue a DISPATCH_LEVEL interrupt to schedule a particular thread for execution, updating the processor's translation look-aside buffer (TLB) cache, system shutdown, or system crash.
- *Clock* level is used for the system's clock, which the kernel uses to track the time of day as well as to measure and allot CPU time to threads.
- The system's real-time clock uses *profile* level when kernel profiling, a performance measurement mechanism, is enabled. When kernel profiling is active, the kernel's profiling trap handler records the address of the code that was executing when the interrupt occurred. A table of address samples is constructed over time that tools can extract and analyze. You can download Kernrate, a kernel profiling tool that you can use to configure and view profiling-generated statistics, from *http://www.microsoft.com/whdc/system/sysperf/krview.mspx*. See the Kernrate experiment for more information on using this tool.
- The *device* IRQLs are used to prioritize device interrupts. (See the previous section for how hardware interrupt levels are mapped to IRQLs.)
- *DPC/dispatch*-level and *APC*-level interrupts are software interrupts that the kernel and device drivers generate. (DPCs and APCs are explained in more detail later in this chapter.)
- The lowest IRQL, *passive* level, isn't really an interrupt level at all; it's the setting at which normal thread execution takes place and all interrupts are allowed to occur.

EXPERIMENT: Using Kernel Profiler to Profile Execution

You can use the Kernel Profiler tool to enable the system profiling timer, collect samples of the code that is executing when the timer fires, and display a summary showing the frequency distribution across image files and functions. It can be used to track CPU usage consumed by individual processes and/or time spent in kernel mode independent of processes (for example, interrupt service routines). Kernel profiling is useful when you want to obtain a breakdown of where the system is spending time.

In its simplest form, Kernrate samples where time has been spent in each kernel module (for example, Ntoskrnl, drivers, and so on). For example, after installing the Krview package referred to previously, try performing the following steps:

1. Open a command prompt.
2. Type **cd c:\program files\krview\kernrates**.
3. Type **dir**. (You will see kernrate images for each platform.)
4. Run the image that matches your platform (with no arguments or switches). For example, Kernrate_i386_XP.exe is the image for Windows XP running on an x86 system.
5. While Kernrate is running, go perform some other activity on the system. For example, run Windows Media Player and play some music, run a graphics-intensive game, or perform network activity such as doing a directory of a remote network share.
6. Press Ctrl+C to stop Kernrate. This causes Kernrate to display the statistics from the sampling period.

In the sample partial output from Kernrate, Windows Media Player was running, playing a track from a CD.

```
C:\Program Files\KrView\Kernrates>Kernrate_i386_XP.exe
 /===============================\
<         KERNRATE LOG            >
 \===============================/
Date: 2004/05/13   Time:  9:48:28
Machine Name: BIGDAVID
Number of Processors: 1
PROCESSOR_ARCHITECTURE: x86
PROCESSOR_LEVEL: 6

Kernrate User-Specified Command Line:
Kernrate_i386_XP.exe

***> Press ctrl-c to finish collecting profile data
===> Finished Collecting Data, Starting to Process Results

------------Overall Summary:--------------
P0     K 0:00:03.234 (11.7%)  U 0:00:08.352 (30.2%)  I 0:00:16.093 (58.1%)
 DPC 0:00:01.772 ( 6.4%)  Interrupt 0:00:00.350 ( 1.3%)
```

```
       Interrupts= 52899, Interrupt Rate= 1911/
sec.Time   7315 hits, 19531 events per hit --------
 Module                                Hits   msec  %Total  Events/Sec
gv3                                    4735      27679    64 %     3341135
smwdm                                   872      27679    11 %      615305
win32k                                  764      27679    10 %      539097
ntoskrnl                                739      27679    10 %      521457
hal                                     124      27679     1 %       87497
```

The overall summary shows that the system spent 11.7 percent of the time in kernel mode, 30.2 percent in user mode, 58.1 percent idle, 6.4 percent at DPC level, and 1.3 percent at interrupt level. The module with the highest hit rate was GV3.SYS, the processor driver for the Pentium M Geyserville family. It is used for performance collection, which is why it is first. The module with the second highest hit rate was Smwdm.sys, the audio driver for the sound card on the machine used for the test. This makes sense because the major activity going on in the system was Windows Media Player sending sound I/O to the sound driver.

If you have symbols available, you can zoom in on individual modules and see the time spent by function name. For example, profiling the system while dragging a window around the screen rapidly resulted in the following (partial) output:

```
C:\Program Files\KrView\Kernrates>Kernrate_i386_XP.exe -z ntoskrnl -z win32k
 /==============================\
<         KERNRATE LOG           >
 \==============================/
Date: 2004/05/13   Time: 10:26:55

Time   4087 hits, 19531 events per hit --------
 Module                                Hits   msec  %Total  Events/Sec
win32k                                 1649      10424    40 %     3089660
ati2dvag                               1269      10424    31 %     2377670
ntoskrnl                                794      10424    19 %     1487683
gv3                                     162      10424     3 %      303532

----- Zoomed module win32k.sys (Bucket size = 16 bytes, Rounding Down) -------
 Module                                Hits   msec  %Total  Events/Sec
EngPaint                                328      10424    19 %      614559
EngLpkInstalled                         302      10424    18 %      565844

----- Zoomed module ntoskrnl.exe (Bucket size = 16 bytes, Rounding Down) -----
Module                                Hits   msec  %Total  Events/Sec
KiDispatchInterrupt                     243      10424    26 %      455298
ZwYieldExecution                         50      10424     5 %       93682
InterlockedDecrement                     39      10424     4 %       73072
```

The module with the highest hit rate was Win32k.sys, the windowing system driver. Second on the list was the video driver. These results make sense because the main activity in the system was drawing on the screen. Note in the zoomed display for Win32k.sys, the function with the highest hit was EngPaint, the main GDI function to paint on the screen.

One important restriction on code running at DPC/dispatch level or above is that it can't wait for an object if doing so would necessitate the scheduler to select another thread to execute, which is an illegal operation because the scheduler synchronizes its data structures at DPC/dispatch level and cannot therefore be invoked to perform a reschedule. Another restriction is that only nonpaged memory can be accessed at IRQL DPC/dispatch level or higher. This rule is actually a side-effect of the first restriction because attempting to access memory that isn't resident results in a page fault. When a page fault occurs, the memory manager initiates a disk I/O and then needs to wait for the file system driver to read the page in from disk. This wait would in turn require the scheduler to perform a context switch (perhaps to the idle thread if no user thread is waiting to run), thus violating the rule that the scheduler can't be invoked (because the IRQL is still DPC/dispatch level or higher at the time of the disk read). If either of these two restrictions is violated, the system crashes with an IRQL_NOT_LESS_OR_EQUAL crash code. (See Chapter 4 for a thorough discussion of system crashes.) Violating these restrictions is a common bug in device drivers. The Windows Driver Verifier, explained in the section "Driver Verifier" in Chapter 7, has an option you can set to assist in finding this particular type of bug.

Interrupt Objects The kernel provides a portable mechanism—a kernel control object called an interrupt object—that allows device drivers to register ISRs for their devices. An interrupt object contains all the information the kernel needs to associate a device ISR with a particular level of interrupt, including the address of the ISR, the IRQL at which the device interrupts, and the entry in the kernel's IDT with which the ISR should be associated. When an interrupt object is initialized, a few instructions of assembly language code, called the dispatch code, are copied from an interrupt handling template, *KiInterruptTemplate*, and stored in the object. When an interrupt occurs, this code is executed.

This interrupt-object resident code calls the real interrupt dispatcher, which is typically either the kernel's *KiInterruptDispatch* or *KiChainedDispatch* routine, passing it a pointer to the interrupt object. *KiInterruptDispatch* is the routine used for interrupt vectors for which only one interrupt object is registered, and *KiChainedDispatch* is for vectors shared among multiple interrupt objects. The interrupt object contains information this second dispatcher routine needs to locate and properly call the ISR the device driver provides. The interrupt object also stores the IRQL associated with the interrupt so that *KiInterruptDispatch* or *KiChainedDispatch* can raise the IRQL to the correct level before calling the ISR and then lower the IRQL after the ISR has returned. This two-step process is required because there's no way to pass a pointer to the interrupt object (or any other argument for that matter) on the initial dispatch because the initial dispatch is done by hardware. On a multiprocessor system, the kernel allocates and initializes an interrupt object for each CPU, enabling the local APIC on that CPU to accept the particular interrupt. Figure 3-6 shows typical interrupt control flow for interrupts associated with interrupt objects.

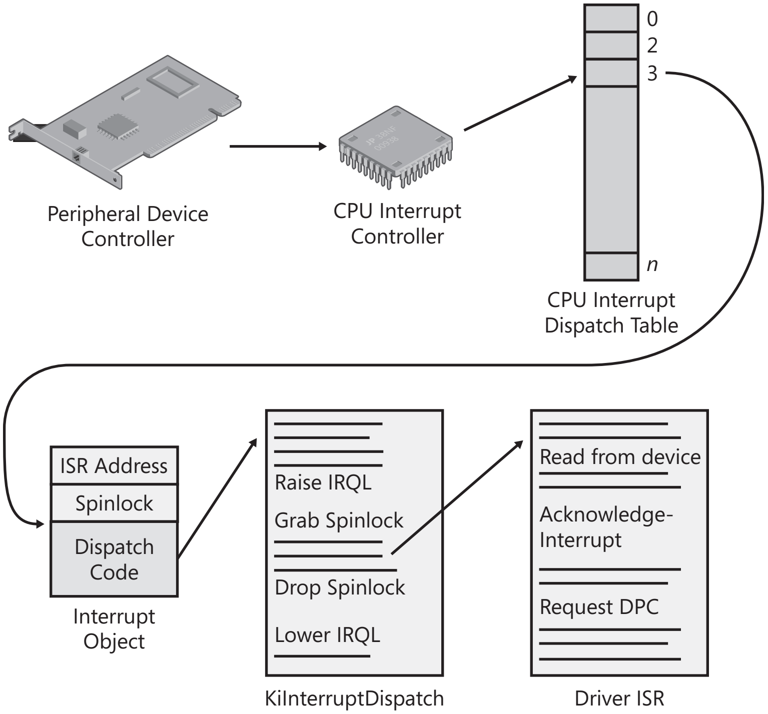


Figure 3-6 Typical interrupt control flow

EXPERIMENT: Examining Interrupt Internals

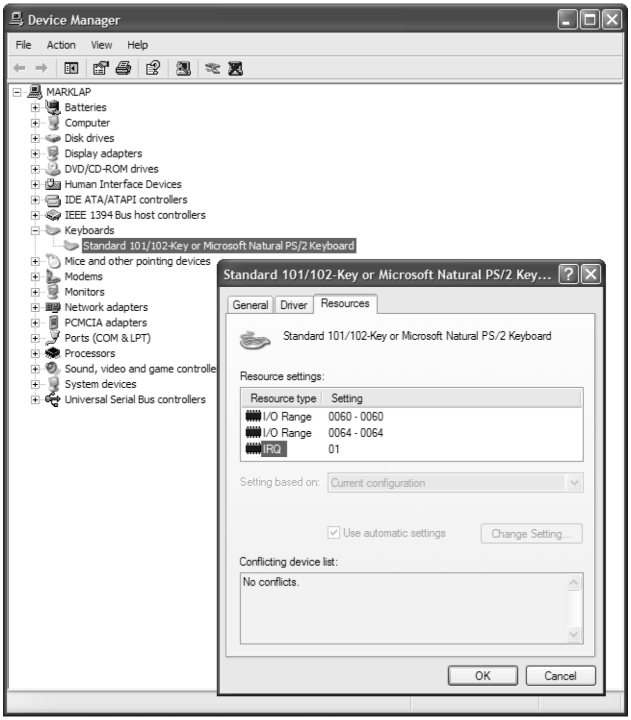
Using the kernel debugger, you can view details of an interrupt object, including its IRQL, ISR address, and custom interrupt dispatching code. First, execute the *!idt* command and locate the entry that includes a reference to *I8042KeyboardInterruptService*, the ISR routine for the PS2 keyboard device:

```
31:     8a39dc3c i8042prt!I8042KeyboardInterruptService (KINTERRUPT 8a39dc00)
```

To view the contents of the interrupt object associated with the interrupt, execute **dt nt!_kinterrupt** with the address following KINTERRUPT:

```
kd> dt nt!_kinterrupt 8a39dc00
nt!_KINTERRUPT
   +0x000 Type             : 22
   +0x002 Size             : 484
   +0x004 InterruptListEntry : _LIST_ENTRY [ 0x8a39dc04 - 0x8a39dc04 ]
   +0x00c ServiceRoutine   : 0xba7e74a2     i8042prt!I8042KeyboardInterruptService+0
   +0x010 ServiceContext   : 0x8a067898
   +0x014 SpinLock         : 0
   +0x018 TickCount        : 0xffffffff
   +0x01c ActualLock       : 0x8a067958  -> 0
   +0x020 DispatchAddress  : 0x80531140     nt!KiInterruptDispatch+0
   +0x024 Vector           : 0x31
   +0x028 Irql             : 0x1a ''
   +0x029 SynchronizeIrql  : 0x1a ''
   +0x02a FloatingSave     : 0 ''
   +0x02b Connected        : 0x1 ''
   +0x02c Number           : 0 ''
   +0x02d ShareVector      : 0 ''
   +0x030 Mode             : 1 ( Latched )
   +0x034 ServiceCount     : 0
   +0x038 DispatchCount    : 0xffffffff
   +0x03c DispatchCode     : [106] 0x56535554
```

In this example, the IRQL Windows assigned to the interrupt is 0x1a (which is 26 in decimal). Because this output is from a uniprocessor x86 system, we calculate that the IRQ is 1, because IRQLs on x86 uniprocessors are calculated by subtracting the IRQ from 27. We can verify this by opening the Device Manager (on the Hardware tab in the System applet in the Control Panel), locating the PS/2 keyboard device, and viewing its resource assignments, as shown in the following figure.


Device Manager
File
Action
View
Help
MARKLAP
Batteries
Computer
Disk drives
Display adapters
DVD/CD-ROM drives
Human Interface Devices
IDE ATA/ATAPI controllers
IEEE 1394 Bus host controllers
Keyboards
Standard 101/102-Key or Microsoft Natural PS/2 Keyboard
Mice and other pointing devices
Modems
Monitors
Network adapters
PCMCIA adapters
Ports (COM & LPT)
Processors
Sound, video and game controlle
System devices
Universal Serial Bus controllers
Standard 101/102-Key or Microsoft Natural PS/2 Key...
General
Driver
Resources
Standard 101/102-Key or Microsoft Natural PS/2 Keyboard
Change Setting...
OK
Cancel

Resource settings:

Resource type	Setting
I/O Range	0060 - 0060
I/O Range	0064 - 0064
IRQ	01

Setting based on: Current configuration

☑ Use automatic settings

Conflicting device list:

No conflicts.

On a multiprocessor x86, the IRQ will be essentially randomly assigned, and on an x64 or IA64 system you will see that the IRQ is the interrupt vector number (0x31—49 decimal—in this example) divided by 16.

The ISR's address for the interrupt object is stored in the ServiceRoutine field (which is what *!idt* displays in its output), and the interrupt code that actually executes when an interrupt occurs is stored in the *DispatchCode* array at the end of the interrupt object. The interrupt code stored there is programmed to build the trap frame on the stack and then call the function stored in the DispatchAddress field (*KiInterruptDispatch* in the example), passing it a pointer to the interrupt object.

Windows and Real-Time Processing

Deadline requirements, either hard or soft, characterize real-time environments. Hard real-time systems (for example, a nuclear power plant control system) have deadlines that the system must meet to avoid catastrophic failures such as loss of equipment or life. Soft real-time systems (for example, a car's fuel-economy optimization system) have deadlines that the system can miss, but timeliness is still a desirable trait. In real-time systems, computers have sensor input devices and control output devices. The designer of a real-time computer system must know worst-case delays between the time an input device generates an interrupt and the time the device's driver can control the output device to respond. This worst-case analysis must take into account the delays the operating system introduces as well as the delays the application and device drivers impose.

Because Windows doesn't prioritize device IRQs in any controllable way and user-level applications execute only when a processor's IRQL is at passive level, Windows isn't always suitable as a real-time operating system. The system's devices and device drivers—not Windows—ultimately determine the worst-case delay. This factor becomes a problem when the real-time system's designer uses off-the-shelf hardware. The designer can have difficulty determining how long every off-the-shelf device's ISR or DPC might take in the worst case. Even after testing, the designer can't guarantee that a special case in a live system won't cause the system to miss an important deadline. Furthermore, the sum of all the delays a system's DPCs and ISRs can introduce usually far exceeds the tolerance of a time-sensitive system.

Although many types of embedded systems (for example, printers and automotive computers) have real-time requirements, Windows XP Embedded doesn't have real-time characteristics. It is simply a version of Windows XP that makes it possible, using system designer technology that Microsoft licensed from VenturCom, to produce small-footprint versions of Windows XP suitable for running on devices with limited resources. For example, a device that has no networking capability would omit all the Windows XP components related to networking, including network management tools and adapter and protocol stack device drivers.

Still, there are third-party vendors that supply real-time kernels for Windows. The approach these vendors take is to embed their real-time kernel in a custom HAL and to have Windows run as a task in the real-time operating system. The task running Windows serves as the user interface to the system and has a lower priority than the tasks responsible for managing the device. See VenturCom's Web site, *www.venturcom.com*, for an example of a third-party real-time kernel extension for Windows.

Associating an ISR with a particular level of interrupt is called *connecting an interrupt object*, and dissociating an ISR from an IDT entry is called *disconnecting an interrupt object*. These operations, accomplished by calling the kernel functions *IoConnectInterrupt* and *IoDisconnectInterrupt*, allow a device driver to "turn on" an ISR when the driver is loaded into the system and to "turn off" the ISR if the driver is unloaded.

Using the interrupt object to register an ISR prevents device drivers from fiddling directly with interrupt hardware (which differs among processor architectures) and from needing to know any details about the IDT. This kernel feature aids in creating portable device drivers because it eliminates the need to code in assembly language or to reflect processor differences in device drivers.

Interrupt objects provide other benefits as well. By using the interrupt object, the kernel can synchronize the execution of the ISR with other parts of a device driver that might share data with the ISR. (See Chapter 9 for more information about how device drivers respond to interrupts.)

Furthermore, interrupt objects allow the kernel to easily call more than one ISR for any interrupt level. If multiple device drivers create interrupt objects and connect them to the same IDT entry, the interrupt dispatcher calls each routine when an interrupt occurs at the specified interrupt line. This capability allows the kernel to easily support "daisy-chain" configurations, in which several devices share the same interrupt line. The chain breaks when one of the ISRs claims ownership for the interrupt by returning a status to the interrupt dispatcher. If multiple devices sharing the same interrupt require service at the same time, devices not acknowledged by their ISRs will interrupt the system again once the interrupt dispatcher has lowered the IRQL. Chaining is permitted only if all the device drivers wanting to use the same interrupt indicate to the kernel that they can share the interrupt; if they can't, the Plug and Play manager reorganizes their interrupt assignments to ensure that it honors the sharing requirements of each. If the interrupt vector is shared, the interrupt object invokes *KiChainedDispatch*, which will invoke the ISRs of each registered interrupt object in turn until one of them claims the interrupt or all have been executed. In the earlier sample *!idt* output, vector 0x3b is connected to several chained interrupt objects.

Software Interrupts

Although hardware generates most interrupts, the Windows kernel also generates software interrupts for a variety of tasks, including these:

- Initiating thread dispatching
- Non-time-critical interrupt processing
- Handling timer expiration
- Asynchronously executing a procedure in the context of a particular thread
- Supporting asynchronous I/O operations

These tasks are described in the following subsections.

Dispatch or Deferred Procedure Call (DPC) Interrupts When a thread can no longer continue executing, perhaps because it has terminated or because it voluntarily enters a wait state, the kernel calls the dispatcher directly to effect an immediate context switch. Sometimes, however, the kernel detects that rescheduling should occur when it is deep within many layers of code. In this situation, the kernel requests dispatching but defers its occurrence until it completes its current activity. Using a DPC software interrupt is a convenient way to achieve this delay.

The kernel always raises the processor's IRQL to DPC/dispatch level or above when it needs to synchronize access to shared kernel structures. This disables additional software interrupts and thread dispatching. When the kernel detects that dispatching should occur, it requests a DPC/dispatch-level interrupt; but because the IRQL is at or above that level, the processor holds the interrupt in check. When the kernel completes its current activity, it sees that it's going to lower the IRQL below DPC/dispatch level and checks to see whether any dispatch interrupts are pending. If there are, the IRQL drops to DPC/dispatch level and the dispatch interrupts are processed. Activating the thread dispatcher by using a software interrupt is a way to defer dispatching until conditions are right. However, Windows uses software interrupts to defer other types of processing as well.

In addition to thread dispatching, the kernel also processes deferred procedure calls (DPCs) at this IRQL. A DPC is a function that performs a system task—a task that is less time-critical than the current one. The functions are called *deferred* because they might not execute immediately.

DPCs provide the operating system with the capability to generate an interrupt and execute a system function in kernel mode. The kernel uses DPCs to process timer expiration (and release threads waiting for the timers) and to reschedule the processor after a thread's quantum expires. Device drivers use DPCs to complete I/O requests. To provide timely service for hardware interrupts, Windows—with the cooperation of device drivers—attempts to keep the IRQL below device IRQL levels. One way that this goal is achieved is for device driver ISRs to perform the minimal work necessary to acknowledge their device, save volatile interrupt state, and defer data transfer or other less time-critical interrupt processing activity for execution in a DPC at DPC/dispatch IRQL. (See Chapter 9 for more information on DPCs and the I/O system.)

A DPC is represented by a *DPC object*, a kernel control object that is not visible to user-mode programs but is visible to device drivers and other system code. The most important piece of information the DPC object contains is the address of the system function that the kernel will call when it processes the DPC interrupt. DPC routines that are waiting to execute are stored in kernel-managed queues, one per processor, called *DPC queues*. To request a DPC, system code calls the kernel to initialize a DPC object and then places it in a DPC queue.

By default, the kernel places DPC objects at the end of the DPC queue of the processor on which the DPC was requested (typically the processor on which the ISR executed). A device driver can override this behavior, however, by specifying a DPC priority (low, medium, or high, where medium is the default) and by targeting the DPC at a particular processor. A DPC aimed at a specific CPU is known as a *targeted DPC*. If the DPC has a low or medium priority, the kernel places the DPC object at the end of the queue; if the DPC has a high priority, the kernel inserts the DPC object at the front of the queue.

When the processor's IRQL is about to drop from an IRQL of DPC/dispatch level or higher to a lower IRQL (APC or passive level), the kernel processes DPCs. Windows ensures that the IRQL remains at DPC/dispatch level and pulls DPC objects off the current processor's queue until the queue is empty (that is, the kernel "drains" the queue), calling each DPC function in turn. Only when the queue is empty will the kernel let the IRQL drop below DPC/dispatch level and let regular thread execution continue. DPC processing is depicted in Figure 3-7.

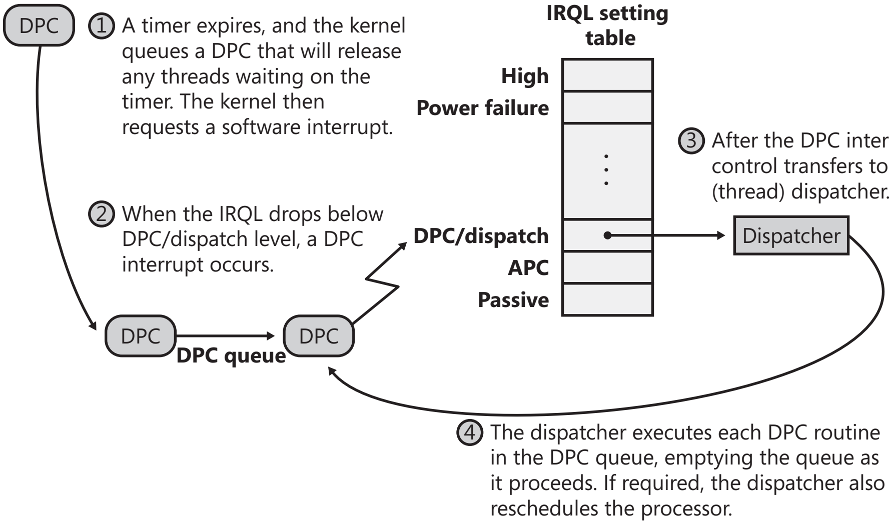

DPC
1 A timer expires, and the kernel queues a DPC that will release any threads waiting on the timer. The kernel then requests a software interrupt.
IRQL setting table
High
Power failure
3 After the DPC interrupt, control transfers to the (thread) dispatcher.
2 When the IRQL drops below DPC/dispatch level, a DPC interrupt occurs.
DPC/dispatch
Dispatcher
APC
Passive
DPC
DPC
DPC queue
4 The dispatcher executes each DPC routine in the DPC queue, emptying the queue as it proceeds. If required, the dispatcher also reschedules the processor.

Figure 3-7 Delivering a DPC

DPC priorities can affect system behavior another way. The kernel usually initiates DPC queue draining with a DPC/dispatch-level interrupt. The kernel generates such an interrupt only if the DPC is directed at the processor the ISR is requested on and the DPC has a high or medium priority. If the DPC has a low priority, the kernel requests the interrupt only if the number of outstanding DPC requests for the processor rises above a threshold or if the number of DPCs requested on the processor within a time window is low. If a DPC is targeted at a CPU different from the one on which the ISR is running and the DPC's priority is high, the kernel immediately signals the target CPU (by sending it a dispatch IPI) to drain its DPC queue. If the priority is medium or low, the number of DPCs queued on the target processor must exceed a threshold for the kernel to trigger a DPC/dispatch interrupt. The system idle thread also drains the DPC queue for the processor it runs on. Although DPC targeting and priority levels are flexible, device drivers rarely need to change the default behavior of their DPC objects. Table 3-1 summarizes the situations that initiate DPC queue draining.

Table 3-1 DPC Interrupt Generation Rules

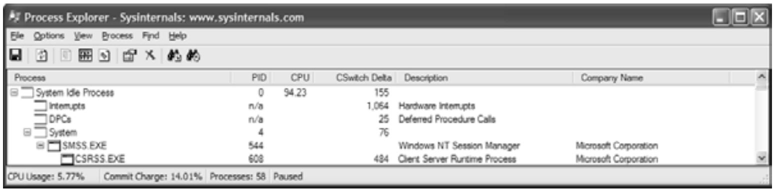
DPC Priority	DPC Targeted at ISR's Processor	DPC Targeted at Another Processor
Low	DPC queue length exceeds maximum DPC queue length or DPC request rate is less than minimum DPC request rate	DPC queue length exceeds maximum DPC queue length or System is idle
Medium	Always	DPC queue length exceeds maximum DPC queue length or System is idle
High	Always	Always

Because user-mode threads execute at low IRQL, the chances are good that a DPC will interrupt the execution of an ordinary user's thread. DPC routines execute without regard to what thread is running, meaning that when a DPC routine runs, it can't assume what process address space is currently mapped. DPC routines can call kernel functions, but they can't call system services, generate page faults, or create or wait for dispatcher objects (explained later in this chapter). They can, however, access nonpaged system memory addresses, because system address space is always mapped regardless of what the current process is.

DPCs are provided primarily for device drivers, but the kernel uses them too. The kernel most frequently uses a DPC to handle quantum expiration. At every tick of the system clock, an interrupt occurs at clock IRQL. The clock interrupt handler (running at clock IRQL) updates the system time and then decrements a counter that tracks how long the current thread has run. When the counter reaches 0, the thread's time quantum has expired and the kernel might need to reschedule the processor, a lower-priority task that should be done at DPC/dispatch IRQL. The clock interrupt handler queues a DPC to initiate thread dispatching and then finishes its work and lowers the processor's IRQL. Because the DPC interrupt has a lower priority than do device interrupts, any pending device interrupts that surface before the clock interrupt completes are handled before the DPC interrupt occurs.

EXPERIMENT: Monitoring Interrupt and DPC Activity

You can use Process Explorer to monitor interrupt and DPC activity by adding the Context Switch Delta column and watching the Interrupt and DPC processes. These are not real processes, but they are shown as processes for convenience and therefore do not incur context switches. Process Explorer's context switch count for these pseudo processes reflects the number of occurrences of each within the previous refresh interval. You can stimulate interrupt and DPC activity by moving the mouse quickly around the screen.


Process Explorer - Sysinternals: www.sysinternals.com
File Options View Process Find Help

Process	PID	CPU	CSwitch Delta	Description	Company Name
System Idle Process	0	94.23	155		
Interrupts	n/a		1,064	Hardware Interrupts	
DPCs	n/a		25	Deferred Procedure Calls	
System	4		76		
SMSS.EXE	544			Windows NT Session Manager	Microsoft Corporation
CSRSS.EXE	608		484	Client Server Runtime Process	Microsoft Corporation

CPU Usage: 5.77% Commit Charge: 14.01% Processes: 58 Paused

You can also trace the execution of specific interrupt service routines and deferred procedure calls with the built-in event tracing support (described later in this chapter) in Windows XP Service Pack 2 and Windows Server 2003 Service Pack 1 and later.

1. Start capturing events by typing the following command:

   ```
   tracelog -start -f kernel.etl -b 64 -UsePerfCounter -
   eflag 8 0x307 0x4084 0 0 0 0 0 0
   ```

2. Stop capturing events by typing:

   ```
   tracelog -stop to stop logging.
   ```

3. Generate reports for the event capture by typing:

   ```
   tracerpt kernel.etl -df -o -report
   ```

   This will generate two files: workload.txt and dumpfile.csv.

4. Open "workload.txt" and you will see summaries of the time spent in ISRs and DPCs by each driver type.

5. Open the file "dumpfile.csv" created in step 4; search for lines with "DPC" or "ISR" in the second value. For example, the following three lines from a dumpfile.csv generated using the above commands show a timer DPC, a DPC, and an ISR:

   ```
       PerfInfo,   TimerDPC, 0xFFFFFFFF,   127383953645422825,         0,
   0, 127383953645421500, 0xFB03A385, 0, 0
       PerfInfo,        DPC, 0xFFFFFFFF,   127383953645424040,         0,
   0, 127383953645421394, 0x804DC87D, 0, 0
       PerfInfo,        ISR, 0xFFFFFFFF,   127383953645470903,         0,
   0, 127383953645468696, 0xFB48D5E0, 0, 0, 0
   ```

Doing an "ln" command in the kernel debugger on the start address in each event record (the eighth value on each line) shows the name of the function that executed the DPC or ISR:

```
lkd> ln 0xFB03A385
(fb03a385)   rdbss!RxTimerDispatch   |  (fb03a41e)   rdbss!RxpWorkerThreadDispatcher
lkd> ln 0x804DC87D
(804dc87d)   nt!KiTimerExpiration   |  (804dc93b)   nt!KeSetTimerEx
lkd> ln 0xFB48D5E0
(fb48d5e0)   atapi!IdePortInterrupt   |  (fb48d622)   atapi!IdeCheckEmptyChannel
```

The first is a DPC for a timer expiration for a timer queued by the file system redirector client driver. The second is a DPC for a generic timer expiration. The third address is the address of the ISR for the ATAPI port driver. For more information, see *http://www.microsoft.com/whdc/driver/perform/mmdrv.mspx.*

Asynchronous Procedure Call (APC) Interrupts Asynchronous procedure calls (APCs) provide a way for user programs and system code to execute in the context of a particular user thread (and hence a particular process address space). Because APCs are queued to execute in the context of a particular thread and run at an IRQL less than DPC/dispatch level, they don't operate under the same restrictions as a DPC. An APC routine can acquire resources (objects), wait for object handles, incur page faults, and call system services.

APCs are described by a kernel control object, called an *APC object*. APCs waiting to execute reside in a kernel-managed *APC queue*. Unlike the DPC queue, which is systemwide, the APC queue is thread-specific—each thread has its own APC queue. When asked to queue an APC, the kernel inserts it into the queue belonging to the thread that will execute the APC routine. The kernel, in turn, requests a software interrupt at APC level, and when the thread eventually begins running, it executes the APC.

There are two kinds of APCs: kernel mode and user mode. Kernel-mode APCs don't require "permission" from a target thread to run in that thread's context, while user-mode APCs do. Kernel-mode APCs interrupt a thread and execute a procedure without the thread's intervention or consent. There are also two types of kernel-mode APCs: normal and special. A thread can disable both types by raising the IRQL to APC_LEVEL or by calling *KeEnterGuardedRegion*, which was introduced in Windows Server 2003. *KeEnterGuardedRegionThread* disables APC delivery by setting the SpecialApcDisable field in the calling thread's KTHREAD structure (described further in Chapter 6). A thread can disable normal APCs only by calling *KeEnterCriticalRegion*, which sets the KernelApcDisable field in the thread's KTHREAD structure.

The executive uses kernel-mode APCs to perform operating system work that must be completed within the address space (in the context) of a particular thread. It can use special kernel-mode APCs to direct a thread to stop executing an interruptible system service, for example, or to record the results of an asynchronous I/O operation in a thread's address space. Environment subsystems use special kernel-mode APCs to make a thread suspend or terminate itself or to get or set its user-mode execution context. The POSIX subsystem uses kernel-mode APCs to emulate the delivery of POSIX signals to POSIX processes.

Device drivers also use kernel-mode APCs. For example, if an I/O operation is initiated and a thread goes into a wait state, another thread in another process can be scheduled to run. When the device finishes transferring data, the I/O system must somehow get back into the context of the thread that initiated the I/O so that it can copy the results of the I/O operation to the buffer in the address space of the process containing that thread. The I/O system uses a special kernel-mode APC to perform this action. (The use of APCs in the I/O system is discussed in more detail in Chapter 9.)

Several Windows APIs, such as *ReadFileEx*, *WriteFileEx*, and *QueueUserAPC*, use user-mode APCs. For example, the *ReadFileEx* and *WriteFileEx* functions allow the caller to specify a completion routine to be called when the I/O operation finishes. The I/O completion is implemented by queueing an APC to the thread that issued the I/O. However, the callback to the completion routine doesn't necessarily take place when the APC is queued because user-mode APCs are delivered to a thread only when it's in an *alertable wait state*. A thread can enter a wait state either by waiting for an object handle and specifying that its wait is alertable (with the Windows *WaitForMultipleObjectsEx* function) or by testing directly whether it has a pending APC (using *SleepEx*). In both cases, if a user-mode APC is pending, the kernel interrupts (alerts) the thread, transfers control to the APC routine, and resumes the thread's execution when the APC routine completes. Unlike kernel-mode APCs, which execute at APC level, user-mode APCs execute at passive level.

APC delivery can reorder the wait queues—the lists of which threads are waiting for what, and in what order they are waiting. (Wait resolution is described in the section "Low-IRQL Synchronization" later in this chapter.) If the thread is in a wait state when an APC is delivered, after the APC routine completes, the wait is reissued or reexecuted. If the wait still isn't resolved, the thread returns to the wait state, but now it will be at the end of the list of objects it's waiting for. For example, because APCs are used to suspend a thread from execution, if the thread is waiting for any objects, its wait will be removed until the thread is resumed, after which that thread will be at the end of the list of threads waiting to access the objects it was waiting for.

Exception Dispatching

In contrast to interrupts, which can occur at any time, exceptions are conditions that result directly from the execution of the program that is running. Windows introduced a facility known as *structured exception handling*, which allows applications to gain control when exceptions occur. The application can then fix the condition and return to the place the exception occurred, unwind the stack (thus terminating execution of the subroutine that raised the exception), or declare back to the system that the exception isn't recognized and the system should continue searching for an exception handler that might process the exception. This section assumes you're familiar with the basic concepts behind Windows structured exception handling—if you're not, you should read the overview in the Windows API reference documentation on the Platform SDK or chapters 23 through 25 in Jeffrey Richter's book *Programming Applications for Microsoft Windows* (Fourth Edition, Microsoft Press, 2000) before proceeding. Keep in mind that although exception handling is made accessible through language extensions (for example, the *__try* construct in Microsoft Visual C++), it is a system mechanism and hence isn't language-specific. Other examples of consumers of Windows exception handling include C++ and Java exceptions.

On the x86, all exceptions have predefined interrupt numbers that directly correspond to the entry in the IDT that points to the trap handler for a particular exception. Table 3-2 shows x86-defined exceptions and their assigned interrupt numbers. Because the first entries of the IDT are used for exceptions, hardware interrupts are assigned entries later in the table, as mentioned earlier.

Table 3-2 x86 Exceptions and Their Interrupt Numbers

Interrupt Number	Exception
0	Divide Error
1	DEBUG TRAP
2	NMI/NPX Error
3	Breakpoint
4	Overflow
5	BOUND/Print Screen
6	Invalid Opcode
7	NPX Not Available
8	Double Exception
9	NPX Segment Overrun
A	Invalid Task State Segment (TSS)
B	Segment Not Present
C	Stack Fault
D	General Protection
E	Page Fault
F	Intel Reserved
10	Floating Point
11	Alignment Check

All exceptions, except those simple enough to be resolved by the trap handler, are serviced by a kernel module called the *exception dispatcher*. The exception dispatcher's job is to find an exception handler that can "dispose of" the exception. Examples of architecture-independent exceptions that the kernel defines include memory access violations, integer divide-by-zero, integer overflow, floating-point exceptions, and debugger breakpoints. For a complete list of architecture-independent exceptions, consult the Windows API reference documentation.

The kernel traps and handles some of these exceptions transparently to user programs. For example, encountering a breakpoint while executing a program being debugged generates an exception, which the kernel handles by calling the debugger. The kernel handles certain other exceptions by returning an unsuccessful status code to the caller.

A few exceptions are allowed to filter back, untouched, to user mode. For example, a memory access violation or an arithmetic overflow generates an exception that the operating system doesn't handle. An environment subsystem can establish *frame-based exception handlers* to deal with these exceptions. The term *frame-based* refers to an exception handler's association with a particular procedure activation. When a procedure is invoked, a *stack frame* representing that activation of the procedure is pushed onto the stack. A stack frame can have one or more exception handlers associated with it, each of which protects a particular block of code in the source program. When an exception occurs, the kernel searches for an exception handler associated with the current stack frame. If none exists, the kernel searches for an exception handler associated with the previous stack frame, and so on, until it finds a frame-based exception handler. If no exception handler is found, the kernel calls its own default exception handlers.

When an exception occurs, whether it is explicitly raised by software or implicitly raised by hardware, a chain of events begins in the kernel. The CPU hardware transfers control to the kernel trap handler, which creates a trap frame (as it does when an interrupt occurs). The trap frame allows the system to resume where it left off if the exception is resolved. The trap handler also creates an exception record that contains the reason for the exception and other pertinent information.

If the exception occurred in kernel mode, the exception dispatcher simply calls a routine to locate a frame-based exception handler that will handle the exception. Because unhandled kernel-mode exceptions are considered fatal operating system errors, you can assume that the dispatcher always finds an exception handler.

If the exception occurred in user mode, the exception dispatcher does something more elaborate. As you'll see in Chapter 6, the Windows subsystem has a debugger port and an exception port to receive notification of user-mode exceptions in Windows processes. The kernel uses these in its default exception handling, as illustrated in Figure 3-8.

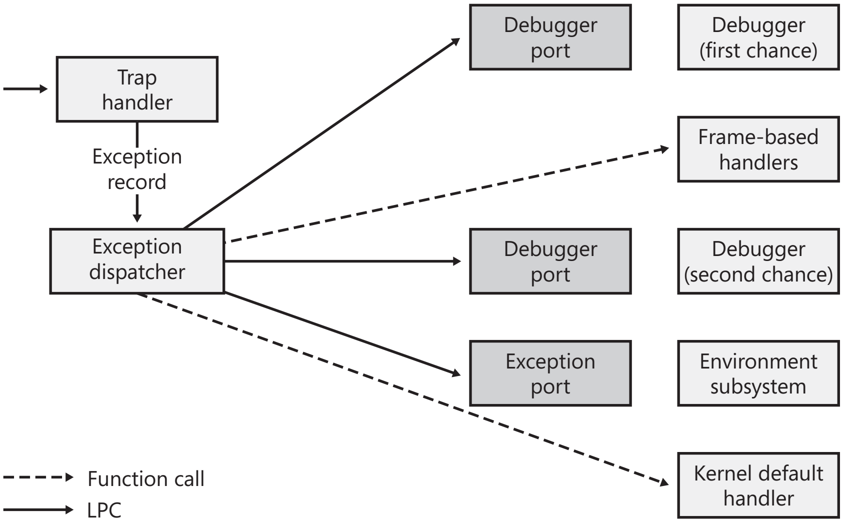


Figure 3-8 Dispatching an exception

Debugger breakpoints are common sources of exceptions. Therefore, the first action the exception dispatcher takes is to see whether the process that incurred the exception has an associated debugger process. If it does and the system is Windows 2000, the exception dispatcher sends the first-chance debug message via an LPC to the debugger port associated with the process that incurred the exception. The LPC message is sent to the session manager process, which then dispatches it to the appropriate debugger process. On Windows XP and Windows Server 2003, the exception dispatcher sends a debugger object message to the *debug object* associated with the process (which internally the system refers to as a port).

If the process has no debugger process attached, or if the debugger doesn't handle the exception, the exception dispatcher switches into user mode, copies the trap frame to the user stack formatted as a CONTEXT data structure (documented in the Platform SDK), and calls a routine to find a frame-based exception handler. If none is found, or if none handles the exception, the exception dispatcher switches back into kernel mode and calls the debugger again to allow the user to do more debugging. (This is called the second-chance notification.)

If the debugger isn't running and no frame-based handlers are found, the kernel sends a message to the exception port associated with the thread's process. This exception port, if one exists, was registered by the environment subsystem that controls this thread. The exception port gives the environment subsystem, which presumably is listening at the port, the opportunity to translate the exception into an environment-specific signal or exception. CSRSS (Client/Server Run-Time Subsystem) simply presents a message box notifying the user of the fault and terminates the process, and when POSIX gets a message from the kernel that one of its threads generated an exception, the POSIX subsystem sends a POSIX-style signal to the thread that caused the exception. However, if the kernel progresses this far in processing the exception and the subsystem doesn't handle the exception, the kernel executes a default exception handler that simply terminates the process whose thread caused the exception.

Unhandled Exceptions

All Windows threads have an exception handler declared at the top of the stack that processes unhandled exceptions. This exception handler is declared in the internal Windows *start-of-process* or *start-of-thread* function. The start-of-process function runs when the first thread in a process begins execution. It calls the main entry point in the image. The start-of-thread function runs when a user creates additional threads. It calls the user-supplied thread start routine specified in the *CreateThread* call.

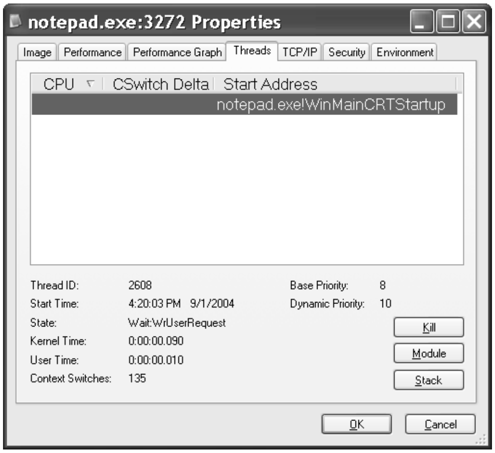
EXPERIMENT: Viewing the Real User Start Address for Windows Threads

The fact that each Windows thread begins execution in a system-supplied function (and not the user-supplied function) explains why the start address for thread 0 is the same for every Windows process in the system (and why the start addresses for secondary threads are also the same). The start address for thread 0 in Windows processes is the Windows start-of-process function; the start address for any other threads would be the Windows start-of-thread function. To see the user-supplied function address, use the Tlist utility in the Windows Support Tools. Type **tlist process-name** or **tlist process-id** to get the detailed process output that includes this information. For example, compare the thread start addresses for the Windows Explorer process as reported by Pstat (in the Platform SDK) and Tlist:

```
C:\> pstat
§
pid:3f8 pri: 8 Hnd:  329 Pf:  80043 Ws:   4620K explorer.exe
 tid pri Ctx Swtch StrtAddr   User Time Kernel Time  State
  7c   9     16442 77E878C1 0:00:01.241 0:00:01.251 Wait:UserRequest
 42c  11    157888 77E92C50 0:00:07.110 0:00:34.309 Wait:UserRequest
 44c   8      6357 77E92C50 0:00:00.070 0:00:00.140 Wait:UserRequest
 1cc   8      3318 77E92C50 0:00:00.030 0:00:00.070 Wait:DelayExecution
 §

C:\> tlist explorer
 1016 explorer.exe      Program Manager
   CWD:     C:\
   CmdLine: Explorer.exe
   VirtualSize:    25348 KB   PeakVirtualSize:    31052 KB
   WorkingSetSize:  1804 KB   PeakWorkingSetSize:  3276 KB
   NumberOfThreads: 4
    149 Win32StartAddr:0x01009dbd LastErr:0x0000007e State:Waiting
     86 Win32StartAddr:0x77c5d4a5 LastErr:0x00000000 State:Waiting
     62 Win32StartAddr:0x00000977 LastErr:0x00000000 State:Waiting
    179 Win32StartAddr:0x0100d8d4 LastErr:0x00000002 State:Waiting
```

The start address of thread 0 reported by Pstat is the internal Windows start-of-process function; the start addresses for threads 1 through 3 are the internal Windows start-of-thread functions. Tlist, on the other hand, shows the user-supplied Windows start address (the user function called by the internal Windows start function).

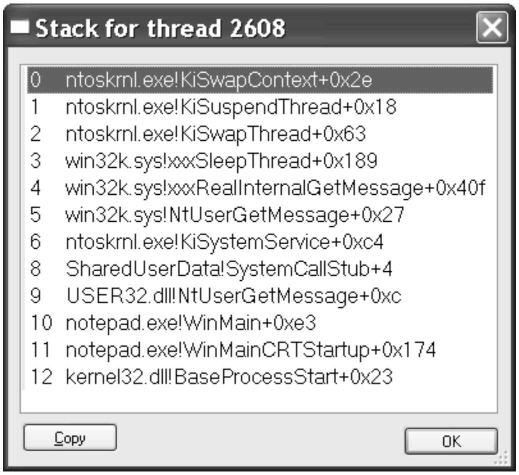
Because most threads in Windows processes start at one of the system-supplied wrapper functions, Process Explorer, when displaying the start address of threads in a process, skips the initial call frame that represents the wrapper function and instead shows the second frame on the stack. For example, notice the thread start address of a process running Notepad.exe:



CPU	CSwitch Delta	Start Address
		notepad.exe!WinMainCRTStartup

Thread ID: 2608
Start Time: 4:20:03 PM 9/1/2004
State: Wait:WrUserRequest
Kernel Time: 0:00:00.090
User Time: 0:00:00.010
Context Switches: 135
Base Priority: 8
Dynamic Priority: 10

Process Explorer does display the complete call hierarchy when it displays the call stack. Notice the following results when the Stack button is clicked:



```
0   ntoskrnl.exe!KiSwapContext+0x2e
1   ntoskrnl.exe!KiSuspendThread+0x18
2   ntoskrnl.exe!KiSwapThread+0x63
3   win32k.sys!xxxSleepThread+0x189
4   win32k.sys!xxxRealInternalGetMessage+0x40f
5   win32k.sys!NtUserGetMessage+0x27
6   ntoskrnl.exe!KiSystemService+0xc4
8   SharedUserData!SystemCallStub+4
9   USER32.dll!NtUserGetMessage+0xc
10  notepad.exe!WinMain+0xe3
11  notepad.exe!WinMainCRTStartup+0x174
12  kernel32.dll!BaseProcessStart+0x23
```

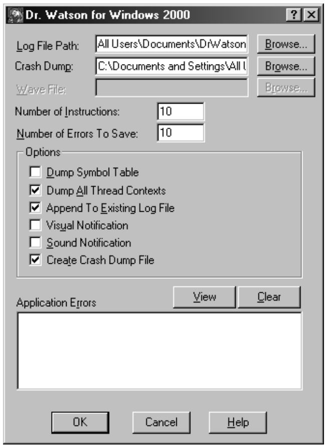
Line 12 in the preceding figure is the first frame on the stack—the start of the process wrapper. The second frame (line 11) is the main entry point into Notepad.exe.

The generic code for these internal start functions is shown here:

```
void Win32StartOfProcess(
    LPTHREAD_START_ROUTINE lpStartAddr,
    LPVOID lpvThreadParm){

    __try {

        DWORD dwThreadExitCode = lpStartAddr(lpvThreadParm);
        ExitThread(dwThreadExitCode);

    } __except(UnhandledExceptionFilter(
            GetExceptionInformation())) {

        ExitProcess(GetExceptionCode());
    }
}
```

Notice that the Windows unhandled exception filter is called if the thread has an exception that it doesn't handle. The purpose of this function is to provide the system-defined behavior for what to do when an exception is not handled, which is based on the contents of the HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\AeDebug registry key. There are two important values: Auto and Debugger. Auto tells the unhandled exception filter whether to automatically run the debugger or ask the user what to do. By default, it is set to 1, which means that it will launch the debugger automatically. However, installing development tools such as Visual Studio changes this to 0. The Debugger value is a string that points to the path of the debugger executable to run in the case of an unhandled exception.

The default debugger is \Windows\System32\Drwtsn32.exe (Dr. Watson), which isn't really a debugger but rather a postmortem tool that captures the state of the application "crash" and records it in a log file (Drwtsn32.log) and a process crash dump file (User.dmp), both found by default in the \Documents And Settings\All Users\Documents\DrWatson folder. To see (or modify) the configuration for Dr. Watson, run it interactively—it displays a window with the current settings, as shown in Figure 3-9.



Log File Path: All Users\Documents\DrWatson
Crash Dump: C:\Documents and Settings\All U
Wave File:
Number of Instructions: 10
Number of Errors To Save: 10

Options

☐ Dump Symbol Table
☑ Dump All Thread Contexts
☑ Append To Existing Log File
☐ Visual Notification
☐ Sound Notification
☑ Create Crash Dump File

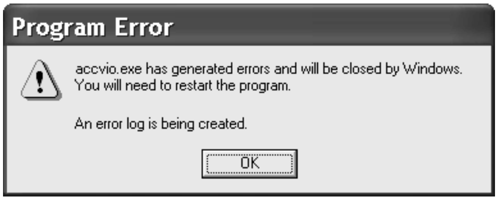
Application Errors

Figure 3-9 Windows 2000 Dr. Watson default settings

The log file contains basic information such as the exception code, the name of the image that failed, a list of loaded DLLs, and a stack and instruction trace for the thread that incurred the exception. For a detailed description of the contents of the log file, run Dr. Watson and click the Help button shown in Figure 3-9.

The crash dump file contains the private pages in the process at the time of the exception. (The file doesn't include code pages from EXEs or DLLs.) This file can be opened by WinDbg, the Windows debugger that comes with the Debugging Tools package, or by Visual Studio 2003 and later. Because the User.dmp file is overwritten each time a process crashes, unless you rename or copy the file after each process crash, you'll have only the latest one on your system.

On Windows 2000 Professional systems, visual notification is turned on by default. The message box shown in Figure 3-10 is displayed by Dr. Watson after it generates the crash dump and records information in its log file.



Program Error

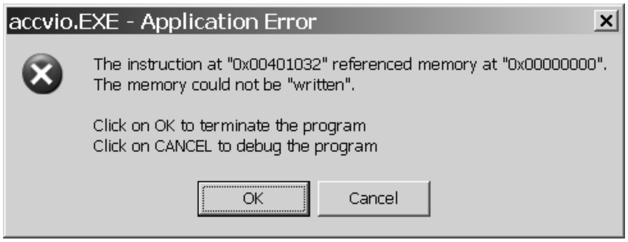
accvio.exe has generated errors and will be closed by Windows.
You will need to restart the program.

An error log is being created.

Figure 3-10 Windows 2000 Dr. Watson error message

The Dr. Watson process remains until the message box is dismissed, which is why on Windows 2000 Server systems visual notification is turned off by default. This default is used because if a server application fails, there is usually nobody at the console to see it and dismiss the message box. Instead, server applications should log errors to the Windows event log.

On Windows 2000, if the Auto value is set to zero, the message box shown in Figure 3-11 is displayed.



accvio.EXE - Application Error

The instruction at "0x00401032" referenced memory at "0x00000000".
The memory could not be "written".

Click on OK to terminate the program
Click on CANCEL to debug the program

Figure 3-11 Windows 2000 Unhandled exception message

If the OK button is clicked, the process exits. If Cancel is clicked, the system defined debugger process (specified by the Debugger's value in the registry path referred to earlier) is launched.

EXPERIMENT: Unhandled Exceptions

To see a sample Dr. Watson log file, download and run the program Accvio.exe, which you can download from *www.sysinternals.com/windowsinternals.shtml*. This program generates a memory access violation by attempting to write to address 0, which is always an invalid address in Windows processes. (See Table 7-6 in Chapter 7.)

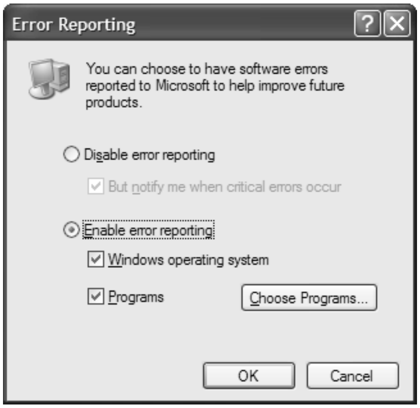
1. Run the Registry Editor, and locate HKLM\SOFTWARE\ Microsoft\Windows NT\CurrentVersion\AeDebug.
2. If the Debugger value is "drwtsn32 -p %ld -e %ld –g", your system is set up to run Dr. Watson as the default debugger. Proceed to step 4.
3. If the value of Debugger was not set up to run Drwtsn32.exe, you can still test Dr. Watson by temporarily installing it and then restoring your previous debugger settings:
    a. Save the current value somewhere (for example, in a Notepad file or in the current paste buffer).
    b. Select Run from the taskbar Start menu, and then type **drwtsn32 –i**. (This initializes the Debugger field to run Dr. Watson.)
3. Run the test program Accvio.exe.
4. You should see one of the message boxes described earlier (depending on which version of Windows you are running).
5. If you have the default Dr. Watson settings, you should now be able to examine the log file and dump file in the dump file directory. To see the configuration settings for Dr. Watson, run *drwtsn32* with no additional arguments. (Select Run from the Start menu, and then type **drwtsn32**.)
6. Alternatively, in the list of Application Errors displayed by Dr. Watson, click on the last entry and then click the View button—the portion of the Dr. Watson log file containing the details of the access violation from Accvio.exe will be displayed. (For details on the log file format, open the help in Dr. Watson and select Dr. Watson Log File Overview.)
7. If the original value of Debugger wasn't the default Dr. Watson settings, restore the saved value from step 1.

As another experiment, try changing the value of Debugger to another program, such as Notepad.exe (Notepad editor) or Sol.exe (Solitaire). Rerun Accvio.exe, and notice that whatever program is specified in the Debugger value is run—that is, there's no validation that the program defined in Debugger is actually a debugger. Make sure you restore your registry settings. (As noted in step 3b, to reset to the system default Dr. Watson settings, type **drwtsn32 –i** in the Run dialog box or at a command prompt.)

Windows Error Reporting

Windows XP and Windows Server 2003 have a new, more sophisticated error-reporting mechanism called Windows Error Reporting that automates the submission of both user-mode process crashes as well as kernel-mode system crashes. (For a description of how this applies to system crashes, see Chapter 14).

Windows Error Reporting can be configured by going to My Computer, selecting Properties, Advanced, and then Error Reporting (which brings up the dialog box shown in Figure 3-12) or by local or domain group policy settings under System, Error Reporting. These settings are stored in the registry under the key HKLM\Software\Microsoft\PCHealth\ErrorReporting.



You can choose to have software errors reported to Microsoft to help improve future products.

○ Disable error reporting

☑ But notify me when critical errors occur

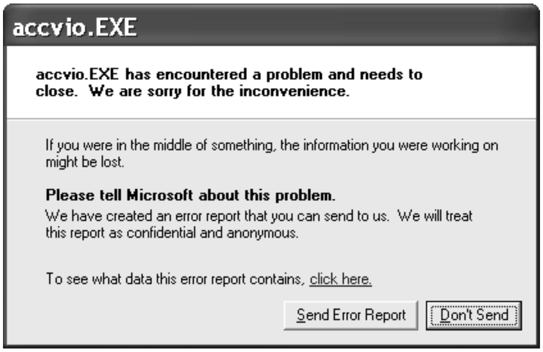
● Enable error reporting

☑ Windows operating system

☑ Programs

Figure 3-12 Error Reporting Configuration dialog box

When an unhandled exception is caught by the unhandled exception filter (described in the previous section), an initial check is made to see whether or not to initiate Windows Error Reporting. If the registry value HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\AeDebug\Auto is set to zero or the Debugger string contains the text "Drwtsn32", the unhandled exception filter loads \Windows\System32\Faultrep.dll into the failing process and calls its *ReportFault* function. *ReportFault* then checks the error-reporting configuration stored under HKLM\Software\Microsoft\PCHealth\ErrorReporting to see whether this process crash should be reported, and if so, how. In the normal case, *ReportFault* creates a process running \Windows\System32\Dwwin.exe, which displays a message box announcing the process crash along with an option to submit the error report to Microsoft as seen in Figure 3-13.


accvio.EXE

accvio.EXE has encountered a problem and needs to close. We are sorry for the inconvenience.

If you were in the middle of something, the information you were working on might be lost.

Please tell Microsoft about this problem.
We have created an error report that you can send to us. We will treat this report as confidential and anonymous.

To see what data this error report contains, click here.

Figure 3-13 Windows Error Reporting dialog box

If the Send Error Report button is clicked, the error report (a minidump and a text file with details on the DLL version numbers loaded in the process) is sent to Microsoft's online crash analysis server, Watson.Microsoft.com. (Unlike kernel mode system crashes, in this situation there is no way to find out whether a solution is available at the time of the report submission.) Then the unhandled exception filter creates a process to run the system-defined debugger (normally Drwtsn32.exe), which by default creates its own dump file and log entry. Unlike Windows 2000, the dump file is a minidump, not a full dump. So, in the case where a full process memory dump is needed to debug a failing application, you can change the configuration of Dr. Watson by running it with no command-line arguments as described in the previous section.

In environments where systems are not connected to the Internet or where the administrator wants to control which error reports are submitted to Microsoft, the destination for the error report can be configured to be an internal file server. Microsoft provides to qualified customers a tool set called Corporate Error Reporting that understands the directory structure created by Windows Error Reporting and provides the administrator with the option to take selective error reports and submit them to Microsoft. (For more information, see *http://www.microsoft.com/resources/satech/cer*.)

System Service Dispatching

As Figure 3-1 illustrated, the kernel's trap handlers dispatch interrupts, exceptions, and system service calls. In the preceding sections, you've seen how interrupt and exception handling work; in this section, you'll learn about system services. A system service dispatch is triggered as a result of executing an instruction assigned to system service dispatching. The instruction that Windows uses for system service dispatching depends on the processor on which it's executing.

32-Bit System Service Dispatching

On x86 processors prior to the Pentium II, Windows uses the *int 0x2e* instruction (46) decimal, which results in a trap. Windows fills in entry 46 in the IDT to point to the system service dispatcher. (Refer to Table 3-1.) The trap causes the executing thread to transition into kernel mode and enter the system service dispatcher. A numeric argument passed in the EAX processor register indicates the system service number being requested. The EBX register points to the list of parameters the caller passes to the system service.

On x86 Pentium II processors and higher, Windows uses the special *sysenter* instruction, which Intel defined specifically for fast system service dispatches. To support the instruction, Windows stores at boot time the address of the kernel's system service dispatcher routine in a register associated with the instruction. The execution of the instruction causes the change to kernel-mode and execution of the system service dispatcher. The system service number is passed in the EAX processor register, and the EDX register points to the list of caller arguments. To return to user-mode, the system service dispatcher usually executes the *sysexit* instruction. (In some cases, like when the single-step flag is enabled on the processor, the system service dispatcher uses the *iretd* instruction instead.)

On K6 and higher 32-bit AMD processors, Windows uses the special *syscall* instruction, which functions similar to the x86 *sysenter* instruction, with Windows configuring a *syscall*-associated processor register with the address of the kernel's system service dispatcher. The system call number is passed in the EAX register, and the stack stores the caller arguments. After completing the dispatch, the kernel executes the *sysret* instruction.

At boot time, Windows detects the type of processor on which it's executing and sets up the appropriate system call code to be used. The system service code for NtReadFile in user mode looks like this:

```
ntdll!NtReadFile:
77f5bfa8 b8b7000000       mov     eax,0xb7
77f5bfad ba0003fe7f       mov     edx,0x7ffe0300
77f5bfb2 ffd2             call    edx
77f5bfb4 c22400           ret     0x24
```

The system service number is 0xb7 (183 in decimal) and the *call* instruction executes the system service dispatch code set up by the kernel, which in this example is at address 0x7ffe0300. Because this was taken from a Pentium M, it uses *sysenter*:

```
SharedUserData!SystemCallStub:
7ffe0300 8bd4             mov     edx,esp
7ffe0302 0f34             sysenter
7ffe0304 c3               ret
```

64-Bit System Service Dispatching

On the x64 architecture, Windows uses the syscall instruction, which functions like the AMD K6's *syscall* instruction, for system service dispatching, passing the system call number in the EAX register, the first four parameters in registers, and any parameters beyond those four on the stack:

```
ntdll!NtReadFile:
00000000`77f9fc60 4c8bd1          mov     r10,rcx
00000000`77f9fc63 b8bf000000      mov     eax,0xbf
00000000`77f9fc68 0f05            syscall
00000000`77f9fc6a c3              ret
```

On the IA64 architecture, Windows uses the *epc* (Enter Privileged Mode) instruction. The first eight system call arguments are passed in registers, and the rest are passed on the stack.

Kernel-Mode System Service Dispatching

As Figure 3-14 illustrates, the kernel uses this argument to locate the system service information in the *system service dispatch table*. This table is similar to the interrupt dispatch table described earlier in the chapter except that each entry contains a pointer to a system service rather than to an interrupt handling routine.

Note System service numbers can change between service packs—Microsoft occasionally adds or removes system services, and the system service numbers are generated automatically as part of a kernel compile.

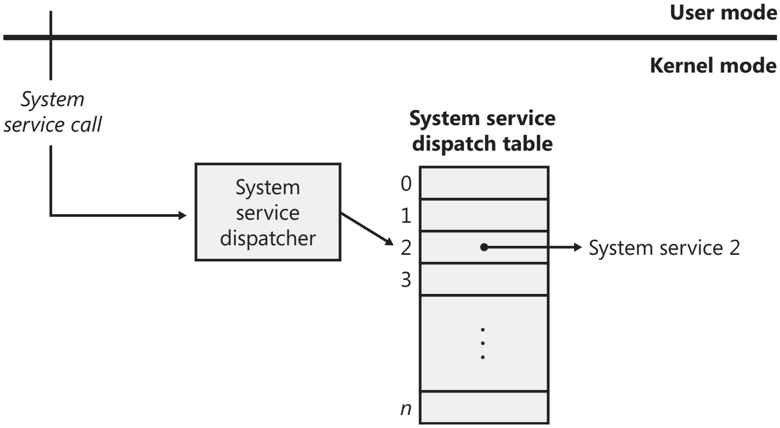


Figure 3-14 System service exceptions

The system service dispatcher, *KiSystemService*, copies the caller's arguments from the thread's user-mode stack to its kernel-mode stack (so that the user can't change the arguments as the kernel is accessing them), and then executes the system service. If the arguments passed to a system service point to buffers in user space, these buffers must be probed for accessibility before kernel-mode code can copy data to or from them.

As you'll see in Chapter 6, each thread has a pointer to its system service table. Windows has two built-in system service tables, but up to four are supported. The system service dispatcher determines which table contains the requested service by interpreting a 2-bit field in the 32-

bit system service number as a table index. The low 12 bits of the system service number serve as the index into the table specified by the table index. The fields are shown in Figure 3-15.

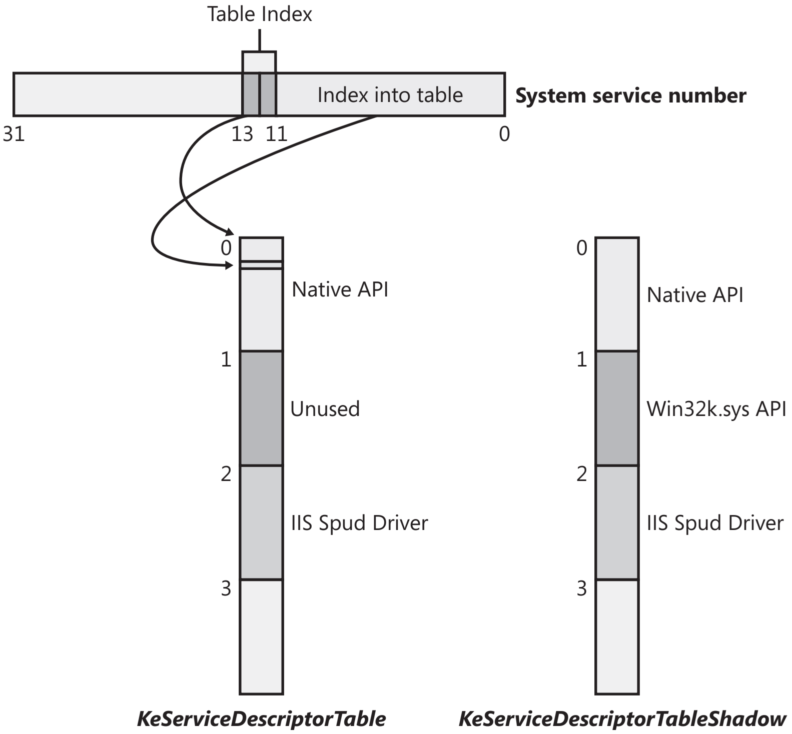


Figure 3-15 System service number to system service translation

Service Descriptor Tables

A primary default array table, *KeServiceDescriptorTable*, defines the core executive system services implemented in Ntosrknl.exe. The other table array, *KeServiceDescriptorTableShadow*, includes the Windows USER and GDI services implemented in the kernel-mode part of the Windows subsystem, Win32k.sys. The first time a Windows thread calls a Windows USER or GDI service, the address of the thread's system service table is changed to point to a table that includes the Windows USER and GDI services. The *KeAddSystemServiceTable* function allows Win32k.sys and other device drivers to add system service tables. If you install Internet Information Services (IIS) on Windows 2000, its support driver (Spud.sys) upon loading defines an additional service table, leaving only one left for definition by third parties. With the exception of the Win32k.sys service table, a service table added with *KeAddSystemServiceTable* is copied into both the *KeServiceDescriptorTable* array and the *KeServiceDescriptorTableShadow* array. Windows supports the addition of only two system service tables beyond the core and Win32 tables.

Note Windows Server 2003 service pack 1 and higher does not support adding additional system service tables beyond that added by Win32k.sys, so adding system service tables is not a way to extend the functionality of those systems.

The system service dispatch instructions for Windows executive services exist in the system library Ntdll.dll. Subsystem DLLs call functions in Ntdll to implement their documented functions. The exception is Windows USER and GDI functions, in which the system service dispatch instructions are implemented directly in User32.dll and Gdi32.dll—there is no Ntdll.dll involved. These two cases are shown in Figure 3-16.

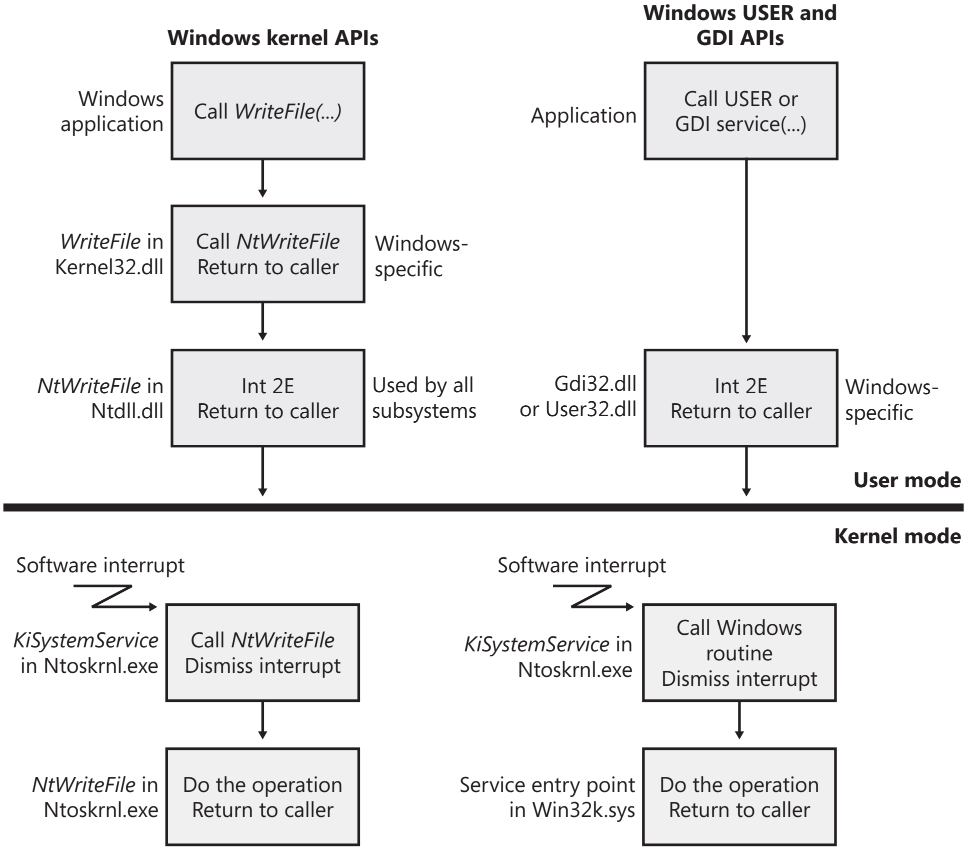


Figure 3-16 System service dispatching

As shown in Figure 3-16, the Windows *WriteFile* function in Kernel32.dll calls the *NtWriteFile* function in Ntdll.dll, which in turn executes the appropriate instruction to cause a system service trap, passing the system service number representing *NtWriteFile*. The system service dispatcher (function *KiSystemService* in Ntoskrnl.exe) then calls the real *NtWriteFile* to process the I/O request. For Windows USER and GDI functions, the system service dispatch calls functions in the loadable kernel-mode part of the Windows subsystem, Win32k.sys.

EXPERIMENT: Viewing System Service Activity

You can monitor system service activity by watching the System Calls/Sec performance counter in the System object. Run the Performance tool, and in chart view, click the Add button to add a counter to the chart; select the System object, select the System Calls/Sec counter, and then click the Add button to add the counter to the chart.

Object Manager

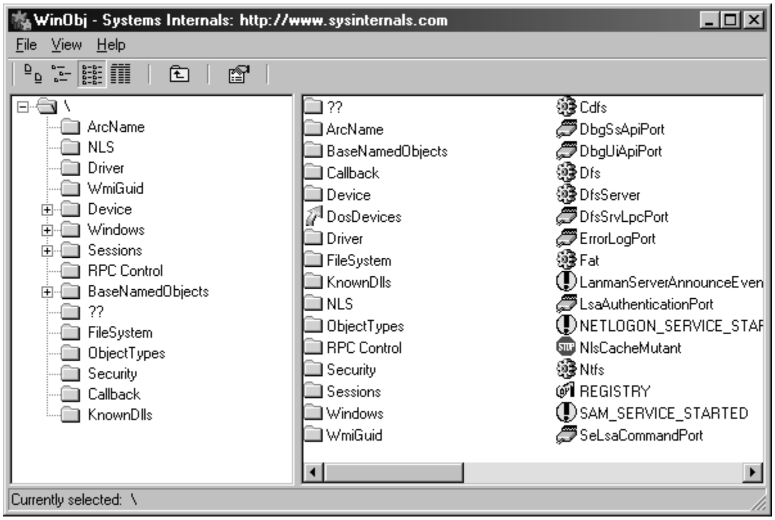
As mentioned in Chapter 2, Windows implements an object model to provide consistent and secure access to the various internal services implemented in the executive. This section describes the Windows *object manager*, the executive component responsible for creating, deleting, protecting, and tracking objects. The object manager centralizes resource control operations that otherwise would be scattered throughout the operating system. It was designed to meet the goals listed on later in the chapter.

EXPERIMENT: Exploring the Object Manager

Throughout this section, you'll find experiments that show you how to peer into the object manager database. These experiments use the following tools, which you should become familiar with if you aren't already:

- Winobj (available from *www.sysinternals.com*) displays the internal object manager's namespace. There is also a version of Winobj in the Platform SDK (in \Program Files\Microsoft Platform SDK\Bin\Winnt\Winobj.exe), but the Winobj from *www.sysinternals.com* displays more accurate information about objects (such as the reference count, the number of open handles, security descriptors, and so forth).
- Process Explorer and Handle from *www.sysinternals.com* (introduced in Chapter 1) displays the open handles for a process.
- Oh.exe (available in Windows resource kits) displays the open handles for a process, but it requires a global flag to be set in order to operate.
- The *Openfiles /query* command (in Windows XP and Windows Server 2003) displays the open handles for a process, but it requires a global flag to be set in order to operate.
- The kernel debugger *!handle* command displays the open handles for a process.

The object viewer provides a way to traverse the namespace that the object manager maintains. (As we'll explain later, not all objects have names.) Try running the WinObj object manager utility from *www.sysinternals.com* and examining the layout, shown here:



As noted previously, both the OH utility and the *Openfiles /query* command require that a Windows global flag called *maintain objects list* be enabled. (See the "Windows Global Flags" section later in this chapter for more details about global flags.) OH will set the flag if it is not set. If you type **Openfiles /Local**, it will tell you whether the flag is enabled. You can enable it with the *Openfiles /Local ON* command. In either case, you must reboot the system for the setting to take effect. Neither Process Explorer nor Handle from *www.sysinternals.com* require object tracking to be turned on because they use a device driver to obtain the information.

The object manager was designed to meet the following goals:

- Provide a common, uniform mechanism for using system resources
- Isolate object protection to one location in the operating system so that C2 security compliance can be achieved
- Provide a mechanism to charge processes for their use of objects so that limits can be placed on the usage of system resources
- Establish an object-naming scheme that can readily incorporate existing objects, such as the devices, files, and directories of a file system, or other independent collections of objects

- Support the requirements of various operating system environments, such as the ability of a process to inherit resources from a parent process (needed by Windows and POSIX) and the ability to create case-sensitive filenames (needed by POSIX)
- Establish uniform rules for object retention (that is, for keeping an object available until all processes have finished using it)

Internally, Windows has two kinds of objects: *executive objects* and *kernel objects*. Executive objects are objects implemented by various components of the executive (such as the process manager, memory manager, I/O subsystem, and so on). Kernel objects are a more primitive set of objects implemented by the Windows kernel. These objects are not visible to user-mode code but are created and used only within the executive. Kernel objects provide fundamental capabilities, such as synchronization, on which executive objects are built. Thus, many executive objects contain (encapsulate) one or more kernel objects, as shown in Figure 3-17.

Details about the structure of kernel objects and how they are used to implement synchronization are given later in this chapter. In the remainder of this section, we'll focus on how the object manager works and on the structure of executive objects, handles, and handle tables. Here we'll just briefly describe how objects are involved in implementing Windows security access checking; we'll cover this topic thoroughly in Chapter 8.

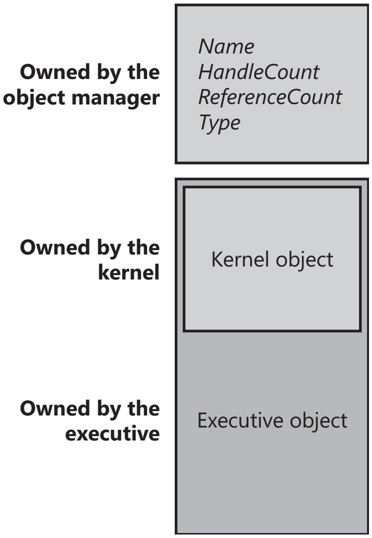


Figure 3-17 Executive objects that contain kernel objects

Executive Objects

Each Windows environment subsystem projects to its applications a different image of the operating system. The executive objects and object services are primitives that the environment subsystems use to construct their own versions of objects and other resources.

Executive objects are typically created either by an environment subsystem on behalf of a user application or by various components of the operating system as part of their normal operation. For example, to create a file, a Windows application calls the Windows *CreateFile* function, implemented in the Windows subsystem DLL Kernel32.dll. After some validation and initialization, *CreateFile* in turn calls the native Windows service *NtCreateFile* to create an executive file object.

The set of objects an environment subsystem supplies to its applications might be larger or smaller than the set the executive provides. The Windows subsystem uses executive objects to export its own set of objects, many of which correspond directly to executive objects. For example, the Windows mutexes and semaphores are directly based on executive objects (which are in turn based on corresponding kernel objects). In addition, the Windows subsystem supplies named pipes and mailslots, resources that are based on executive file objects. Some subsystems, such as POSIX, don't support objects as objects at all. The POSIX subsystem uses executive objects and services as the basis for presenting POSIX-style processes, pipes, and other resources to its applications.

Table 3-3 lists the primary objects the executive provides and briefly describes what they represent. You can find further details on executive objects in the chapters that describe the related executive components (or in the case of executive objects directly exported to Windows, in the Windows API reference documentation).

> **Note** The executive implements a total of 27 object types in Windows 2000 and 29 on Windows XP and Windows Server 2003. (These newer Windows versions add the *DebugObject* and *KeyedEvent* objects.) Many of these objects are for use only by the executive component that defines them and are not directly accessible by Windows APIs. Examples of these objects include *Driver*, *Device*, and *EventPair*.

Table 3-3 Executive Objects Exposed to the Windows API

Object Type	Represents
Symbolic link	A mechanism for referring to an object name indirectly.
Process	The virtual address space and control information necessary for the execution of a set of thread objects.
Thread	An executable entity within a process.
Job	A collection of processes manageable as a single entity through the job.
Section	A region of shared memory (known as a file mapping object in Windows).
File	An instance of an opened file or an I/O device.
Access token	The security profile (security ID, user rights, and so on) of a process or a thread.
Event	An object with a persistent state (signaled or not signaled) that can be used for synchronization or notification.
Semaphore	A counter that provides a resource gate by allowing some maximum number of threads to access the resources protected by the semaphore.
Mutex*	A synchronization mechanism used to serialize access to a resource.
Timer	A mechanism to notify a thread when a fixed period of time elapses.

Table 3-3 Executive Objects Exposed to the Windows API

Object Type	Represents
IoCompletion	A method for threads to enqueue and dequeue notifications of the completion of I/O operations (known as an I/O completion port in the Windows API).
Key	A mechanism to refer to data in the registry. Although keys appear in the object manager namespace, they are managed by the configuration manager, in a way similar to that in which file objects are managed by file system drivers. Zero or more key values are associated with a key object; key values contain data about the key.
WindowStation	An object that contains a clipboard, a set of global atoms, and a group of desktop objects.
Desktop	An object contained within a window station. A desktop has a logical display surface and contains windows, menus, and hooks.

Note Externally in the Windows API, mutants are called *mutexes*. Internally, the kernel object that underlies mutexes is called a *mutant*.

Object Structure

As shown in Figure 3-18, each object has an object header and an object body. The object manager controls the object headers, and the owning executive components control the object bodies of the object types they create. In addition, each object header points to the list of processes that have the object open and to a special object called the *type object* that contains information common to each instance of the object.

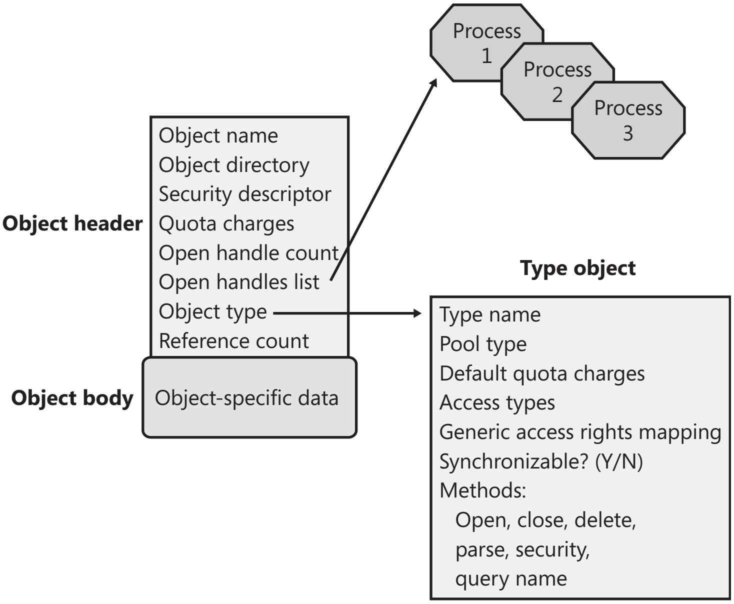


Figure 3-18 Structure of an object

Object Headers and Bodies

The object manager uses the data stored in an object's header to manage objects without regard to their type. Table 3-4 briefly describes the object header attributes.

Table 3-4 Standard Object Header Attributes

Attribute	Purpose
Object name	Makes an object visible to other processes for sharing
Object directory	Provides a hierarchical structure in which to store object names
Security descriptor	Determines who can use the object and what they can do with it (Note: it might be null for objects without a name.)
Quota charges	Lists the resource charges levied against a process when it opens a handle to the object
Open handle count	Counts the number of times a handle has been opened to the object
Open handles list	Points to the list of processes that have opened handles to the object (not present for all objects)
Object type	Points to a type object that contains attributes common to objects of this type
Reference count	Counts the number of times a kernel-mode component has referenced the address of the object

In addition to an object header, each object has an object body whose format and contents are unique to its object type; all objects of the same type share the same object body format. By creating an object type and supplying services for it, an executive component can control the manipulation of data in all object bodies of that type.

The object manager provides a small set of generic services that operate on the attributes stored in an object's header and can be used on objects of any type (although some generic services don't make sense for certain objects). These generic services, some of which the Windows subsystem makes available to Windows applications, are listed in Table 3-5.

Although these generic object services are supported for all object types, each object has its own create, open, and query services. For example, the I/O system implements a create file service for its file objects, and the process manager implements a create process service for its process objects. Although a single create object service could have been implemented, such a routine would have been quite complicated, because the set of parameters required to initialize a file object, for example, differs markedly from that required to initialize a process object. Also, the object manager would have incurred additional processing overhead each time a thread called an object service to determine the type of object the handle referred to and to call the appropriate version of the service. For these reasons and others, the create, open, and query services are implemented separately for each object type.

Table 3-5 **Generic Object Services**

Service	Purpose
Close	Closes a handle to an object
Duplicate	Shares an object by duplicating a handle and giving it to another process
Query object	Gets information about an object's standard attributes
Query security	Gets an object's security descriptor
Set security	Changes the protection on an object
Wait for a single object	Synchronizes a thread's execution with one object
Wait for multiple objects	Synchronizes a thread's execution with multiple objects

Type Objects

Object headers contain data that is common to all objects but that can take on different values for each instance of an object. For example, each object has a unique name and can have a unique security descriptor. However, objects also contain some data that remains constant for all objects of a particular type. For example, you can select from a set of access rights specific to a type of object when you open a handle to objects of that type. The executive supplies terminate and suspend access (among others) for thread objects and read, write, append, and delete access (among others) for file objects. Another example of an object-type-specific attribute is synchronization, which is described shortly.

To conserve memory, the object manager stores these static, object-type-specific attributes once when creating a new object type. It uses an object of its own, a type object, to record this data. As Figure 3-19 illustrates, if the object-tracking debug flag (described in the "Windows Global Flags" section later in this chapter) is set, a type object also links together all objects of the same type (in this case the Process type), allowing the object manager to find and enumerate them, if necessary.

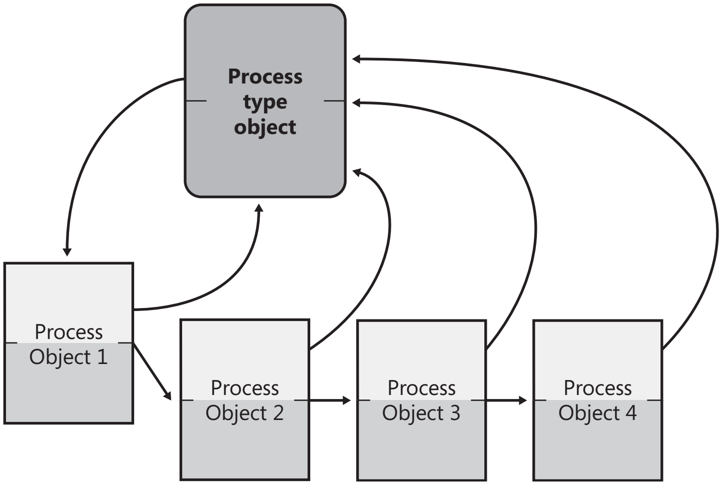
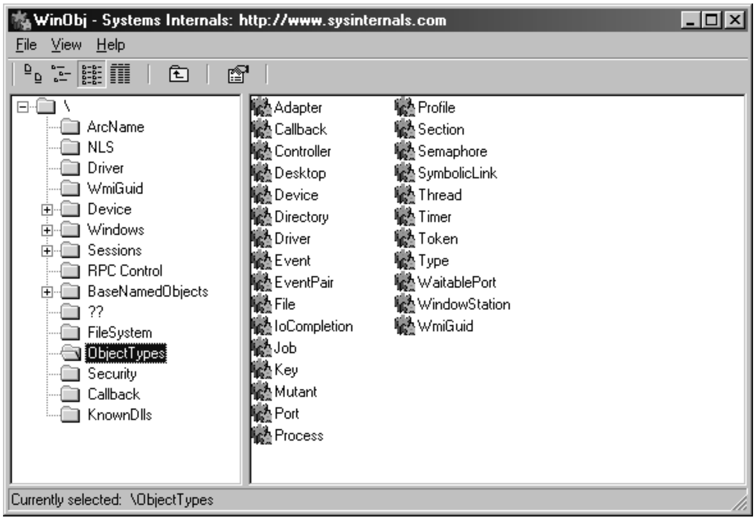


Figure 3-19 Process objects and the process type object

EXPERIMENT: Viewing Object Headers and Type Objects

You can see the list of type objects declared to the object manager with the Winobj tool from *www.sysinternals.com*. After running Winobj, open the \ObjectTypes directory, as shown here:



You can look at the process object type data structure in the kernel debugger by first identifying a process object with the *!process* command:

```
kd> !process 0 0
**** NT ACTIVE PROCESS DUMP ****
PROCESS 8a4ce668  SessionId: none  Cid: 0004    Peb: 00000000  ParentCid: 0000
    DirBase: 00039000  ObjectTable: e1001c88  HandleCount: 474.
    Image: System
```

Then execute the !object command with the process object address as the argument:

```
kd> !object 8a4ce668
Object: 8a4ce668  Type: (8a4ceca0) Process
    ObjectHeader: 8a4ce650
    HandleCount: 2  PointerCount: 89
```

Notice that the object header starts 0x18 (24 decimal) bytes prior to the start of the object body. You can view the object header with this command:

```
kd> dt _object_header 8a4ce650
nt!_OBJECT_HEADER
   +0x000 PointerCount     : 79
   +0x004 HandleCount      : 2
   +0x004 NextToFree       : 0x00000002
   +0x008 Type             : 0x8a4ceca0
   +0x00c NameInfoOffset   : 0 ''
   +0x00d HandleInfoOffset : 0 ''
   +0x00e QuotaInfoOffset  : 0 ''
   +0x00f Flags            : 0x22 ’"‘
   +0x010 ObjectCreateInfo : 0x80545620
   +0x010 QuotaBlockCharged : 0x80545620
   +0x014 SecurityDescriptor : 0xe10001dc
   +0x018 Body             : _QUAD
```

Now look at the object type data structure by obtaining its address from the Type field of the object header data structure:

```
kd> dt _object_type 8a4ceca0
ntdll!_OBJECT_TYPE
   +0x000 Mutex            : _ERESOURCE
   +0x038 TypeList         : _LIST_ENTRY [ 0x8a4cecd8 - 0x8a4cecd8 ]
   +0x040 Name             : _UNICODE_STRING “Process"
   +0x048 DefaultObject    : (null)
   +0x04c Index            : 5
   +0x050 TotalNumberOfObjects : 0x30
   +0x054 TotalNumberOfHandles : 0x1b4
   +0x058 HighWaterNumberOfObjects : 0x3f
   +0x05c HighWaterNumberOfHandles : 0x1b8
   +0x060 TypeInfo         : _OBJECT_TYPE_INITIALIZER
   +0x0ac Key              : 0x636f7250
   +0x0b0 ObjectLocks      : [4] _ERESOURCE
```

The output shows that the object type structure includes the name of the object type, tracks the total number of active objects of that type, and tracks the peak number of handles and objects of that type. The TypeInfo field stores the pointer to the data structure that stores attributes common to all objects of the object type as well as pointers to the object type’s methods:

```
kd> dt _object_type_initializer 8a4ceca0+60
ntdll!_OBJECT_TYPE_INITIALIZER
   +0x000 Length           : 0x4c
   +0x002 UseDefaultObject : 0 ’’
   +0x003 CaseInsensitive  : 0 ’’
   +0x004 InvalidAttributes : 0xb0
   +0x008 GenericMapping   : _GENERIC_MAPPING
   +0x018 ValidAccessMask  : 0x1f0fff
   +0x01c SecurityRequired : 0x1 ’’
   +0x01d MaintainHandleCount : 0 ’’
   +0x01e MaintainTypeList : 0 ’’
   +0x020 PoolType         : 0 ( NonPagedPool )
   +0x024 DefaultPagedPoolCharge : 0x1000
   +0x028 DefaultNonPagedPoolCharge : 0x288
   +0x02c DumpProcedure    : (null)
   +0x030 OpenProcedure    : (null)
   +0x034 CloseProcedure   : (null)
   +0x038 DeleteProcedure  : 0x805abe6e     nt!PspProcessDelete+0
   +0x03c ParseProcedure   : (null)
   +0x040 SecurityProcedure : 0x805cf682     nt!SeDefaultObjectMethod+0
   +0x044 QueryNameProcedure : (null)
   +0x048 OkayToCloseProcedure :  (null)
```

Type objects can't be manipulated from user mode because the object manager supplies no services for them. However, some of the attributes they define are visible through certain native services and through Windows API routines. The attributes stored in the type objects are described in Table 3-6.

Table 3-6 Type Object Attributes

Attribute	Purpose
Type name	The name for objects of this type ("process," "event," "port," and so on)
Pool type	Indicates whether objects of this type should be allocated from paged or nonpaged memory
Default quota charges	Default paged and nonpaged pool values to charge to process quotas
Access types	The types of access a thread can request when opening a handle to an object of this type ("read," "write," "terminate," "suspend," and so on)
Generic access rights mapping	A mapping between the four generic access rights (read, write, execute, and all) to the type-specific access rights
Synchronization	Indicates whether a thread can wait for objects of this type
Methods	One or more routines that the object manager calls automatically at certain points in an object's lifetime

Synchronization, one of the attributes visible to Windows applications, refers to a thread's ability to synchronize its execution by waiting for an object to change from one state to another. A thread can synchronize with executive job, process, thread, file, event, semaphore, mutex, and timer objects. Other executive objects don't support synchronization. An object's ability to support synchronization is based on whether the object contains an embedded dispatcher object, a kernel object that is covered in the section "Low-IRQL Synchronization" later in this chapter.

Object Methods

The last attribute in Table 3-6, methods, comprises a set of internal routines that are similar to C++ constructors and destructors—that is, routines that are automatically called when an object is created or destroyed. The object manager extends this idea by calling an object method in other situations as well, such as when someone opens or closes a handle to an object or when someone attempts to change the protection on an object. Some object types specify methods, whereas others don't, depending on how the object type is to be used.

When an executive component creates a new object type, it can register one or more methods with the object manager. Thereafter, the object manager calls the methods at well-defined points in the lifetime of objects of that type, usually when an object is created, deleted, or modified in some way. The methods that the object manager supports are listed in Table 3-7.

Table 3-7 **Object Methods**

Method	When Method Is Called
Open	When an object handle is opened
Close	When an object handle is closed
Delete	Before the object manager deletes an object
Query name	When a thread requests the name of an object, such as a file, that exists in a secondary object namespace
Parse	When the object manager is searching for an object name that exists in a secondary object namespace
Security	When a process reads or changes the protection of an object, such as a file, that exists in a secondary object namespace

The object manager calls the open method whenever it creates a handle to an object, which it does when an object is created or opened. However, only one object type, the Windowstation, defines an open method. The Windowstation object type requires an open method so that Win32k.sys can share a piece of memory with the process that serves as a desktop-related memory pool.

An example of the use of a close method occurs in the I/O system. The I/O manager registers a close method for the file object type, and the object manager calls the close method each time it closes a file object handle. This close method checks whether the process that is closing the file handle owns any outstanding locks on the file and, if so, removes them. Checking for file locks isn't something the object manager itself could or should do.

The object manager calls a delete method, if one is registered, before it deletes a temporary object from memory. The memory manager, for example, registers a delete method for the section object type that frees the physical pages being used by the section. It also verifies that any internal data structures the memory manager has allocated for a section are deleted before the section object is deleted. Once again, the object manager can't do this work because it knows nothing about the internal workings of the memory manager. Delete methods for other types of objects perform similar functions.

The parse method (and similarly, the query name method) allows the object manager to relinquish control of finding an object to a secondary object manager if it finds an object that exists outside the object manager namespace. When the object manager looks up an object name, it suspends its search when it encounters an object in the path that has an associated parse method. The object manager calls the parse method, passing to it the remainder of the object name it is looking for. There are two namespaces in Windows in addition to the object manager's: the registry namespace, which the configuration manager implements, and the file system namespace, which the I/O manager implements with the aid of file system drivers. (See Chapter 5 for more information on the configuration manager and Chapter 9 for more about the I/O manager and file system drivers.)

For example, when a process opens a handle to the object named \Device\Floppy0\docs\resume.doc, the object manager traverses its name tree until it reaches the device object named Floppy0. It sees that a parse method is associated with this object, and it calls the method, passing to it the rest of the object name it was searching for—in this case, the string \docs\resume.doc. The parse method for device objects is an I/O routine because the I/O manager defines the device object type and registers a parse method for it. The I/O manager's parse routine takes the name string and passes it to the appropriate file system, which finds the file on the disk and opens it.

The security method, which the I/O system also uses, is similar to the parse method. It is called whenever a thread tries to query or change the security information protecting a file. This information is different for files than for other objects because security information is stored in the file itself rather than in memory. The I/O system, therefore, must be called to find the security information and read or change it.

Object Handles and the Process Handle Table

When a process creates or opens an object by name, it receives a *handle* that represents its access to the object. Referring to an object by its handle is faster than using its name because the object manager can skip the name lookup and find the object directly. Processes can also acquire handles to objects by inheriting handles at process creation time (if the creator specifies the inherit handle flag on the *CreateProcess* call and the handle was marked as inheritable, either at the time it was created or afterward by using the Windows *SetHandleInformation* function) or by receiving a duplicated handle from another process. (See the Windows *DuplicateHandle* function.)

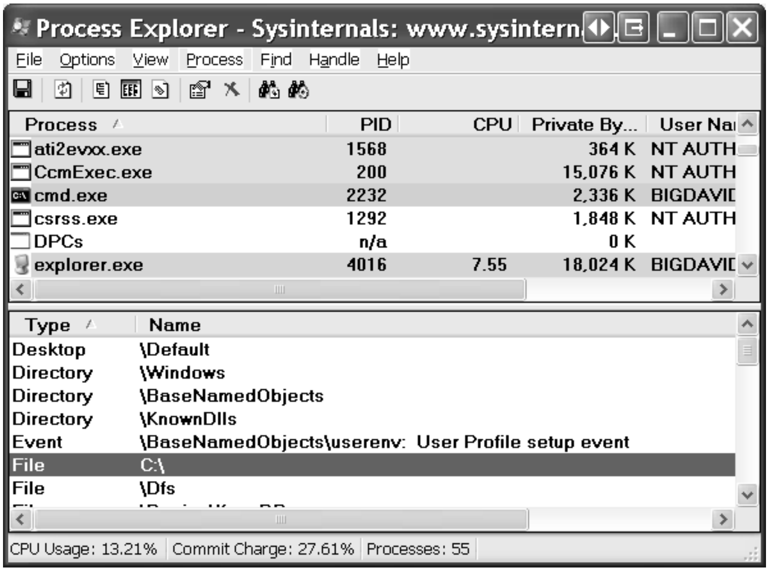
All user-mode processes must own a handle to an object before their threads can use the object. Using handles to manipulate system resources isn't a new idea. C and Pascal (and other language) run-time libraries, for example, return handles to opened files. Handles serve as indirect pointers to system resources; this indirection keeps application programs from fiddling directly with system data structures.

Note Executive components and device drivers can access objects directly because they are running in kernel mode and therefore have access to the object structures in system memory. However, they must declare their usage of the object by incrementing the reference count so that the object won't be deallocated while it's still being used. (See the section "Object Retention" later in this chapter for more details.)

Object handles provide additional benefits. First, except for what they refer to, there is no difference between a file handle, an event handle, and a process handle. This similarity provides a consistent interface to reference objects, regardless of their type. Second, the object manager has the exclusive right to create handles and to locate an object that a handle refers to. This means that the object manager can scrutinize every user-mode action that affects an object to see whether the security profile of the caller allows the operation requested on the object in question.

EXPERIMENT: Viewing Open Handles

Run Process Explorer, and make sure the lower pane is enabled and configured to show open handles. (Click on View, Lower Pane View, and then Handles). Then open a command prompt and view the handle table for the new Cmd.exe process. You should see an open file handle to the current directory. For example, assuming the current directory is C:\, Process Explorer shows the following:



Process	PID	CPU	Private By...	User Na
ati2evxx.exe	1568		364 K	NT AUTH
CcmExec.exe	200		15,076 K	NT AUTH
cmd.exe	2232		2,336 K	BIGDAVIC
csrss.exe	1292		1,848 K	NT AUTH
DPCs	n/a		0 K	
explorer.exe	4016	7.55	18,024 K	BIGDAVIC

Type	Name
Desktop	\Default
Directory	\Windows
Directory	\BaseNamedObjects
Directory	\KnownDlls
Event	\BaseNamedObjects\userenv: User Profile setup event
File	C:\
File	\Dfs

CPU Usage: 13.21% Commit Charge: 27.61% Processes: 55

If you then change the current directory with the CD command, you will see in Process Explorer that the handle to the previous current directory is closed and a new handle is opened to the new current directory. The previous handle is highlighted briefly in red, and the new handle is highlighted in green. The duration of the highlight can be adjusted by clicking Options and then Difference Highlight Duration.

Process Explorer's differences highlighting feature makes it easy to see changes in the handle table. For example, if a process is leaking handles, viewing the handle table with Process Explorer can quickly show what handle or handles are being opened but not closed. This information can assist the programmer to find the handle leak.

You can also display the open handle table by using the command line Handle tool from *www.sysinternals.com*. For example, note the following partial output of Handle examining the handle table for a Cmd.exe process before and after changing the directory:

```
C:\>handle -p cmd.exe
Handle v2.2
Copyright (C) 1997-2004 Mark Russinovich
Sysinternals - www.sysinternals.com
------------------------------------------------------------
cmd.exe pid: 3184 BIGDAVID\dsolomon
   b0: File          C:\
C:\>cd windows
C:\WINDOWS>handle -p cmd.exe
.
.
cmd.exe pid: 3184 BIGDAVID\dsolomon
   b4: File          C:\WINDOWS
```

An object handle is an index into a process-specific *handle table*, pointed to by the executive process (EPROCESS) block (described in Chapter 6). The first handle index is 4, the second 8, and so on. A process's handle table contains pointers to all the objects that the process has opened a handle to. Handle tables are implemented as a three-level scheme, similar to the way that the x86 memory management unit implements virtual-to-physical address translation, giving a maximum of more than 16,000,000 handles per process. (See Chapter 7 for details about memory management in x86 systems.)

In Windows 2000, when a process is created, the object manager allocates the top level of the handle table, which contains pointers to the middle-level tables; the middle level, which contains the first array of pointers to subhandle tables; and the lowest level, which contains the first subhandle table. Figure 3-20 illustrates the Windows 2000 handle table architecture. In Windows 2000, the object manager treats the low 24 bits of an object handle's value as three 8-bit fields that index into each of the three levels in the handle table. In Windows XP and Windows Server 2003, only the lowest level handle table is allocated on process creation—the other levels are created as needed. In Windows 2000, the subhandle table consists of 255 usable entries. In Windows XP and Windows Server 2003, the subhandle table consists of as many entries as will fit in a page minus one entry that is used for handle auditing. For example, for x86 systems a page is 4096 bytes, divided by the size of a handle table entry (8 bytes), which is 512, minus 1, which is a total of 511 entries in the lowest level handle table. In Windows XP and Windows Server 2003, the mid-level handle table contains a full page of pointers to subhandle tables, so the number of subhandle tables depends on the size of the page and the size of a pointer for the platform.

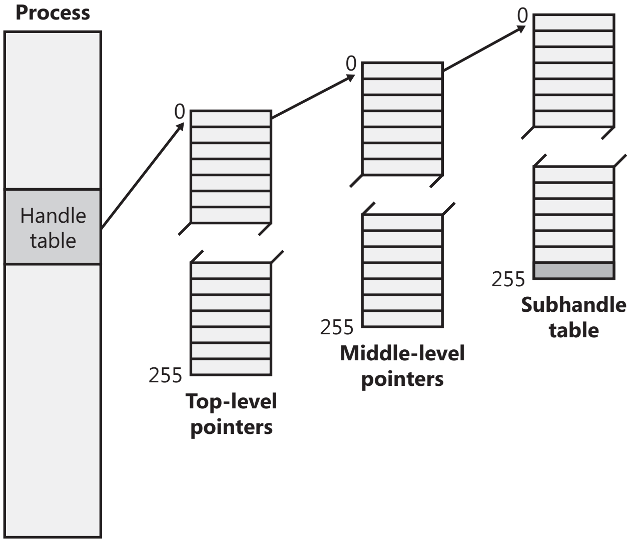


Figure 3-20 Windows 2000 process handle table architecture

EXPERIMENT: Creating the Maximum Number of Handles

The test program Testlimit from *www.sysinternals.com/windowsinternals.shtml* has an option to open handles to an object until it cannot open any more handles. You can use this to see how many handles can be created in a single process on your system. Because handle tables are allocated from paged pool, you might run out of paged pool before you hit the maximum number of handles that can be created in a single process. To see how many handles you can create on your system, follow these steps:

1. Download the Testlimit zip file from the link just mentioned, and unzip it into a directory.
2. Run Process Explorer, and click View and then System Information. Notice the current and maximum size of paged pool. (To display the maximum pool size values, Process Explorer must be configured properly to access the symbols for the kernel image, Ntoskrnl.exe.) Leave this system information display running so that you can see pool utilization when you run the Testlimit program.
3. Open a command prompt.
4. Run the Testlimit program with the "-h" switch (do this by typing **testlimit –h**). When Testlimit fails to open a new handle, it will display the total number of handles it was able to create. If the number is less than approximately 16 million, you are probably running out of paged pool before hitting the theoretical per-process handle limit.
5. Close the command-prompt window; doing this will kill the Testlimit process, thus closing all the open handles.

As shown in Figure 3-21, on x86 systems, each handle entry consists of a structure with two 32-bit members: a pointer to the object (with flags), and the granted access mask. On 64-bit systems, a handle table entry is 12 bytes long: a 64-bit pointer to the object header and a 32-bit access mask. (Access masks are described in Chapter 8.)

On Windows 2000, the first 32-bit member contains both a pointer to the object header and four flags. Because object headers are always 8-byte aligned, the low-order 3 bits of this field are free for use as flags. An entry's high bit is used as a lock. When the object manager translates a handle to an object pointer, it locks the handle entry while the translation is in progress. Because all objects are located in the system address space, the high bit of the object pointer is set. (The addresses are guaranteed to be higher than 0x80000000 even on systems with the /3GB boot switch.) Thus, the object manager can keep the high bit clear when a handle table entry is unlocked and, in the process of locking the entry, set the bit and obtain the object's correct pointer value. The object manager needs to lock a process's entire handle table, using a handle table lock associated with each process, only when the process creates a new handle or closes an existing handle. In Windows XP and Windows Server 2003, the lock bit is the low-order bit of the object pointer. The flag that was stored in this low-order bit in Windows 2000 is now stored in an unused bit in the access mask.

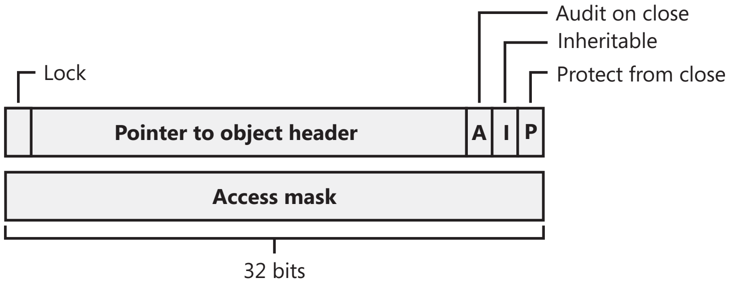


Figure 3-21 Structure of a handle table entry

The first flag indicates whether the caller is allowed to close this handle. The second flag is the inheritance designation—that is, it indicates whether processes created by this process will get a copy of this handle in their handle tables. As already noted, handle inheritance can be specified on handle creation or later with the *SetHandleInformation* function. (This flag can also be specified with the Windows *SetHandleInformation* function.) The third flag indicates whether closing the object should generate an audit message. (This flag isn't exposed to Windows—the object manager uses it internally.)

System components and device drivers often need to open handles to objects that user-mode applications shouldn't have access to. This is done by creating handles in the *kernel handle table* (referenced internally with the name *ObpKernelHandleTable*). The handles in this table are accessible only from kernel mode and in any process context. This means that a kernel-mode function can reference the handle in any process context with no performance impact. The object manager recognizes references to handles from the kernel handle table when the high bit of the handle is set—that is, when references to kernel-handle-table handles have values greater than 0x80000000. On Windows 2000, the kernel-handle table is an independent handle table, but on Windows XP and Windows Server 2003 the kernel-handle table also serves as the handle table for the System process.

EXPERIMENT: Viewing the Handle Table with the Kernel Debugger

The *!handle* command in the kernel debugger takes three arguments:

```
!handle <handle index> <flags> <processid>
```

The handle index identifies the handle entry in the handle table. (Zero means display all handles.) The first handle is index 4, the second 8, and so on. For example, typing **!handle 4** will show the first handle for the current process.

The flags you can specify are a bitmask, where bit 0 means display only the information in the handle entry, bit 1 means display free handles (not just used handles), and bit 2 means display information about the object that the handle refers to. The following command displays full details about the handle table for process ID 0x408:

```
kd> !handle 0 7 408

processor number 0
Searching for Process with Cid == 408
PROCESS 865f0790 SessionId: 0 Cid: 0408 Peb: 7ffdf000 ParentCid: 01dc
    DirBase: 04fd3000  ObjectTable: 856ca888  TableSize:  21.
    Image: i386kd.exe

Handle Table at e2125000 with 21 Entries in use
0000: free handle
0004: Object: e20da2e0  GrantedAccess: 000f001f
Object: e20da2e0  Type: (81491b80) Section
    ObjectHeader: e20da2c8
        HandleCount: 1  PointerCount: 1

0008: Object: 80b13330  GrantedAccess: 00100003
Object: 80b13330  Type: (81495100) Event
    ObjectHeader: 80b13318
        HandleCount: 1  PointerCount: 1
```

Object Security

When you open a file, you must specify whether you intend to read or to write. If you try to write to a file that is opened for read access, you get an error. Likewise, in the executive, when a process creates an object or opens a handle to an existing object, the process must specify a set of *desired access rights*—that is, what it wants to do with the object. It can request either a set of standard access rights (such as read, write, and execute) that apply to all object types or specific access rights that vary depending on the object type. For example, the process can request delete access or append access to a file object. Similarly, it might require the ability to suspend or terminate a thread object.

When a process opens a handle to an object, the object manager calls the *security reference monitor*, the kernel-mode portion of the security system, sending it the process's set of desired access rights. The security reference monitor checks whether the object's security descriptor permits the type of access the process is requesting. If it does, the reference monitor returns a set of *granted access rights* that the process is allowed, and the object manager stores them in the object handle it creates. How the security system determines who gets access to which objects is explored in Chapter 8.

Thereafter, whenever the process's threads use the handle, the object manager can quickly check whether the set of granted access rights stored in the handle corresponds to the usage implied by the object service the threads have called. For example, if the caller asked for read access to a section object but then calls a service to write to it, the service fails.

Object Retention

There are two types of objects: temporary and permanent. Most objects are temporary—that is, they remain while they are in use and are freed when they are no longer needed. Permanent objects remain until they are explicitly freed. Because most objects are temporary, the rest of this section describes how the object manager implements *object retention*—that is, retaining temporary objects only as long as they are in use and then deleting them. Because all user-mode processes that access an object must first open a handle to it, the object manager can easily track how many of these processes, and even which ones, are using an object. Tracking these handles represents one part in implementing retention. The object manager implements object retention in two phases. The first phase is called *name retention*, and it is controlled by the number of open handles to an object that exist. Every time a process opens a handle to an object, the object manager increments the open handle counter in the object's header. As processes finish using the object and close their handles to it, the object manager decrements the open handle counter. When the counter drops to 0, the object manager deletes the object's name from its global namespace. This deletion prevents new processes from opening a handle to the object.

The second phase of object retention is to stop retaining the objects themselves (that is, to delete them) when they are no longer in use. Because operating system code usually accesses objects by using pointers instead of handles, the object manager must also record how many object pointers it has dispensed to operating system processes. It increments a *reference count* for an object each time it gives out a pointer to the object; when kernel-mode components finish using the pointer, they call the object manager to decrement the object's reference count. The system also increments the reference count when it increments the handle count, and likewise decrements the reference count when the handle count decrements, because a handle is also a reference to the object that must be tracked. (For further details on object retention, see the DDK documentation on the functions *ObReferenceObjectByPointer* and *ObDereferenceObject*.)

Figure 3-22 illustrates two event objects that are in use. Process A has the first event open. Process B has both events open. In addition, the first event is being referenced by some kernel-mode structure; thus, the reference count is 3. So even if processes A and B closed their handles to the first event object, it would continue to exist because its reference count is 1. However, when process B closes its handle to the second event object, the object would be deallocated.

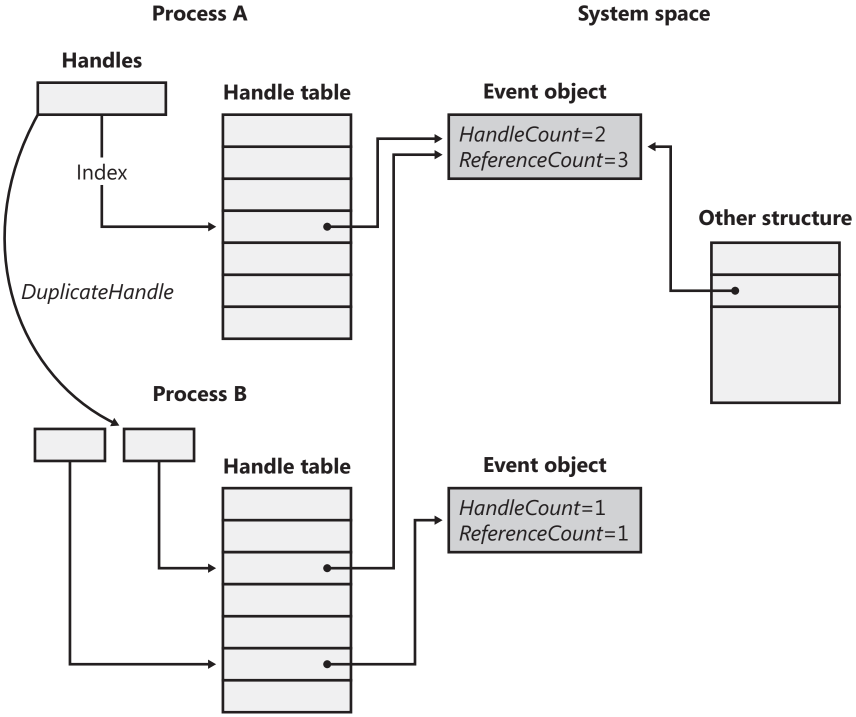


Figure 3-22 Handles and reference counts

So even after an object's open handle counter reaches 0, the object's reference count might remain positive, indicating that the operating system is still using the object. Ultimately, when the reference count drops to 0, the object manager deletes the object from memory.

Because of the way object retention works, an application can ensure that an object and its name remain in memory simply by keeping a handle open to the object. Programmers who write applications that contain two or more cooperating processes need not be concerned that one process might delete an object before the other process has finished using it. In addition, closing an application's object handles won't cause an object to be deleted if the operating system is still using it. For example, one process might create a second process to execute a program in the background; it then immediately closes its handle to the process. Because the operating system needs the second process to run the program, it maintains a reference to its process object. Only when the background program finishes executing does the object manager decrement the second process's reference count and then delete it.

Resource Accounting

Resource accounting, like object retention, is closely related to the use of object handles. A positive open handle count indicates that some process is using that resource. It also indicates that some process is being charged for the memory the object occupies. When an object's handle count and reference count drop to 0, the process that was using the object should no longer be charged for it.

Many operating systems use a quota system to limit processes' access to system resources. However, the types of quotas imposed on processes are sometimes diverse and complicated, and the code to track the quotas is spread throughout the operating system. For example, in some operating systems, an I/O component might record and limit the number of files a process can open, whereas a memory component might impose a limit on the amount of memory a process's threads can allocate. A process component might limit users to some maximum number of new processes they can create or a maximum number of threads within a process. Each of these limits is tracked and enforced in different parts of the operating system.

In contrast, the Windows object manager provides a central facility for resource accounting. Each object header contains an attribute called *quota charges* that records how much the object manager subtracts from a process's allotted paged and/or nonpaged pool quota when a thread in the process opens a handle to the object.

Each process on Windows points to a quota structure that records the limits and current values for nonpaged pool, paged pool, and page file usage. (Type **dt nt!_EPROCESS_QUOTA_ENTRY** in the kernel debugger to see the format of this structure.) These quotas default to 0 (no limit) but can be specified by modifying registry values. (See *NonPagedPoolQuota*, *PagedPoolQuota*, and *PagingFileQuota* under HKLM\System\CurrentControlSet\Session Manager\Memory Management.) Note that all the processes in an interactive session share the same quota block (and there's no documented way to create processes with their own quota blocks).

Object Names

An important consideration in creating a multitude of objects is the need to devise a successful system for keeping track of them. The object manager requires the following information to help you do so:

- A way to distinguish one object from another
- A method for finding and retrieving a particular object

The first requirement is served by allowing names to be assigned to objects. This is an extension of what most operating systems provide—the ability to name selected resources, files, pipes, or a block of shared memory, for example. The executive, in contrast, allows any resource represented by an object to have a name. The second requirement, finding and retrieving an object, is also satisfied by object names. If the object manager stores objects by name, it can find an object by looking up its name.

Object names also satisfy a third requirement, which is to allow processes to share objects. The executive's object namespace is a global one, visible to all processes in the system. One process can create an object and place its name in the global namespace, and a second process can open a handle to the object by specifying the object's name. If an object isn't meant to be shared in this way, its creator doesn't need to give it a name.

To increase efficiency, the object manager doesn't look up an object's name each time someone uses the object. Instead, it looks up a name under only two circumstances. The first is when a process creates a named object: the object manager looks up the name to verify that it doesn't already exist before storing the new name in the global namespace. The second is when a process opens a handle to a named object: the object manager looks up the name, finds the object, and then returns an object handle to the caller; thereafter, the caller uses the handle to refer to the object. When looking up a name, the object manager allows the caller to select either a case-sensitive or a case-insensitive search, a feature that supports POSIX and other environments that use case-sensitive filenames.

Where the names of objects are stored depends on the object type. Table 3-8 lists the standard object directories found on all Windows systems and what types of objects have their names stored there. Of the directories listed, only \BaseNamedObjects and \GLOBAL?? (\?? on Windows 2000) are visible to user programs (see the Session Namespace section later in this chapter for more information).

Because the base kernel objects such as mutexes, events, semaphores, waitable timers, and sections have their names stored in a single object directory, no two of these objects can have the same name, even if they are of a different type. This restriction emphasizes the need to choose names carefully so that they don't collide with other names (for example, prefix names with your company and product name).

Table 3-8 Standard Object Directories

Directory	Types of Object Names Stored
\GLOBAL?? (\?? in Windows 2000)	MS-DOS device names (\DosDevices is a symbolic link to this directory.)
\BaseNamedObjects	Mutexes, events, semaphores, waitable timers, and section objects
\Callback	Callback objects
\Device	Device objects
\Driver	Driver objects
\FileSystem	File system driver objects and file system recognizer device objects
\KnownDlls	Section names and path for known DLLs (DLLs mapped by the system at startup time)
\Nls	Section names for mapped national language support tables
\ObjectTypes	Names of types of objects
\RPC Control	Port objects used by remote procedure calls (RPCs)

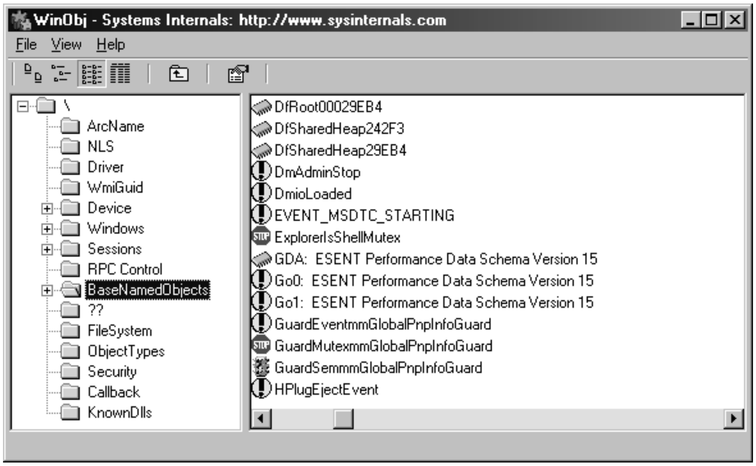
Table 3-8 Standard Object Directories

Directory	Types of Object Names Stored
\Security	Names of objects specific to the security subsystem
\Windows	Windows subsystem ports and window stations

Object names are global to a single computer (or to all processors on a multiprocessor computer), but they're not visible across a network. However, the object manager's parse method makes it possible to access named objects that exist on other computers. For example, the I/O manager, which supplies file object services, extends the functions of the object manager to remote files. When asked to open a remote file object, the object manager calls a parse method, which allows the I/O manager to intercept the request and deliver it to a network redirector, a driver that accesses files across the network. Server code on the remote Windows system calls the object manager and the I/O manager on that system to find the file object and return the information back across the network.

EXPERIMENT: Looking at the Base Named Objects

You can see the list of base objects that have names with the Winobj tool from *www.sysinternals.com*. Run Winobj.exe and click on \BaseNamedObjects, as shown here:


WinObj - Systems Internals: http://www.sysinternals.com
File
View
Help
\
ArcName
NLS
Driver
WmiGuid
Device
Windows
Sessions
RPC Control
BaseNamedObjects
??
FileSystem
ObjectTypes
Security
Callback
KnownDlls
DfRoot00029EB4
DfSharedHeap242F3
DfSharedHeap29EB4
DmAdminStop
DmioLoaded
EVENT_MSDTC_STARTING
ExplorerIsShellMutex
GDA: ESENT Performance Data Schema Version 15
Go0: ESENT Performance Data Schema Version 15
Go1: ESENT Performance Data Schema Version 15
GuardEventmmGlobalPnpInfoGuard
GuardMutexmmGlobalPnpInfoGuard
GuardSemmmGlobalPnpInfoGuard
HPlugEjectEvent

The named objects are shown on the right. The icons indicate the object type.

- Mutexes are indicated with a stop sign.
- Sections (Windows file mapping objects) are shown as memory chips.
- Events are shown as exclamation points.
- Semaphores are indicated with an icon that resembles a traffic signal.
- Symbolic links have icons that are curved arrows.

Object directories The object directory object is the object manager's means for supporting this hierarchical naming structure. This object is analogous to a file system directory and contains the names of other objects, possibly even other object directories. The object directory object maintains enough information to translate these object names into pointers to the objects themselves. The object manager uses the pointers to construct the object handles that it returns to user-mode callers. Both kernel-mode code (including executive components and device drivers) and user-mode code (such as subsystems) can create object directories in which to store objects. For example, the I/O manager creates an object directory named \Device, which contains the names of objects representing I/O devices.

Symbolic links In certain file systems (on NTFS and some UNIX systems, for example), a symbolic link lets a user create a filename or a directory name that, when used, is translated by the operating system into a different file or directory name. Using a symbolic link is a simple method for allowing users to indirectly share a file or the contents of a directory, creating a cross-link between different directories in the ordinarily hierarchical directory structure.

The object manager implements an object called a *symbolic link object*, which performs a similar function for object names in its object namespace. A symbolic link can occur anywhere within an object name string. When a caller refers to a symbolic link object's name, the object manager traverses its object namespace until it reaches the symbolic link object. It looks inside the symbolic link and finds a string that it substitutes for the symbolic link name. It then restarts its name lookup.

One place in which the executive uses symbolic link objects is in translating MS-DOS-style device names into Windows internal device names. In Windows, a user refers to floppy and hard disk drives using the names A:, B:, C:, and so on and serial ports as COM1, COM2, and so on. The Windows subsystem makes these symbolic link objects protected, global data by placing them in the object manager namespace under the \?? object directory on Windows 2000 and the \Global?? directory on Windows XP and Windows Server 2003.

Session Namespace

Windows NT was originally written with the assumption that only one user would log on to the system interactively and that the system would run only one instance of any interactive application. The addition of Windows Terminal Services in Windows 2000 Server and fast user switching in Windows XP changed these assumptions, thus requiring changes to the object manager namespace model to support multiple users. (For a basic description of terminal services and sessions, see Chapter 1.)

A user logged on to the console session has access to the *global* namespace, a namespace that serves as the first instance of the namespace. Additional sessions are given a session-private view of the namespace known as a *local* namespace. The parts of the namespace that are localized for each session include \DosDevices, \Windows, and \BaseNamedObjects. Making separate copies of the same parts of the namespace is known as *instancing* the namespace. Instancing \DosDevices makes it possible for each user to have different network drive letters and Windows objects such as serial ports. On Windows 2000, the global \DosDevices directory is named \?? and is the directory to which the \DosDevices symbolic link points, and local \DosDevices directories are identified by the session id for the terminal server session. On Windows XP and later, the global \DosDevices directory is named \Global?? and is the directory to which \DosDevices points, and local \DosDevices directories are identified by the logon session ID.

The \Windows directory is where Win32k.sys creates the interactive window station, \WinSta0. A Terminal Services environment can support multiple interactive users, but each user needs an individual version of WinSta0 to preserve the illusion that he or she is accessing the predefined interactive window station in Windows. Finally, applications and the system create shared objects in \BaseNamedObjects, including events, mutexes, and memory sections. If two users are running an application that creates a named object, each user session must have a private version of the object so that the two instances of the application don't interfere with one another by accessing the same object.

The object manager implements a local namespace by creating the private versions of the three directories mentioned under a directory associated with the user's session under \Sessions\X (where X is the session identifier). When a Windows application in remote session two creates a named event, for example, the object manager transparently redirects the object's name from \BaseNamedObjects to \Sessions\2\BaseNamedObjects.

All object manager functions related to namespace management are aware of the instanced directories and participate in providing the illusion that nonconsole sessions use the same namespace as the console session. Windows subsystem DLLs prefix names passed by Windows applications that reference objects in \DosDevices with \?? (for example, C:\Windows becomes \??\C:\Windows). When the object manager sees the special \?? prefix, the steps it takes depends on the version of Windows, but it always relies on a field named *DeviceMap* in the executive process object (EPROCESS, which is described further in Chapter 6) that points to a data structure shared by other processes in the same session. The *DosDevicesDirectory* field of the DeviceMap structure points at the object manager directory that represents the process's local \DosDevices. The target directory varies depending on the system:

- If the system is Windows 2000 and Terminal Services are not installed, the *DosDevicesDirectory* field of the DeviceMap structure of the process points at the \?? directory because there are no local namespaces.

- If the system is Windows 2000 and Terminal Services are installed, when a new session becomes active the system copies all the objects from the global \?? directory into the session's local \Devices directory and the *DosDevicesDirectory* field of the DeviceMap structure points at the local directory.
- On Windows XP and Windows Server 2003, the system does not make copies of global objects in the local DosDevices directories. When the object manager sees a reference to \??, it locates the process's local \DosDevices by using the *DosDevicesDirectory* field of the DeviceMap. If the object manager doesn't find the object in that directory, it checks the DeviceMap field of the directory object, and if it's valid it looks for the object in the directory pointed to by the GlobalDosDevicesDirectory field of the DeviceMap structure, which is always \Global??.

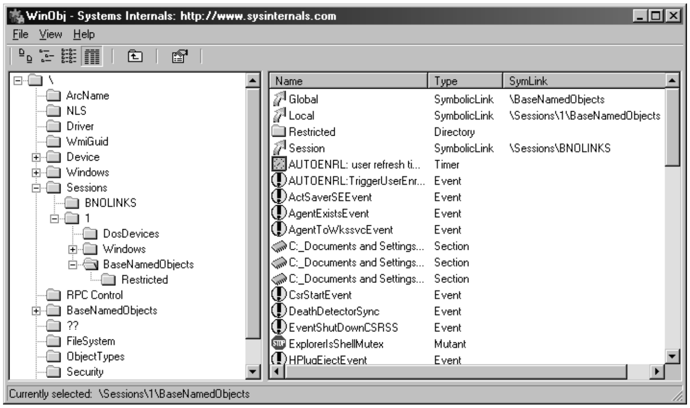
Under certain circumstances, applications that are Terminal Services–aware need to access objects in the console session even if the application is running in a remote session. The application might want to do this to synchronize with instances of itself running in other remote sessions or with the console session. For these cases, the object manager provides the special override "\Global" that an application can prefix to any object name to access the global namespace. For example, an application in session two opening an object named \Global\ApplicationInitialized is directed to \BasedNamedObjects\ApplicationInitialized instead of \Sessions\2\BaseNamedObjects\ApplicationInitialized.

On Windows XP and Windows Server 2003, an application that wants to access an object in the global \DosDevices directory does not need to use the \Global prefix as long as the object doesn't exist in its local \DosDevices directory. This is because the object manager will automatically look in the global directory for the object if it doesn't find it in the local directory. However, an application running on Windows 2000 with Terminal Services must always specify the \Global prefix to access objects in the global \DosDevices directory.

EXPERIMENT: Viewing Namespace Instancing

You can see the object manager instance of the namespace by creating a session other than the console session and then viewing the handle table for a process in that session. On Windows XP Home Edition or on a Windows XP Professional system that is not a member of a domain, disconnect the console session (by clicking Start, clicking Log Off, and choosing Disconnect and Switch User, or by pressing the Windows key + L) and logging in to a new account. If you have a Windows 2000 Server, Advanced Server, or Datacenter Server system, run the Terminal Services client, connect to the server, and log in.

Once you are logged in to the new session, run Winobj.exe from *www.sysinternals.com* and click on the \Sessions directory. You'll see a subdirectory with a numeric name for each active remote session. If you open one of these directories, you'll see subdirectories named \DosDevices, \Windows, and \BaseNamedObjects, which are the local namespace subdirectories of the session. The following screen shot shows a local namespace:



```
\
  ArcName
  NLS
  Driver
  WmiGuid
  Device
  Windows
  Sessions
    BNOLINKS
    1
      DosDevices
      Windows
      BaseNamedObjects
        Restricted
  RPC Control
  BaseNamedObjects
  ??
  FileSystem
  ObjectTypes
  Security
```

Name	Type	SymLink
Global	SymbolicLink	\BaseNamedObjects
Local	SymbolicLink	\Sessions\1\BaseNamedObjects
Restricted	Directory	
Session	SymbolicLink	\Sessions\BNOLINKS
AUTOENRL: user refresh ti...	Timer	
AUTOENRL:TriggerUserEnr...	Event	
ActSaverSEEvent	Event	
AgentExistsEvent	Event	
AgentToWkssvcEvent	Event	
C:_Documents and Settings...	Section	
C:_Documents and Settings...	Section	
C:_Documents and Settings...	Section	
CsrStartEvent	Event	
DeathDetectorSync	Event	
EventShutDownCSRSS	Event	
ExplorerIsShellMutex	Mutant	
HPlugEjectEvent	Event	

Currently selected: \Sessions\1\BaseNamedObjects

Next run Process Explorer and select a process in the new session (such as Explorer.exe), and then view the handle table (by clicking View, Lower Pane View, and then Handles). You should see a handle to \Windows\Windowstations\WinSta0 underneath \Sessions*n*, where *n* is the session id. Objects with global names will appear under \Sessions\n\BaseNamedObjects.

Synchronization

The concept of *mutual exclusion* is a crucial one in operating systems development. It refers to the guarantee that one, and only one, thread can access a particular resource at a time. Mutual exclusion is necessary when a resource doesn't lend itself to shared access or when sharing would result in an unpredictable outcome. For example, if two threads copy a file to a printer port at the same time, their output could be interspersed. Similarly, if one thread reads a memory location while another one writes to it, the first thread will receive unpredictable data. In general, writable resources can't be shared without restrictions, whereas resources that aren't subject to modification can be shared. Figure 3-23 illustrates what happens when two threads running on different processors both write data to a circular queue.

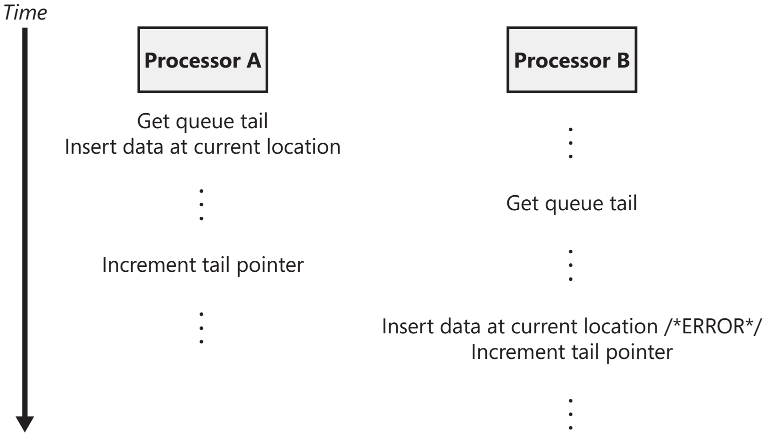


Figure 3-23 Incorrect sharing of memory

Because the second thread got the value of the queue tail pointer before the first thread had finished updating it, the second thread inserted its data into the same location that the first thread had used, overwriting data and leaving one queue location empty. Even though this figure illustrates what could happen on a multiprocessor system, the same error could occur on a single-processor system if the operating system were to perform a context switch to the second thread before the first thread updated the queue tail pointer.

Sections of code that access a nonshareable resource are called *critical* sections. To ensure correct code, only one thread at a time can execute in a critical section. While one thread is writing to a file, updating a database, or modifying a shared variable, no other thread can be allowed to access the same resource. The pseudocode shown in Figure 3-23 is a critical section that incorrectly accesses a shared data structure without mutual exclusion.

The issue of mutual exclusion, although important for all operating systems, is especially important (and intricate) for a *tightly coupled, symmetric multiprocessing* (SMP) operating system such as Windows, in which the same system code runs simultaneously on more than one processor, sharing certain data structures stored in global memory. In Windows, it is the kernel's job to provide mechanisms that system code can use to prevent two threads from modifying the same structure at the same time. The kernel provides mutual-exclusion primitives that it and the rest of the executive use to synchronize their access to global data structures.

Because the scheduler synchronizes access to its data structures at DPC/Dispatch level IRQL, the kernel and executive cannot rely on synchronization mechanisms that would result in a page fault or reschedule operation to synchronize access to data structures when the IRQL is DPC/Dispatch level or higher (levels known as an *elevated* or *high* IRQL). In the following sections, you'll find out how the kernel and executive uses mutual exclusion to protect its global data structures when the IRQL is high and what mutual-exclusion and synchronization mechanisms the kernel and executive use when the IRQL is *low* (below DPC/Dispatch level).

High-IRQL Synchronization

At various stages during its execution, the kernel must guarantee that one, and only one, processor at a time is executing within a critical section. Kernel critical sections are the code segments that modify a global data structure such as the kernel's dispatcher database or its DPC queue. The operating system can't function correctly unless the kernel can guarantee that threads access these data structures in a mutually exclusive manner.

The biggest area of concern is interrupts. For example, the kernel might be updating a global data structure when an interrupt occurs whose interrupt-handling routine also modifies the structure. Simple single-processor operating systems sometimes prevent such a scenario by disabling all interrupts each time they access global data, but the Windows kernel has a more sophisticated solution. Before using a global resource, the kernel temporarily masks those interrupts whose interrupt handlers also use the resource. It does so by raising the processor's IRQL to the highest level used by any potential interrupt source that accesses the global data. For example, an interrupt at DPC/dispatch level causes the dispatcher, which uses the dispatcher database, to run. Therefore, any other part of the kernel that uses the dispatcher database raises the IRQL to DPC/dispatch level, masking DPC/dispatch-level interrupts before using the dispatcher database.

This strategy is fine for a single-processor system, but it's inadequate for a multiprocessor configuration. Raising the IRQL on one processor doesn't prevent an interrupt from occurring on another processor. The kernel also needs to guarantee mutually exclusive access across several processors.

Interlocked Operations

The simplest form of synchronization mechanisms rely on hardware support for multiprocessor-safe manipulating integer values and for performing comparisons. They include functions such as *InterlockedIncrement*, *InterlockedDecrement*, *InterlockedExchange*, and *InterlockedCompareExchange*. The *InterlockedDecrement* function, for example, uses the x86 *lock* instruction prefix (for example, *lock xadd*) to lock the multiprocessor bus during the subtraction operation so that another processor that's also modifying the memory location being decremented won't be able to modify between the decrement's read of the original value and write of the decremented value. This form of basic synchronization is used by the kernel and drivers.

Spinlocks

The mechanism the kernel uses to achieve multiprocessor mutual exclusion is called a *spinlock*. A spinlock is a locking primitive associated with a global data structure, such as the DPC queue shown in Figure 3-24.

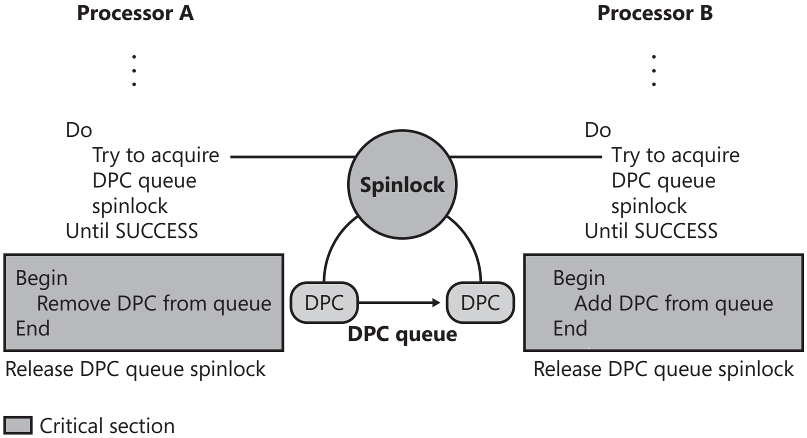


Figure 3-24 Using a spinlock

Before entering either critical section shown in the figure, the kernel must acquire the spinlock associated with the protected DPC queue. If the spinlock isn't free, the kernel keeps trying to acquire the lock until it succeeds. The spinlock gets its name from the fact that the kernel (and thus, the processor) is held in limbo, "spinning," until it gets the lock.

Spinlocks, like the data structures they protect, reside in global memory. The code to acquire and release a spinlock is written in assembly language for speed and to exploit whatever locking mechanism the underlying processor architecture provides. On many architectures, spinlocks are implemented with a hardware-supported test-and-set operation, which tests the value of a lock variable and acquires the lock in one atomic instruction. Testing and acquiring the lock in one instruction prevents a second thread from grabbing the lock between the time when the first thread tests the variable and the time when it acquires the lock.

All kernel-mode spinlocks in Windows have an associated IRQL that is always at DPC/dispatch level or higher. Thus, when a thread is trying to acquire a spinlock, all other activity at the spinlock's IRQL or lower ceases on that processor. Because thread dispatching happens at DPC/dispatch level, a thread that holds a spinlock is never preempted because the IRQL masks the dispatching mechanisms. This masking allows code executing a critical section protected by a spinlock to continue executing so that it will release the lock quickly. The kernel uses spinlocks with great care, minimizing the number of instructions it executes while it holds a spinlock.

> **Note** Because the IRQL is an effective synchronization mechanism on uniprocessors, the spinlock acquisition and release functions of uniprocessor HALs don't implement spinlocks—they simply raise and lower the IRQL.

The kernel makes spinlocks available to other parts of the executive through a set of kernel functions, including *KeAcquireSpinlock* and *KeReleaseSpinlock*. Device drivers, for example, require spinlocks in order to guarantee that device registers and other global data structures are accessed by only one part of a device driver (and from only one processor) at a time. Spinlocks are not for use by user programs–user programs should use the objects described in the next section.

Kernel spinlocks carry with them restrictions for code that uses them. Because spinlocks always have an IRQL of DPC/dispatch level or higher, as explained earlier, code holding a spinlock will crash the system if it attempts to make the scheduler perform a dispatch operation or if it causes a page fault.

Queued Spinlocks

A special type of spinlock called a *queued spinlock* is used in some circumstances instead of a standard spinlock. A queued spinlock is a form of spinlock that scales better on multiprocessors than a standard spinlock. In general, Windows will use only standard spinlocks when it expects there to be low contention for the lock.

A queued spinlock work like this: When a processor wants to acquire a queued spinlock that is currently held, it places its identifier in a queue associated with the spinlock. When the processor that's holding the spinlock releases it, it hands the lock over to the first processor identified in the queue. In the meantime, a processor waiting for a busy spinlock checks the status not of the spinlock itself but of a per-processor flag that the processor ahead of it in the queue sets to indicate that the waiting processor's turn has arrived.

The fact that queued spinlocks result in spinning on per-processor flags rather than global spinlocks has two effects. The first is that the multiprocessor's bus isn't as heavily trafficked by interprocessor synchronization. The second is that instead of a random processor in a waiting group acquiring a spinlock, the queued spinlock enforces first-in, first-out (FIFO) ordering to the lock. FIFO ordering means more consistent performance across processors accessing the same locks.

Windows defines a number of global queued spinlocks by storing pointers to them in an array contained in each processor's *processer control region* (PCR). A global spinlock can be acquired by calling *KeAcquireQueuedSpinlock* with the index into the PCR array at which the pointer to the spinlock is stored. The number of global spinlocks has grown in each release of the operating system, and the table of index definitions for them is published in the DDK header file Ntddk.h.

EXPERIMENT: Viewing Global Queued Spinlocks

You can view the state of the global queued spinlocks (the ones pointed to by the queued spinlock array in each processor's PCR) by using the *!qlock* kernel debugger command. This command is meaningful only on a multiprocessor system because uniprocessor HALs don't implement spinlocks. In the following example, taken from a Windows 2000 system, the dispatcher database queued spinlock is held by processor 1, and the other queued spinlocks are not acquired. (The dispatcher database is described in Chapter 6.)

```
kd> !qlocks
Key: O = Owner, 1-n = Wait order, blank = not owned/waiting, C = Corrupt

                       Processor Number
    Lock Name      0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15

KE - Dispatcher       O
KE - Context Swap
MM - PFN
MM - System Space
CC - Vacb CC – Master
```

Instack Queued Spinlocks

In addition to using static queued spinlocks that are globally defined, Windows XP and Windows Server 2003 kernels support dynamically allocated queued spinlocks with the *KeAcquireInStackQueuedSpinlock* and *KeReleaseInStackQueuedSpinlock* functions. Several components—including the cache manager, executive pool manager, and NTFS—take advantage of these types of locks, and the functions are documented in the DDK for use by third-party driver writers.

KeAcquireInStackQueuedSpinlock takes a pointer to a spinlock data structure and a spin lock queue handle. The spin lock handle is actually a data structure in which the kernel stores information about the lock's status, including the lock's ownership and the queue of processors that might be waiting for the lock to become available.

Executive Interlocked Operations

The kernel supplies a number of simple synchronization functions constructed on spinlocks for more advanced operations, such as adding and removing entries from singly and doubly linked lists. Examples include *ExInterlockedPopEntryList* and *ExInterlockedPushEntryList* for singly linked lists, and *ExInterlockedInsertHeadList* and *ExInterlockedRemoveHeadList* for doubly linked lists. All these functions require a standard spinlock as a parameter and are used throughout the kernel and device drivers.

Low-IRQL Synchronization

Executive software outside the kernel also needs to synchronize access to global data structures in a multiprocessor environment. For example, the memory manager has only one page frame database, which it accesses as a global data structure, and device drivers need to ensure that they can gain exclusive access to their devices. By calling kernel functions, the executive can create a spinlock, acquire it, and release it.

Spinlocks only partially fill the executive's needs for synchronization mechanisms, however. Because waiting for a spinlock literally stalls a processor, spinlocks can be used only under the following strictly limited circumstances:

- The protected resource must be accessed quickly and without complicated interactions with other code.
- The critical section code can't be paged out of memory, can't make references to pageable data, can't call external procedures (including system services), and can't generate interrupts or exceptions.

These restrictions are confining and can't be met under all circumstances. Furthermore, the executive needs to perform other types of synchronization in addition to mutual exclusion, and it must also provide synchronization mechanisms to user mode.

There are several additional synchronization mechanisms for use when spinlocks are not suitable:

- Kernel Dispatcher Objects
- Fast Mutexes and Guarded Mutexes
- Push Locks
- Executive Resources

Table 3-9 serves as a reference that compares and contrasts the capabilities of these mechanisms and their interaction with kernel-mode APC delivery.

Table 3-9 Kernel Synchronization Mechanisms

	Exposed for Use by Device Drivers	Disables Normal Kernel-Mode APCs	Disables Special Kernel-Mode APCs	Supports Recursive Acquisition	Supports Shared and Exclusive Acquisition
Kernel Dispatcher Mutexes	Yes	Yes	No	Yes	No
Kernel Dispatcher Semaphores	Yes	No	No	No	No
Fast Mutexes	Yes	Yes	Yes	No	No
Guarded Mutexes	No	Yes	Yes	No	No
Push Locks	No	No	No	No	Yes
Executive Resources	Yes	Yes	No	Yes	Yes

Kernel Dispatcher Objects

The kernel furnishes additional synchronization mechanisms to the executive in the form of kernel objects, known collectively as *dispatcher objects*. The user-visible synchronization objects acquire their synchronization capabilities from these kernel dispatcher objects. Each user-visible object that supports synchronization encapsulates at least one kernel dispatcher object. The executive's synchronization semantics are visible to Windows programmers through the *WaitForSingleObject* and *WaitForMultipleObjects* functions, which the Windows subsystem implements by calling analogous system services the object manager supplies. A thread in a Windows application can synchronize with a Windows process, thread, event, semaphore, mutex, waitable timer, I/O completion port, or file object.

One other type of executive synchronization object worth noting is called *executive resources*. Executive resources provide both exclusive access (like a mutex) as well as shared read access (multiple readers sharing read-only access to a structure). However, they're available only to kernel-mode code and thus aren't accessible from the Windows API. Executive resources are not dispatcher objects but rather data structures allocated directly from nonpaged pool that have their own specialized services to initialize, lock, release, query, and wait for them. The executive resource structure is defined in Ntddk.h, and the executive support routines are documented in the DDK reference documentation.

The remaining subsections describe the implementation details of waiting for dispatcher objects.

Waiting for Dispatcher Objects A thread can synchronize with a dispatcher object by waiting for the object's handle. Doing so causes the kernel to suspend the thread and change its dispatcher state from running to waiting, as shown in Figure 3-25. The kernel removes the thread from the dispatcher ready queue and no longer considers it for execution.

Note Figure 3-25 is a process state transition diagram with focus on the ready, waiting, and running states (the states related to waiting for objects). The other states are described in Chapter 6.

At any given moment, a synchronization object is in one of two states: either the *signaled state* or the *nonsignaled state*. A thread can't resume its execution until the kernel changes its dispatcher state from waiting to ready. This change occurs when the dispatcher object whose handle the thread is waiting for also undergoes a state change, from the nonsignaled state to the signaled state (when a thread sets an event object, for example). To synchronize with an object, a thread calls one of the wait system services the object manager supplies, passing a handle to the object it wants to synchronize with. The thread can wait for one or several objects and can also specify that its wait should be canceled if it hasn't ended within a certain amount of time. Whenever the kernel sets an object to the signaled state, the kernel's *KiWaitTest* function checks to see whether any threads are waiting for the object and not also waiting for other objects to become signaled. If there are, the kernel releases one or more of the threads from their waiting state so that they can continue executing.

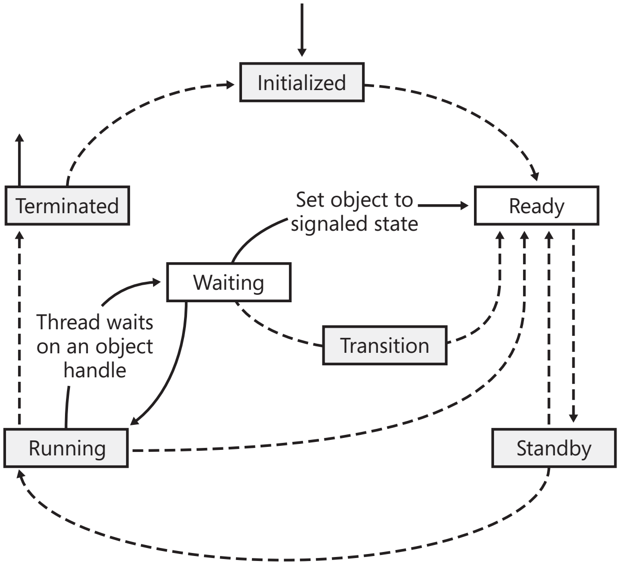

Initialized
Terminated
Set object to signaled state
Ready
Waiting
Thread waits on an object handle
Transition
Running
Standby

Figure 3-25 Waiting for a dispatcher object

The following example of setting an event illustrates how synchronization interacts with thread dispatching:

- A user-mode thread waits for an event object's handle.
- The kernel changes the thread's scheduling state from ready to waiting and then adds the thread to a list of threads waiting for the event.
- Another thread sets the event.
- The kernel marches down the list of threads waiting for the event. If a thread's conditions for waiting are satisfied (see Note below), the kernel changes the thread's state from waiting to ready. If it is a variable-priority thread, the kernel might also boost its execution priority.
- Because a new thread has become ready to execute, the dispatcher reschedules. If it finds a running thread with a priority lower than that of the newly ready thread, it preempts the lower-priority thread and issues a software interrupt to initiate a context switch to the higher-priority thread.
- If no processor can be preempted, the dispatcher places the ready thread in the dispatcher ready queue to be scheduled later.

> **Note** Some threads might be waiting for more than one object, so they continue waiting.

What Signals an Object The signaled state is defined differently for different objects. A thread object is in the nonsignaled state during its lifetime and is set to the signaled state by the kernel when the thread terminates. Similarly, the kernel sets a process object to the signaled state when the process's last thread terminates. In contrast, the timer object, like an alarm, is set to "go off" at a certain time. When its time expires, the kernel sets the timer object to the signaled state.

When choosing a synchronization mechanism, a program must take into account the rules governing the behavior of different synchronization objects. Whether a thread's wait ends when an object is set to the signaled state varies with the type of object the thread is waiting for, as Table 3-10 illustrates.

Table 3-10 Definitions of the Signaled State

Object Type	Set to Signaled State When	Effect on Waiting Threads
Process	Last thread terminates	All released
Thread	Thread terminates	All released
File	I/O operation completes	All released
Debug Object	Debug message is queued to the object	All released
Event (notification type)	Thread sets the event	All released
Event (synchronization type)	Thread sets the event	One thread released; event object reset
Keyed Event	Thread sets event with a key	Thread waiting for key and which is of same process as signaler is released
Semaphore	Semaphore count drops by 1	One thread released
Timer (notification type)	Set time arrives or time interval expires	All released
Timer (synchronization type)	Set time arrives or time interval expires	One thread released
Mutex	Thread releases the mutex	One thread released
File	I/O completes	All threads released
Queue	Item is placed on queue	One thread released

When an object is set to the signaled state, waiting threads are generally released from their wait states immediately. Some of the kernel dispatcher objects and the system events that induce their state changes are shown in Figure 3-26.

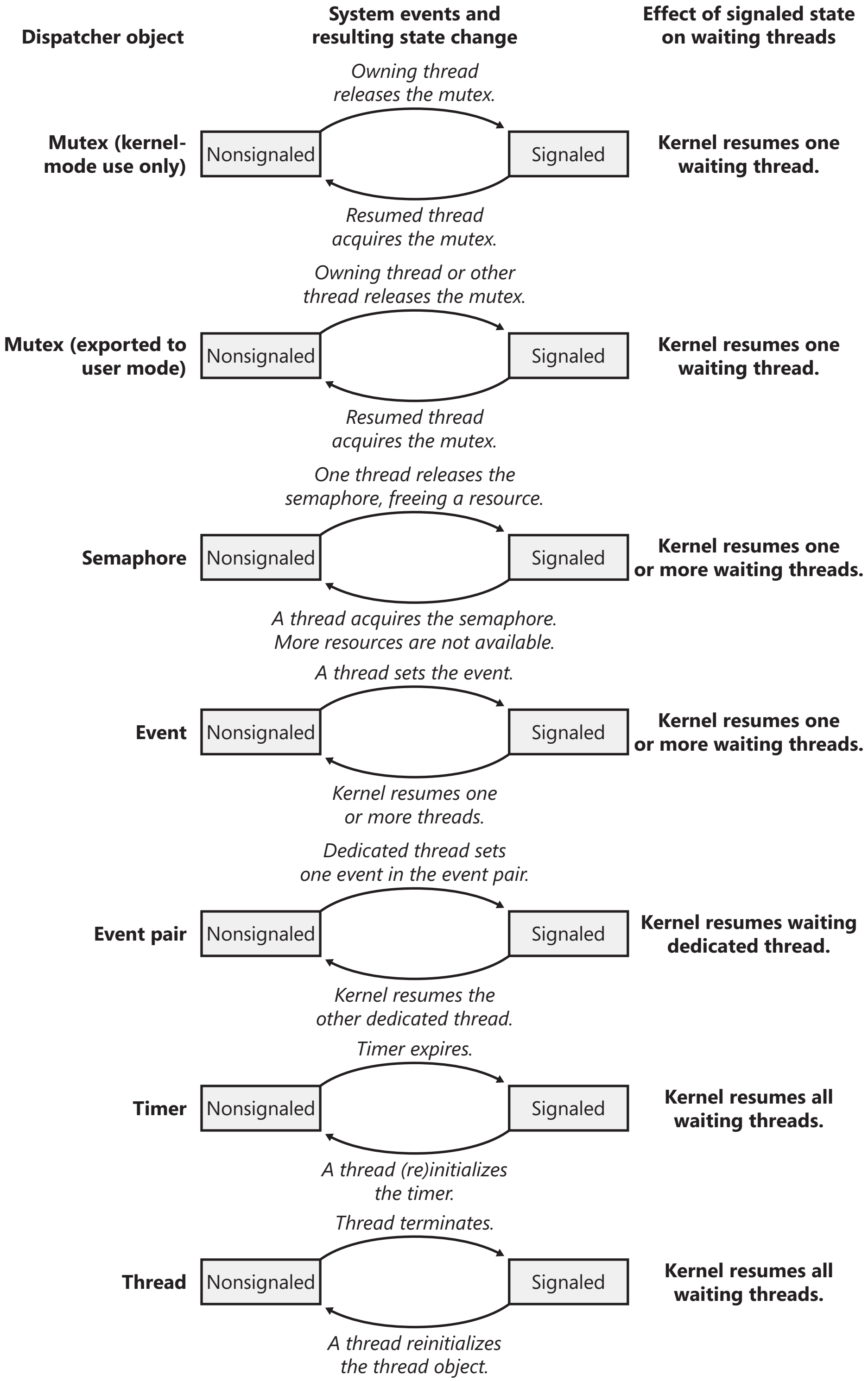

Dispatcher object
System events and resulting state change
Effect of signaled state on waiting threads
Owning thread releases the mutex.
Mutex (kernel-mode use only)
Nonsignaled
Signaled
Kernel resumes one waiting thread.
Resumed thread acquires the mutex.
Owning thread or other thread releases the mutex.
Mutex (exported to user mode)
Nonsignaled
Signaled
Kernel resumes one waiting thread.
Resumed thread acquires the mutex.
One thread releases the semaphore, freeing a resource.
Semaphore
Nonsignaled
Signaled
Kernel resumes one or more waiting threads.
A thread acquires the semaphore. More resources are not available.
A thread sets the event.
Event
Nonsignaled
Signaled
Kernel resumes one or more waiting threads.
Kernel resumes one or more threads.
Dedicated thread sets one event in the event pair.
Event pair
Nonsignaled
Signaled
Kernel resumes waiting dedicated thread.
Kernel resumes the other dedicated thread.
Timer expires.
Timer
Nonsignaled
Signaled
Kernel resumes all waiting threads.
A thread (re)initializes the timer.
Thread terminates.
Thread
Nonsignaled
Signaled
Kernel resumes all waiting threads.
A thread reinitializes the thread object.

Figure 3-26 Selected kernel dispatcher objects

For example, a notification event object (called a manual reset event in the Windows API) is used to announce the occurrence of some event. When the event object is set to the signaled state, all threads waiting for the event are released. The exception is any thread that is waiting for more than one object at a time; such a thread might be required to continue waiting until additional objects reach the signaled state.

In contrast to an event object, a mutex object has ownership associated with it. It is used to gain mutually exclusive access to a resource, and only one thread at a time can hold the mutex. When the mutex object becomes free, the kernel sets it to the signaled state and then selects one waiting thread to execute. The thread selected by the kernel acquires the mutex object, and all other threads continue waiting.

Keyed Events and Critical Sections

A synchronization object new to Windows XP, called a *keyed event*, bears special mention because of the role it plays in helping processes deal with low-memory situations when using *critical sections*. A keyed event, which is not documented, allows a thread to specify a "key" for which it waits, where the thread wakes when another thread of the same process signals the event with the same key.

Windows processes often use critical section functions, *EnterCriticalSection* and *LeaveCriticalSection*, to synchronize thread access to resources private to the process. These functions have advantages over direct use of mutex objects because if there is no contention they do not make a transition to kernel mode. If there is contention, *EnterCriticalSection* dynamically allocates an event object and the thread wanting to acquire the critical section waits for the thread that owns the critical section to signal it in *LeaveCriticalSection*.

EnterCriticalSection uses a global keyed event named *CritSecOutOfMemoryEvent* (in the \Kernel directory of the object manager namespace) when the allocation of the event object for the critical section fails because system memory is low. If *EnterCriticalSection* has to use *CritSecOutOfMemoryEvent* instead of a standard event, a thread waiting for the critical section uses the address of the critical section as the key. This allows the critical section functions to operate properly even when memory is temporarily low.

This brief discussion wasn't meant to enumerate all the reasons and applications for using the various executive objects but rather to list their basic functionality and synchronization behavior. For information on how to put these objects to use in Windows programs, see the Windows reference documentation on synchronization objects or Jeffrey Richter's *Programming Applications for Microsoft Windows.*

Data Structures Two data structures are key to tracking who is waiting for what: *dispatcher headers* and *wait blocks*. Both these structures are publicly defined in the DDK include file Ntddk.h. The definitions are reproduced here for convenience:

```
typedef struct _DISPATCHER_HEADER {
    UCHAR Type;
    UCHAR Absolute;
    UCHAR Size;
    UCHAR Inserted;
    LONG SignalState;
    LIST_ENTRY WaitListHead;
} DISPATCHER_HEADER;

typedef struct _KWAIT_BLOCK {
    LIST_ENTRY WaitListEntry;
    struct _KTHREAD *RESTRICTED_POINTER Thread;
    PVOID Object;
    struct _KWAIT_BLOCK *RESTRICTED_POINTER NextWaitBlock;
    USHORT WaitKey;
    USHORT WaitType;
} KWAIT_BLOCK, *PKWAIT_BLOCK, *RESTRICTED_POINTER PRKWAIT_BLOCK;
```

The dispatcher header contains the object type, signaled state, and a list of the threads waiting for that object. The wait block represents a thread waiting for an object. Each thread that is in a wait state has a list of the wait blocks that represent the objects the thread is waiting for. Each dispatcher object has a list of the wait blocks that represent which threads are waiting for the object. This list is kept so that when a dispatcher object is signaled, the kernel can quickly determine who is waiting for that object. The wait block has a pointer to the object being waited for, a pointer to the thread waiting for the object, and a pointer to the next wait block (if the thread is waiting for more than one object). It also records the type of wait (any or all) as well as the position of that entry in the array of handles passed by the thread on the *WaitForMultipleObjects* call (position zero if the thread was waiting for only one object).

Figure 3-27 shows the relationship of dispatcher objects to wait blocks to threads. In this example, thread 1 is waiting for object B, and thread 2 is waiting for objects A and B. If object A is signaled, the kernel will see that because thread 2 is also waiting for another object, thread 2 can't be readied for execution. On the other hand, if object B is signaled, the kernel can ready thread 1 for execution right away because it isn't waiting for any other objects.

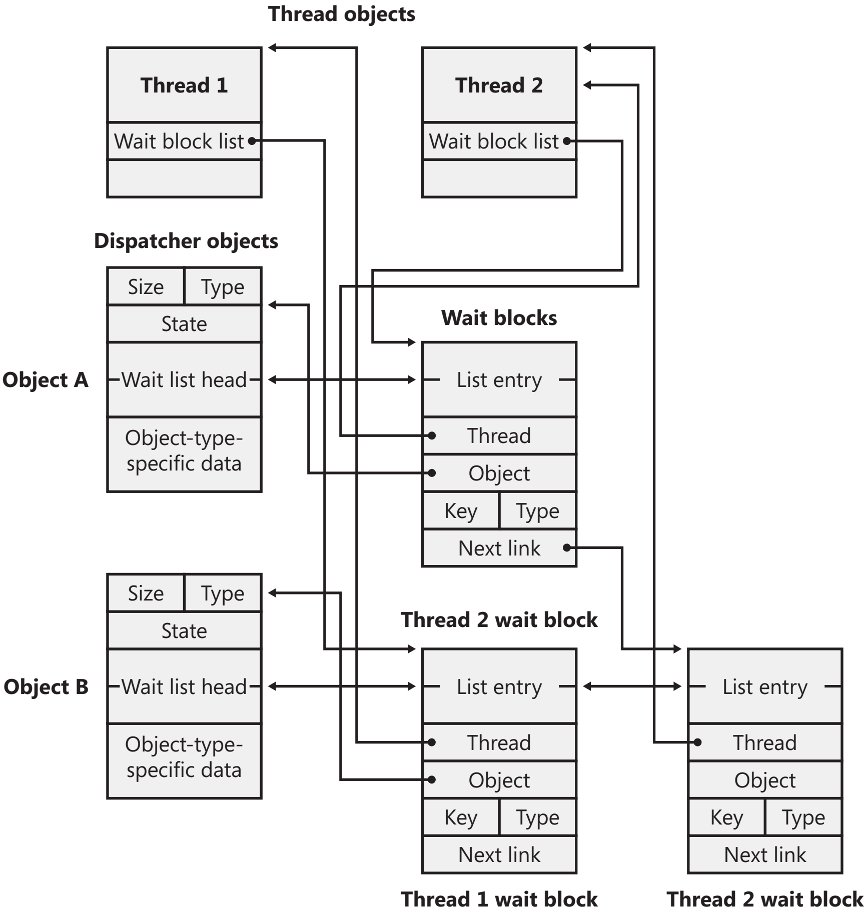


Figure 3-27 Wait data structures

EXPERIMENT: Looking at Wait Queues

Although many process viewer utilities indicate whether a thread is in a wait state (and if so, they also indicate what kind of wait), you can see the list of objects a thread is waiting for only with the kernel debugger *!thread* command. For example, the following excerpt from the output of a *!process* command shows that the thread is waiting for an event object:

```
kd> !process
§
        THREAD 8a12a328  Cid 0bb8.0d50  Teb: 7ffdd000 Win32Thread: e7c9aeb0 WAIT
: (WrUserRequest) UserMode Non-Alertable
            8a21bf58  SynchronizationEvent
```

You can use the *dt* command to interpret the dispatcher header of the object like this:

```
kd> dt nt!_dispatcher_header 8a21bf58
nt!_DISPATCHER_HEADER
   +0x000 Type             : 0x1 ''
   +0x001 Absolute         : 0 ''
   +0x002 Size             : 0x4 ''
   +0x003 Inserted         : 0 ''
   +0x004 SignalState      : 0
   +0x008 WaitListHead     : _LIST_ENTRY [ 0x8a12a398 - 0x8a12a398 ]
```

From this, we can ascertain that no other threads are waiting for this event object because the wait list head forward and backward pointers point to the same location (a single wait block). Dumping the wait block (at address 0x8a12a398) yields the following:

```
kd> dt nt!_kwait_block 0x8a12a398
nt!_KWAIT_BLOCK
   +0x000 WaitListEntry    : _LIST_ENTRY [ 0x8a21bf60 - 0x8a21bf60 ]
   +0x008 Thread           : 0x8a12a328
   +0x00c Object           : 0x8a21bf58
   +0x010 NextWaitBlock    : 0x8a12a398
   +0x014 WaitKey          : 0
   +0x016 WaitType         : 1
```

If the wait list had more than one entry, you could execute the same command on the second pointer value in the *WaitListEntry* field of each wait block (by executing *!thread* on the thread pointer in the wait block) to traverse the list and see what other threads are waiting for the object.

Fast Mutexes and Guarded Mutexes

Fast mutexes, which are also known as executive mutexes, usually offer better performance than mutex objects because, although they are built on dispatcher event objects, they avoid waiting for the event object (and therefore the spinlocks on which an event object is based) if there's no contention for the fast mutex. This gives the fast mutex especially good performance in a multiprocessor environment. Fast mutexes are used widely throughout the kernel and device drivers.

However, fast mutexes are suitable only when normal kernel-mode APC (described earlier in this chapter) delivery can be disabled. The executive defines two functions for acquiring them: *ExAcquireFastMutex* and *ExAcquireFastMutexUnsafe*. The former function blocks all APC delivery by raising the IRQL of the processor to APC_LEVEL and the latter expects to be called with normal kernel-mode APC delivery disabled, which can be done by raising the IRQL to APC level or by calling *KeEnterCriticalRegion*. Another limitation of fast mutexes is that they can't be acquired recursively like mutex objects can.

Guarded mutexes are new to Windows Server 2003 and are essentially the same as fast mutexes (although they use a different synchronization object, the KGATE, internally). They are acquired with the *KeAcquireGuardedMutex* function, but instead of disabling APCs by calling *KeEnterCriticalRegion*, which disables only normal kernel-mode APCs, it disables all kernel-mode APC delivery by calling *KeEnterGuardedRegion*. They are not exposed for use outside of the kernel and are used primarily by the memory manager, which uses them to protect global operations such as creating paging files, deleting certain types of shared memory sections, and performing paged pool expansion. (See Chapter 7 for more information on the memory manager.)

Executive Resources

Executive resources are a synchronization mechanism that supports shared and exclusive access, and like fast mutexes, require that normal kernel-mode APC delivery be disabled before they are acquired. They are also built on dispatcher objects that are only used when there is contention. Executive resources are used throughout the system, especially in file-system drivers.

Threads waiting to acquire a resource for shared access wait for a semaphore associated with the resource, and threads waiting to acquire a resource for exclusive access wait for an event. A semaphore with unlimited count is used for shared waiters because they can all be woken and granted access to the resource when an exclusive holder releases the resource simply by signaling the semaphore. When a thread waits for exclusive access of a resource that is currently owned, it waits on a synchronization event object because only one of the waiters will wake when the event is signaled.

Because of the flexibility that shared and exclusive access offers, there are a number of functions for acquiring resources: *ExAcquireResourceSharedLite*, *ExAcquireResourceExclusiveLite*, *ExAcquireSharedStarveExclusive*, *ExAcquireWaitForExclusive*, and *ExTryToAcquireResourceExclusiveLite*. These functions are documented in the DDK.

EXPERIMENT: Listing Acquired Executive Resources

The kernel debugger *!locks* command searches paged pool for executive resource objects and dumps their state. By default, the command lists only executive resources that are currently owned, but the *–d* option will list all executive resources. Here is partial output of the command:

```
lkd> !locks
**** DUMP OF ALL RESOURCE OBJECTS ****
KD: Scanning for held locks.

Resource @ nt!MmSystemWsLock (0x805439a0)    Exclusively owned
    Contention Count = 123
     Threads: 89b36020-01<*>
KD: Scanning for held locks..........................................................
```

```
.......................................................................................
..............................

Resource @ 0x89da1a68    Shared 1 owning threads
     Threads: 8a4cb533-01<*> *** Actual Thread 8a4cb530
```

Note that the contention count, which is extracted from the resource structure, records the number of times threads have tried to acquire the resource and had to wait because it was already owned.

You can examine the details of a specific resource object, including the thread that owns the resource and any threads that are waiting for the resource, by specifying the *–v* switch and the address of the resource:

```
lkd> !locks -v 0x805439a0

Resource @ nt!MmSystemWsLock (0x805439a0)    Exclusively owned
    Contention Count = 123
     Threads: 89b36020-01<*>

     THREAD 89b36020  Cid 0e98.0bd4  Teb: 7ffd9000 Win32Thread: e2bcc538 RUNNING on pr
ocessor 0
     Not impersonating
     DeviceMap                 e1df7d18
     Owning Process            8999d020
     Wait Start TickCount      492582         Elapsed Ticks: 15
     Context Switch Count      532                 LargeStack
     UserTime                  00:00:01.0462
     KernelTime                00:00:00.0320
     Start Address 0x77e7d342
     Win32 Start Address 0x0101f1d0
     Stack Init a9d20000 Current a9d1fd44 Base a9d20000 Limit a9d1d000 Call 0
     Priority 11 BasePriority 8 PriorityDecrement 2 DecrementCount 16
Unable to get context for thread running on processor 0, HRESULT 0x80004001
```

Push Locks

Push locks, which were introduced in Windows XP, are another optimized synchronization mechanism built on the event object (and in Windows Server 2003 they are built on the internal KGATE synchronization object), and like fast mutexes, they wait for an event object only when there's contention on the lock. They offer advantages over the fast mutex in that they can be acquired in shared or exclusive mode. They are not documented or exported by the kernel and are therefore reserved for use by the operating system.

There are two types of push locks: normal and cache aware. Normal push locks require only the size of a pointer in storage (4 bytes on 32-bit systems and 8 bytes on 64-bit systems). When a thread acquires a normal push lock, the push lock code marks the push lock as owned if it is not currently owned. If the push lock is owned exclusively or the thread wants to acquire the thread exclusively and the push lock is owned on a shared basis, the thread allocates a wait block on the thread's stack, initializes an event object in the wait block, and adds the wait block to the wait list associated with the push lock. When a thread releases a push lock, the thread wakes a waiter, if any are present, by signaling the event in the waiter's wait block.

A cache-aware push lock layers on the basic push lock by allocating a push lock for each processor in the system and associating it with the cache-aware push lock. When a thread wants to acquire a cache-aware push lock for shared access, it simply acquires the push lock allocated for its current processor in shared mode; to acquire a cache-aware push lock exclusively, it acquires the push lock for each processor in exclusive mode.

Areas where push locks are used include the object manager, where they protect global object manager data structures and object security descriptors, and the memory manager, where they protect AWE data structures.

Deadlock Detection with Driver Verifier

A deadlock is a synchronization issue resulting from two threads or processors holding resources that the other wants and neither will yield what it has. This situation might result in system or process hangs. Driver Verifier, described in Chapter 7 and Chapter 9, has an option to check for deadlocks involving spinlocks, fast mutexes, and mutexes. For information on when to enable Driver Verifier to help resolve system hangs, see Chapter 14.

System Worker Threads

During system initialization, Windows creates several threads in the System process, called *system worker threads*, that exist solely to perform work on behalf of other threads. In many cases, threads executing at DPC/dispatch level need to execute functions that can be performed only at a lower IRQL. For example, a DPC routine, which executes in an arbitrary thread context (because DPC execution can usurp any thread in the system) at DPC/dispatch level IRQL, might need to access paged pool or wait for a dispatcher object used to synchronize execution with an application thread. Because a DPC routine can't lower the IRQL, it must pass such processing to a thread that executes at an IRQL below DPC/dispatch level.

Some device drivers and executive components create their own threads dedicated to processing work at passive level; however, most use system worker threads instead, which avoids the unnecessary scheduling and memory overhead associated with having additional threads in the system. A device driver or an executive component requests a system worker thread's services by calling the executive functions *ExQueueWorkItem* or *IoQueueWorkItem*. These functions place a *work item* on a queue dispatcher object where the threads look for work. (Queue dispatcher objects are described in more detail in the section “I/O Completion Ports” in Chapter 9.) Work items include a pointer to a routine and a parameter that the thread passes to the routine when it processes the work item. The routine is implemented by the device driver or executive component that requires passive-level execution.

For example, a DPC routine that must wait for a dispatcher object can initialize a work item that points to the routine in the driver that waits for the dispatcher object, and perhaps points to a pointer to the object. At some stage, a system worker thread will remove the work item from its queue and execute the driver’s routine. When the driver’s routine finishes, the system worker thread checks to see whether there are more work items to process. If there aren’t any more, the system worker thread blocks until a work item is placed on the queue. The DPC routine might or might not have finished executing when the system worker thread processes its work item. (On a uniprocessor system, a DPC routine always finishes executing before its work item is processed because thread scheduling doesn’t take place when the IRQL is at DPC/dispatch level.)

There are three types of system worker threads:

- *Delayed worker threads* execute at priority 12, process work items that aren’t considered time-critical, and can have their stack paged out to a paging file while they wait for work items.
- *Critical worker threads* execute at priority 13, process time-critical work items, and on Windows Server systems, have their stacks present in physical memory at all times.
- A single *hypercritical worker thread* executes at priority 15 and also keeps its stack in memory. The process manager uses the hypercritical work item to execute the thread “reaper” function that frees terminated threads.

The number of delayed and critical worker threads created by the executive’s *ExpWorkerInitialization* function, which is called early in the boot process, depends on the amount of memory present on the system and whether the system is a server. Table 3-11 shows the initial number of threads created on different system configurations. You can specify that *ExpInitializeWorker* create up to 16 additional delayed and 16 additional critical worker threads with the AdditionalDelayedWorkerThreads and AdditionalCriticalWorkerThreads values under the registry key HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Executive.

Table 3-11 Initial Number of System Worker Threads

	Windows 2000	Windows 2000 Server	Windows XP and Windows Server 2003
Delayed	3	3	7
Critical	5	10	5
Hypercritical	1	1	1

The executive tries to match the number of critical worker threads with changing workloads as the system executes. Once every second, the executive function *ExpWorkerThreadBalanceManager* determines whether it should create a new critical worker thread. The critical worker threads that are created by *ExpWorkerThreadBalanceManager* are called *dynamic worker threads*, and all the following conditions must be satisfied before such a thread is created:

- Work items exist in the critical work queue.
- The number of inactive critical worker threads (ones that are either blocked waiting for work items or that have blocked on dispatcher objects while executing a work routine) must be less than the number of processors on the system.
- There are fewer than 16 dynamic worker threads.

Dynamic worker threads exit after 10 minutes of inactivity. Thus, when the workload dictates, the executive can create up to 16 dynamic worker threads.

EXPERIMENT: Listing System Worker Threads

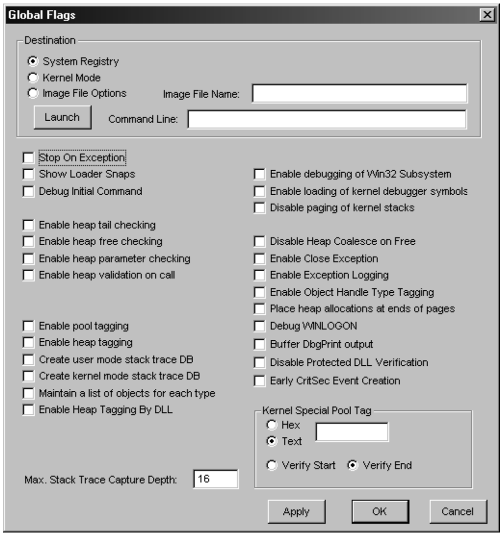
You can use the *!exqueue* kernel debugger command to see a listing of system worker threads classified by their type:

```
kd> !exqueue
Dumping ExWorkerQueue: 8046A5C0

**** Critical WorkQueue( current = 0 maximum = 1 )
THREAD 818a2d40  Cid 8.c   Teb: 00000000  Win32Thread: 00000000 WAIT
THREAD 818a2ac0  Cid 8.10  Teb: 00000000  Win32Thread: 00000000 WAIT
THREAD 818a2840  Cid 8.14  Teb: 00000000  Win32Thread: 00000000 WAIT
THREAD 818a25c0  Cid 8.18  Teb: 00000000  Win32Thread: 00000000 WAIT
THREAD 818a2340  Cid 8.1c  Teb: 00000000  Win32Thread: 00000000 WAIT

**** Delayed WorkQueue( current = 0 maximum = 1 )
THREAD 818a20c0  Cid 8.20  Teb: 00000000  Win32Thread: 00000000 WAIT
THREAD 818a1020  Cid 8.24  Teb: 00000000  Win32Thread: 00000000 WAIT
THREAD 818a1da0  Cid 8.28  Teb: 00000000  Win32Thread: 00000000 WAIT

**** HyperCritical WorkQueue( current = 0 maximum = 1 )
THREAD 818a1b20  Cid 8.2c  Teb: 00000000  Win32Thread: 00000000 WAIT
```

Windows Global Flags

Windows has a set of flags stored in a systemwide global variable named *NtGlobalFlag* that enable various internal debugging, tracing, and validation support in the operating system. The system variable *NtGlobalFlag* is initialized from the registry key HKLM\SYSTEM\CurrentControlSet\Control\Session Manager in the value GlobalFlag at system boot time. By default, this registry value is 0, so it's likely that on your systems, you're not using any global flags. In addition, each image has a set of global flags that also turn on internal tracing and validation code (although the bit layout of these flags is entirely different than the systemwide global flags). These flags aren't documented or supported for customer use, but they can be useful tools for exploring the internal operation of Windows.

Fortunately, the Platform SDK and the debugging tools contain a utility named Gflags.exe that allows you to view and change the system global flags (either in the registry or in the running system) as well as image global flags. Gflags has both a command-line and a GUI interface. To see the command-line flags, type **gflags /?**. If you run the utility without any switches, the dialog box shown in Figure 3-28 is displayed.



Destination
● System Registry
○ Kernel Mode
○ Image File Options
Image File Name:
Command Line:

☐ Stop On Exception
☐ Show Loader Snaps
☐ Debug Initial Command
☐ Enable heap tail checking
☐ Enable heap free checking
☐ Enable heap parameter checking
☐ Enable heap validation on call
☐ Enable pool tagging
☐ Enable heap tagging
☐ Create user mode stack trace DB
☐ Create kernel mode stack trace DB
☐ Maintain a list of objects for each type
☐ Enable Heap Tagging By DLL

Max. Stack Trace Capture Depth: 16

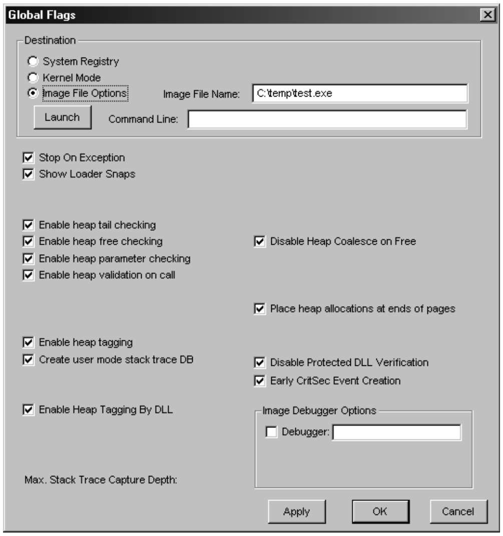
☐ Enable debugging of Win32 Subsystem
☐ Enable loading of kernel debugger symbols
☐ Disable paging of kernel stacks
☐ Disable Heap Coalesce on Free
☐ Enable Close Exception
☐ Enable Exception Logging
☐ Enable Object Handle Type Tagging
☐ Place heap allocations at ends of pages
☐ Debug WINLOGON
☐ Buffer DbgPrint output
☐ Disable Protected DLL Verification
☐ Early CritSec Event Creation

Kernel Special Pool Tag
○ Hex
● Text
○ Verify Start ● Verify End

Figure 3-28 Setting system debugging options with Gflags

You can toggle between the settings in the registry (by clicking System Registry) and the current value of the variable in system memory (by clicking Kernel Mode). You must click the Apply button to make the changes. (You'll exit if you click the OK button.) Although you can change flag settings on a running system, most flags require a reboot to take effect, and there's no documentation on which flags do and which don't require rebooting. So when in doubt, reboot after changing a global flag.

The Image File Options choice requires that you fill in the filename of a valid executable image. This option is used to change a set of global flags that apply to an individual image (rather than to the whole system). In Figure 3-29, notice that the flags are different than the operating system ones shown in Figure 3-28.



○ System Registry
○ Kernel Mode
● Image File Options Image File Name: C:\temp\test.exe
Command Line:

☑ Stop On Exception
☑ Show Loader Snaps

☑ Enable heap tail checking
☑ Enable heap free checking
☑ Enable heap parameter checking
☑ Enable heap validation on call
☑ Disable Heap Coalesce on Free
☑ Place heap allocations at ends of pages

☑ Enable heap tagging
☑ Create user mode stack trace DB
☑ Disable Protected DLL Verification
☑ Early CritSec Event Creation

☑ Enable Heap Tagging By DLL
□ Debugger:

Max. Stack Trace Capture Depth:

Figure 3-29 Setting image global flags with Gflags

EXPERIMENT: Enabling Image Loader Tracing and Viewing *NtGlobalFlag*

To see an example of the detailed tracing information you can obtain by setting global flags, try running Gflags on a system booted with the kernel debugger that is connected to a host system running Kd or Windbg.

As an example, try enabling the Show Loader Snaps flag. To do this, choose Kernel Mode, select the Show Loader Snaps check box, and click the Apply button. Then run an image on this machine, and in the kernel debugger you'll see volumes of output like the following:

```
LDR: PID: 0xb8 started - 'notepad'
LDR: NEW PROCESS
     Image Path: C:\Windows\system32\notepad.exe (notepad.exe)
     Current Directory: C:\ddk\bin
     Search Path: C:\Windows\System32;C:\Windows\system;C:\Windows
LDR: notepad.exe bound to comdlg32.dll
LDR: ntdll.dll used by comdlg32.dll
LDR: Snapping imports for comdlg32.dll from ntdll.dll
    §
LDR: KERNEL32.dll loaded. - Calling init routine at 77f01000
LDR: RPCRT4.dll loaded. - Calling init routine at 77e1b6d5
LDR: ADVAPI32.dll loaded. - Calling init routine at 77dc1000
LDR: USER32.dll loaded. - Calling init routine at 77e78037
```

You can use the *!gflags* and *!gflag* kernel debugger commands to view the state of the *NtGlobalFlag* kernel variable. The *!gflags* command lists all the flags, indicating which ones are enabled, whereas *!gflag* reports only the flags that are enabled.

```
kd> !gflags

NT!NtGlobalFlag 0x4400

  STOP_ON_EXCEPTION               SHOW_LDR_SNAPS
  DEBUG_INITIAL_COMMAND           STOP_ON_HUNG_GUI
  HEAP_ENABLE_TAIL_CHECK          HEAP_ENABLE_FREE_CHECK
  HEAP_VALIDATE_PARAMETERS        HEAP_VALIDATE_ALL
 *POOL_ENABLE_TAGGING             HEAP_ENABLE_TAGGING
  USER_STACK_TRACE_DB             KERNEL_STACK_TRACE_DB
 *MAINTAIN_OBJECT_TYPELIST        HEAP_ENABLE_TAG_BY_DLL
  ENABLE_CSRDEBUG                 ENABLE_KDEBUG_SYMBOL_LOAD
  DISABLE_PAGE_KERNEL_STACKS      HEAP_DISABLE_COALESCING
  ENABLE_CLOSE_EXCEPTIONS         ENABLE_EXCEPTION_LOGGING
  ENABLE_HANDLE_TYPE_TAGGING      HEAP_PAGE_ALLOCS
  DEBUG_INITIAL_COMMAND_EX        DISABLE_DBGPRINT

kd> !gflag
NtGlobalFlag at 8046a164 Current NtGlobalFlag contents: 0x00004400
    ptg - Enable pool tagging
    otl - Maintain a list of objects for each type
```

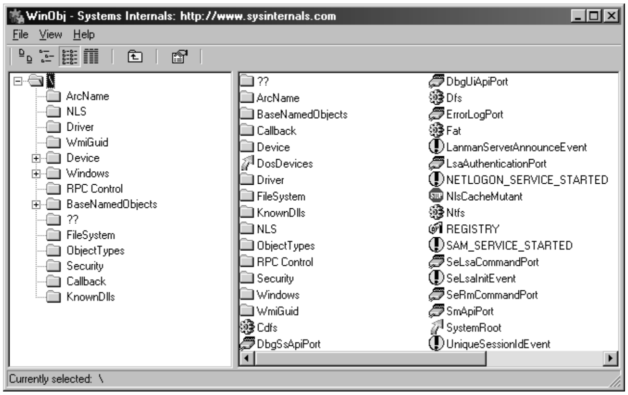
Local Procedure Calls (LPCs)

A local procedure call (LPC) is an interprocess communication facility for high-speed message passing. It is not directly available through the Windows API; it is an internal mechanism available only to Windows operating system components. Here are some examples of where LPCs are used:

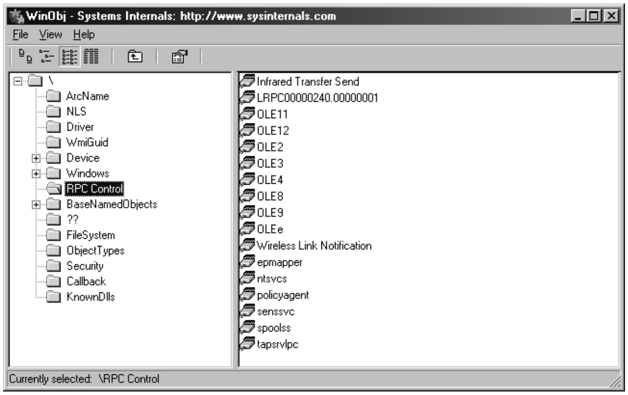
- Windows applications that use remote procedure calls (RPCs), a documented API, indirectly use LPCs when they specify local-RPC, a form of RPC used to communicate between processes on the same system.
- A few Windows APIs result in sending messages to the Windows subsystem process.
- Winlogon uses LPCs to communicate with the local security authentication server process, LSASS.
- The security reference monitor (an executive component explained in Chapter 8) uses LPCs to communicate with the LSASS process.

EXPERIMENT: Viewing LPC Port Objects

You can see named LPC port objects with the Winobj tool from *www.sysinternals.com*. Run Winobj.exe and select the root directory. A plug icon identifies the port objects, as shown here:



To see the LPC port objects used by RPC, select the \RPC Control directory, as shown here:



You can also view LPC port objects by using the *!lpc* kernel debugger command. The command accepts parameters that direct it to show LPC ports, LPC messages, and threads that are waiting or sending LPC messages. To view the LSASS authentication port (the port that Winlogon sends logon requests to), first obtain a list of the ports on the system:

```
kd> !lpc
Usage:
    !lpc                     - Display this help
    !lpc message [MessageId] - Display the message with a
                               given ID and all related
                               information
                               If MessageId is not
                               specified, dump all messages
    !lpc port [PortAddress]  - Display the port information
    !lpc scan PortAddress    - Search this port and any
                               connected port
    !lpc thread [ThreadAddr] - Search the thread in rundown
                               port queues and display the
                               port info
                               If ThreadAddr is missing,
                               display all threads marked
                               as doing some lpc operations

kd> !lpc port
Scanning 206 objects
       1  Port: 0xe1360320 Connection: 0xe1360320
          Communication: 0x00000000 'SeRmCommandPort'
       1  Port: 0xe136bc20 Connection: 0xe136bc20
          Communication: 0x00000000 'SmApiPort'
       1  Port: 0xe133ba80 Connection: 0xe133ba80
          Communication: 0x00000000 'DbgSsApiPort'
       1  Port: 0xe13606e0 Connection: 0xe13606e0
          Communication: 0x00000000 'DbgUiApiPort'
       §
       1  Port: 0xe205f040 Connection: 0xe205f040
          Communication: 0x00000000 'LsaAuthenticationPort'
       §
```

Locate the port named LsaAuthenticationPort in the output and then examine it by passing its address to the *!lpc* command, as shown in the following code segment.

```
kd> !lpc port 0xe205f040
Server connection port e205f040 Name: LsaAuthenticationPort
    Handles: 1   References: 37
    Server process  : ff7d56c0 (lsass.exe)
    Queue semaphore : ff7bfcc8
    Semaphore state 0 (0x0)
    The message queue is empty
    The LpcDataInfoChainHead queue is empty
```

Typically, LPCs are used between a server process and one or more client processes of that server. An LPC connection can be established between two user-mode processes or between a kernel-mode component and a user-mode process. For example, as noted in Chapter 2, Windows processes send occasional messages to the Windows subsystem by using LPCs. Also, some system processes use LPCs to communicate, such as Winlogon and Lsass. An example of a kernel-mode component using an LPC to talk to a user process is the communication between the security reference monitor and the LSASS process.

LPCs are designed to allow three methods of exchanging messages:

- A message that is shorter than 256 bytes can be sent by calling the LPC with a buffer containing the message. This message is then copied from the address space of the sending process into system address space, and from there to the address space of the receiving process.
- If a client and a server want to exchange more than 256 bytes of data, they can choose to use a shared section to which both are mapped. The sender places message data in the shared section and then sends a small message to the receiver with pointers to where the data is to be found in the shared section.
- When a server wants to read or write larger amounts of data than will fit in a shared section, data can be directly read from or written to a client's address space. The LPC component supplies two functions that a server can use to accomplish this. A message sent by the first function is used to synchronize the message passing.

An LPC exports a single executive object called the *port object* to maintain the state needed for communication. Although an LPC uses a single object type, it has several kinds of ports:

- **Server connection port** A named port that is a server connection request point. Clients can connect to the server by connecting to this port.
- **Server communication port** An unnamed port a server uses to communicate with a particular client. The server has one such port per active client.
- **Client communication port** An unnamed port a particular client thread uses to communicate with a particular server.
- **Unnamed communication port** An unnamed port created for use by two threads in the same process.

LPCs are typically used as follows: A server creates a named server connection port object. A client makes a connect request to this port. If the request is granted, two new unnamed ports, a client communication port and a server communication port, are created. The client gets a handle to the client communication port, and the server gets a handle to the server communication port. The client and the server will then use these new ports for their communication.

A completed connection between a client and a server is shown in Figure 3-30.

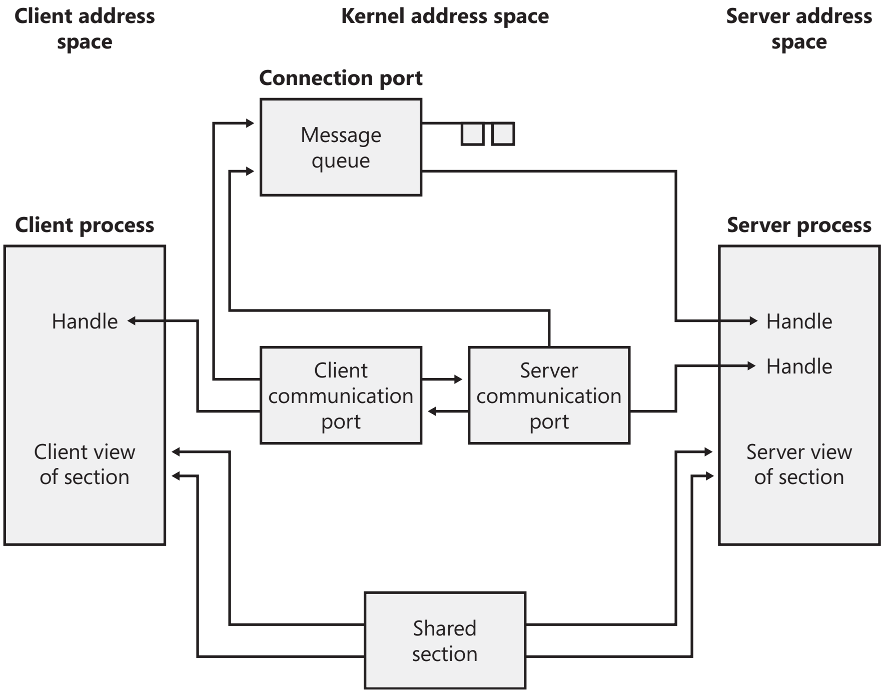


Figure 3-30 Use of LPC ports

Kernel Event Tracing

Various components of the Windows kernel and several core device drivers are instrumented to record trace data of their operation for use in system troubleshooting. They rely on a common infrastructure in the kernel that provides trace data to the user-mode Event Tracing for Windows (ETW) facility. An application that uses ETW falls into one or more of three categories:

- **Controller** A controller starts and stops logging sessions and manages buffer pools.
- **Provider** A provider defines GUIDs (globally unique identifiers) for the event classes it can produce traces for and registers them with ETW. The provider accepts commands from a controller for starting and stopping traces of the event classes for which it's responsible.
- **Consumer** A consumer selects one or more trace sessions for which it wants to read trace data. They can receive the events in buffers in real-time or in log files.

Windows Server systems include several built-in providers in user mode, including ones for Active Directory, Kerberos, and Netlogon. ETW defines a logging session with the name NT Kernel Logger (also known as the kernel logger) for use by the kernel and core drivers. The provider for the NT Kernel Logger is implemented by the Windows Management Instrumentation (WMI) device driver (driver name Wmixwdm), which is part of Ntoskrnl.exe. (See the WMI section in Chapter 5 for more information on WMI.) Besides serving as the core of the kernel logger, the driver manages user-mode ETW event class registration.

The WMI driver exports I/O control interfaces for use by the ETW routines in user mode and the device drivers that provide traces data for the kernel logger. (See Chapter 9 for more information on I/O control commands.) It also implements functions for use by the components in Ntoskrnl.exe kernel mode that produce trace output.

When a controller in user mode enables the kernel logger, the ETW library, which is implemented in \Windows\System32\Ntdll.dll, sends an I/O control request to the WMI driver telling it which event classes the controller wants to start tracing. If file logging is configured (as opposed to in-memory logging to a buffer), the WMI driver creates a system thread in the system process that creates a log file. When the WMI driver receives trace events from the enabled trace sources, it records them to a buffer. If it was started, the file logging thread wakes up once per second to dump the contents of the buffers to the log file.

Trace records generated for the kernel logger have a standard ETW trace event header, which records timestamp, process, and thread IDs, as well as information on what class of event the record corresponds to. Event classes can provide additional data specific to their events. For example, disk event class trace records indicate the operation type (read or write), disk number at which the operation is directed, and sector offset and length of the operation.

The trace classes that can be enabled for the kernel logger and the component that generates each class include:

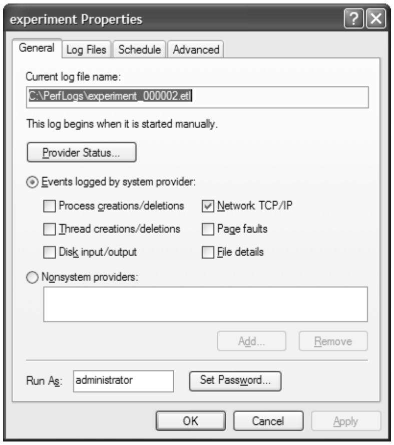
- **Disk I/O** Disk class driver
- **File I/O** File system drivers
- **Hardware Configuration** Plug and play manager (See Chapter 9 for information on the Plug and Play Manager.)
- **Image Load/Unload** The system image loader in the kernel
- **Page Faults** Memory manager (See Chapter 7 for more information on page faults.)
- **Process Create/Delete** Process manager (See Chapter 6 for more information on the process manager.)
- **Thread Create/Delete** Process manager
- **Registry Activity** Configuration manager (See "The Registry" section in Chapter 4 for more information on the configuration manager.)
- **TCP/UDP Activity** TCP/IP driver

You can find more information on ETW and the kernel logger, including sample code for controllers and consumers, in the Platform SDK.

EXPERIMENT: Tracing TCP/IP Activity with the Kernel Logger

To enable the kernel logger and have it generate a log file of TCP/IP activity, follow these steps:

1. Run the Performance Tool, and select the Performance Logs And Alerts node.
2. Select Trace Logs, and then select New Log Settings from the Action menu.
3. When prompted, enter a name for the settings (for example, **experiment**).
4. On the dialog box that opens, select the Events Logged By System Provider option and then deselect everything except the Network TCP/IP option.
5. In the Run As edit box, enter the Administrator account name and set the password to match it.



Current log file name: C:\PerfLogs\experiment_000002.etl

This log begins when it is started manually.

● Events logged by system provider:

☐ Process creations/deletions ☑ Network TCP/IP
☐ Thread creations/deletions ☐ Page faults
☐ Disk input/output ☐ File details

○ Nonsystem providers:

Run As: administrator

6. Dismiss the dialog box, and generate network activity by opening a browser and visiting a Web site.
7. Select the trace log you created in the trace log node, and select Stop from the Action menu.
8. Open a command prompt, and change to the C:\Perflogs directory (or the directory into which you specified that the trace log file be stored).
9. If you are running Windows XP or Windows Server 20003, run Tracerpt (located in the \Windows\System32 directory) and pass it the name of the trace log file. If you are running Windows 2000, download and run Tracedmp from the Windows 2000 Resource Kit. Both tools generate two files: dumpfile.csv and summary.txt.

10. Open dumpfile.csv in Microsoft Excel or in a text editor. You should see TCP and/or UDP trace records like the following:

TcpIp	Recv	0xFFFFFFFF	1.27E+17	0	0	4	88	192.168.001.101	192.168.001.108	4608	0	0	0
TcpIp	Send	0xFFFFFFFF	1.27E+17	0	0	4	76	192.168.001.101	192.168.001.108	4608	0	0	0
TcpIp	Recv	0xFFFFFFFF	1.27E+17	0	0	4	88	192.168.001.101	192.168.001.108	4608	0	0	0
TcpIp	Send	0xFFFFFFFF	1.27E+17	0	0	4	76	192.168.001.101	192.168.001.108	4608	0	0	0
TcpIp	Recv	0xFFFFFFFF	1.27E+17	0	0	4	88	192.168.001.101	192.168.001.108	4608	0	0	0
TcpIp	Send	0xFFFFFFFF	1.27E+17	0	0	4	76	192.168.001.101	192.168.001.108	4608	0	0	0
TcpIp	Recv	0xFFFFFFFF	1.27E+17	0	0	4	88	192.168.001.101	192.168.001.108	4608	0	0	0
TcpIp	Send	0xFFFFFFFF	1.27E+17	0	0	4	76	192.168.001.101	192.168.001.108	4608	0	0	0

Wow64

Wow64 (Win32 emulation on 64-bit Windows) refers to the software that permit the execution of 32-bit x86 applications on 64-bit Windows. It is implemented as a set of user-mode Dlls:

- Wow64.dll: Manages process and thread creation, hooks exception dispatching and base system calls exported by Ntoskrnl.exe. It also implements file system redirection and registry redirection and reflection.
- Wow64Cpu.dll: Manages the 32-bit CPU context of each running thread inside Wow64, and provides processor architecture-specific support for switching CPU mode from 32-bit to 64-bit and vice versa.
- Wow64Win.dll: Intercepts the GUI system calls exported by Win32k.sys.

The relationship of these Dlls is shown in Figure 3-31.

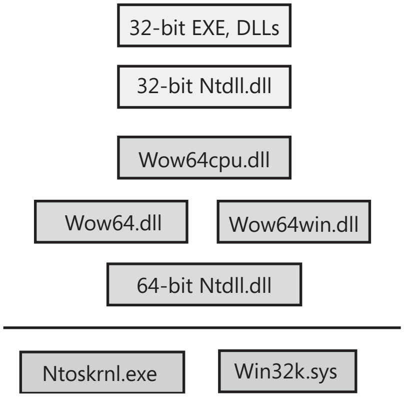


Figure 3-31 Wow64 architecture

Wow64 Process Address Space Layout

Wow64 processes may run with 2 GB or 4 GB of virtual space. If the image header has the large address aware flag set, then the memory manager will reserve the user mode address space above the 4 GB boundary through the end of the user mode boundary. If the image is not marked large address space aware, the memory manager will reserve the user mode address space above 2 GB. (For more information on large address space support, see the section "x86 User Address Space Layouts" in Chapter 7.)

System Calls

Wow64 hooks all the code paths where 32-bit code would transition to the native 64-bit system or when the native system needs to call into 32-bit user mode code. During process creation, the process manager maps into the process address space the native 64-bit Ntdll.dll. When the loader initialization is called, it inspects the image header and if it is 32-bit x86, it loads Wow64.dll. Wow64 then maps in the 32-bit Ntdll.dll (stored in the \Windows\System32\Syswow64 directory). Wow64 then sets up the startup context inside Ntdll, switches the CPU mode to 32-bits, and starts executing the 32-bit loader. From this point onward, execution continues as if the process is running on a native 32-bit system.

Special 32-bit versions of Ntdll.dll, User32.dll, and Gdi32.dll are located in the \Windows\System32\Syswow64 folder. These call into Wow64 rather than issuing the native 32-bit system call instruction. Wow64 transitions to native 64-bit mode, captures the parameters associated with the system call (converting 32-bit pointers to 64-bit pointers), and issues the corresponding native 64-bit system call. When the native system call returns, Wow64 converts any output parameters if necessary from 64-bit to 32-bit formats before returning to 32-bit mode.

Exception Dispatching

Wow64 hooks exception dispatching through ntdll's KiUserExceptionDispatcher. Whenever the 64-bit kernel is about to dispatch an exception to a Wow64 process, Wow64 captures the native exception and context record in user mode and then prepares a 32-bit exception and context record and dispatches it the same way the native 32-bit kernel would do.

User Callbacks

Wow64 intercepts all callbacks from the kernel into user mode. Wow64 treats such calls as system calls; however, the data conversion is done in the reverse order: input parameters are converted from 64-bits to 32-bits and output parameters are converted when the callback returns from 32-bit to 64-bit.

File System Redirection

To maintain application compatibility and to reduce the effort of porting applications from Win32 to 64-bit Windows, system directory names were kept the same. Therefore, the \Windows\System32 contains native 64-bit images. Wow64, as it hooks all the system calls, translates all the path-related APIs, and replaces the path name of the \Windows\System32 folder with \Windows\System32\Syswow64. Wow64 also redirects \Windows\System32\Ime to \Windows\System32\IME (x86) to help 32-bit application compatibility on 64-bit systems with Far East languages installed. Also, 32-bit programs are installed in \Program Files (x86), while 64-bit programs go in the normal \Program Files folder.

There are a few subdirectories of \Windows\System32 which, for compatibility reasons, are exempted from being redirected such that accesses to them made by 32-bit applications actually access the real one. These directories include:

- %windir%\system32\drivers\etc
- %windir%\system32\spool
- %windir%\system32\catroot2
- %windir%\system32\logfiles

Finally, Wow64 provides a mechanism to disable the file system redirection built into Wow64 on a per-thread basis using the *Wow64EnableWow64FsRedirection* function, available on Windows Server 2003 and later.

Registry Redirection and Reflection

Applications and components store their configuration data in the registry. Components usually write their configuration data in the registry when they are registered during installation. If the same component is installed and registered both as a 32-bit binary and a 64-bit binary, then the last component being registered will override the registration of the previous component as they both write to the same location in the registry.

To help solve this problem transparently without introducing any code changes to 32-bit components, the registry is split into two portions: Native and Wow64. By default, 32-bit components access the 32-bit view, and 64-bit components access the 64-bit view. This provides a safe execution environment for 32-bit and 64-bit components and separates the 32-bit application state from the 64-bit one if it exists.

To implement this, Wow64 intercepts all the system calls that open registry keys and re-translates the key path to point it to the Wow64 view of the registry. Wow64 splits the registry at these points:

- HKLM\Software
- HKEY_CLASSES_ROOT

- HKEY_CURRENT_USER\Software\Classes

Under each of these keys, Wow64 creates a key called Wow6432Node. Under this key is stored 32-bit configuration information. All other portions of the registry are shared between 32-bit and 64-bit applications (e.g., HKLM\System).

For applications that need to explicitly specify a registry key for a certain view, the following flags on the *RegOpenKeyEx* and *RegCreateKeyEx* permit this:

- KEY_WOW64_64KEY – explicitly opens a 64-bit key from either a 32-bit or 64-bit application
- KEY_WOW64_32KEY – explicitly opens a 32-bit key from either a 32-bit or 64-bit application

To enable interoperability through 32-bit and 64-bit COM components, Wow64 mirrors certain portions of the registry when updated in one view to the other. It does this by intercepting updates to any of the reflected keys, and mirrors the changes intelligently to the other view of the registry. The list of reflected keys is:

- HKEY_LOCAL_MACHINE\Software\Classes
- HKEY_LOCAL_MACHINE\Software\Ole
- HKEY_LOCAL_MACHINE\Software\Rpc
- HKEY_LOCAL_MACHINE\Software\COM3
- HKEY_LOCAL_MACHINE\Software\EventSystem

Reflection of HKLM\Software\Classes\CLSID is intelligent; only LocalServer32 CLSIDs are reflected because they run out of process, thus they can be COM-activated by 32-bit or 64-bit applications. However, InProcServer32 CLSIDs are not reflected because 32-bit COM DLLs can't be loaded in a 64-bit process and likewise 64-bit COM DLLs can't be loaded in a 32-bit process.

When reflecting a key/value, the registry reflector marks the key so that it understands that it has been created by the reflector. This is to help the deletion case when deleting a key that has been reflected; thus the reflector will be able to tell if it needs to delete the reflected key if it has been written by the reflector.

I/O Control Requests

Besides normal read and write operations, applications can communicate with some device drivers through device I/O control functions using the Windows *DeviceIoControlFile* API. The application may specify an input and/or output buffer along with the call. If the buffer contains pointer-dependent data, and the process sending the control request is a Wow64 process, then the view of the input and/or output structure is different between the 32-bit application and the 64-bit driver, since pointers are 4 bytes for 32-bit applications and 8 bytes for 64-bit applications. In this case, the kernel driver is expected to convert the associated pointer-dependent structures. Drivers can call the *IoIs32bitProcess* function to detect if an I/O request originated from a Wow64 process or not.

16-Bit Installer Applications

Wow64 doesn't support running 16-bit applications. However, since many application installers are 16-bit programs, Wow64 has special case code to make references to certain well known 16-bit installers work. These installers include:

- Microsoft ACME Setup version: 2.6, 3.0, 3.01, and 3.1.
- InstallShield version 5.x (where x is any minor version number).

Whenever a 16-bit process is about to be created using CreateProcess() API, Ntvdm64.dll is loaded and control is transferred to it to inspect whether the 16-bit executable is one of the supported installers. If it is, another CreateProcess is issued to launch a 32-bit version of the installer with the same command line arguments.

Printing

32-bit printer drivers cannot be used on 64-bit Windows. Print drivers must be ported to native 64-bit versions. However, since printer drivers run in the user mode address space of the requesting process, and since only native 64-bit printer drivers are supported on 64-bit Windows, a special mechanism is needed to support printing from 32-bit processes. This is done by redirecting all printing functions to Splwow64.exe, the Wow64 RPC print server. Since Splwow64 is a 64-bit process, it can load 64-bit printer drivers.

Restrictions

Wow64 does not support the execution of 16-bit applications (this is supported on 32-bit versions of Windows) or the loading of 32-bit kernel mode device drivers (they must be ported to native 64-bits). Wow64 processes can only load 32-bit DLLs and can't load native 64-bit Dlls. Likewise, native 64-bit processes can't load 32-bit DLLs.

In addition to the above, due to page size differences, Wow64 on IA-64 systems does not support the *ReadFileScatter*, *WriteFileGather*, *GetWriteWatch,* or Address Window Extension (AWE) functions. Also, hardware acceleration through DirectX is not available (software emulation is provided for Wow64 processes).

Conclusion

In this chapter, we've examined the key base system mechanisms on which the Windows executive is built. In the next chapter, we'll look at three important mechanisms involved with the management infrastructure of Windows: the registry, services, and Windows Management Instrumentations (WMI).

Chapter 4

Management Mechanisms

This chapter describes three fundamental mechanisms in Microsoft Windows that are critical to the management and configuration of the system:

- The registry
- Services
- Windows Management Instrumentation

The Registry

The registry plays a key role in the configuration and control of Windows systems. It is the repository for both systemwide and per-user settings. Although most people think of the registry as static data stored on the hard disk, as you'll see in this section, the registry is also a window into various in-memory structures maintained by the Windows executive and kernel. This section isn't meant to be a complete reference to the contents of the Windows registry. That kind of in-depth information is documented in the "Technical Reference to the Windows 2000 Registry" help file in the Windows 2000 resource kits (Regentry.chm), and for Windows XP and Windows Server 2003 that information can be found online as part of the Windows Server 2003 Deployment Kit at *http://www.microsoft.com/windowsserver2003/techinfo/reskit/deploykit.mspx*.

We'll start by providing you with an overview of the registry structure, a discussion of the data types it supports, and a brief tour of the key information Windows maintains in the registry. Then we'll look inside the internals of the *configuration manager*, the executive component responsible for implementing the registry database. Among the topics we'll cover are the internal on-disk structure of the registry, how Windows retrieves configuration information when an application requests it, and what measures are employed to protect this critical system database.

Viewing and Changing the Registry

In general, you should never have to edit the registry directly: application and system settings stored in the registry that might require manual changes should have a corresponding user interface to control their modification. However, as you've already seen a number of times in this book, some advanced and debug settings have no editing user interface. Therefore, a number of tools are included with Windows that enable you to view and modify the registry.

Windows 2000 comes with two tools for editing the registry—Regedit.exe and Regedt32.exe—whereas Windows XP and Windows Server 2003 have only Regedit.exe. The reason is that the Windows 2000 version of Regedit, which has flexible searching, importing, and exporting capabilities, was ported from Windows 98 and therefore does not support editing or viewing registry security or registry data types not defined on Windows 98. Windows 2000 includes Regedt32 because although it doesn't have as powerful a search feature or support importing and exporting, it was written to run only on Windows 2000 and so it supports security and Windows 2000–specific data types. The Regedit included with Windows XP and Windows Server 2003 includes security editing and knowledge of all registry data types, and thus obviates the need for Regedt32.

There are also a number of command-line registry tools. Reg.exe, for instance, which is included in Windows XP and Windows Server 2003 and available in the Windows 2000 Support Tools, has the ability to import, export, back up, and restore keys, as well as to compare, modify, and delete keys and values.

Registry Usage

There are three principal times that configuration data is read:

- During the boot process, the system reads settings that specify what device drivers to load and how various subsystems—such as the memory manager and process manager—configure themselves and tune system behavior.
- During login, Explorer and other Windows components read per-user preferences from the registry, including network drive-letter mappings, desktop wallpaper, screen saver, menu behavior, and icon placement.
- During their startup, applications read systemwide settings, such as a list of optionally installed components and licensing data, as well as per-user settings that might include menu and toolbar placement and a list of most-recently accessed documents.

However, the registry can be read at other times as well, such as in response to a modification of a registry value or key. Some applications monitor their configuration settings in the registry and read updated settings when they see a change. In general, however, on an idle system there should be no registry activity.

The registry is commonly modified in the following cases:

- Although not a modification, the registry's initial structure and many default settings are defined by a prototype version of the registry that ships on the Windows setup media that is copied onto a new installation.
- Application setup utilities create default application settings and settings that reflect installation configuration choices.

- During the installation of a device driver, the Plug and Play system creates settings in the registry that tell the I/O manager how to start the driver and creates other settings that configure the driver's operation. (See Chapter 9 for more information on how device drivers are installed.)
- When you change application or system settings through user interfaces, the changes are often stored in the registry.

Note Sadly, some applications poll the registry looking for changes when they should be using the registry's *RegNotifyChangeKey* function, which puts a thread to sleep until a change occurs to the area of the registry in which they're interested.

Registry Data Types

The registry is a database whose structure is similar to that of a disk volume. The registry contains *keys*, which are similar to a disk's directories, and *values*, which are comparable to files on a disk. A key is a container that can consist of other keys (*subkeys*) or values. Values, on the other hand, store data. Top-level keys are *root keys*. Throughout this section, we'll use the words *subkey* and *key* interchangeably. (Only root keys are not subkeys.)

Both keys and values borrow their naming convention from the file system. Thus, you can uniquely identify a value with the name mark, which is stored in a key called trade, with the name trade\mark. One exception to this naming scheme is each key's unnamed value. The two Registry Editor utilities, Regedit and Regedt32, display these values differently: Regedit displays the unnamed value as (Default); Regedt32 uses <No Name>.

Values store different kinds of data and can be one of the 15 types listed in Table 4-1. The majority of registry values are REG_DWORD, REG_BINARY, or REG_SZ. Values of type REG_DWORD can store numbers or Booleans (on/off values); REG_BINARY values can store numbers larger than 32 bits or raw data such as encrypted passwords; REG_SZ values store strings (Unicode, of course) that can represent elements such as names, filenames, paths, and types.

Table 4-1 Registry Value Types

Value Type	Description
REG_NONE	No value type.
REG_SZ	Fixed-length Unicode string.
REG_EXPAND_SZ	Variable-length Unicode string that can have embedded environment variables.
REG_BINARY	Arbitrary-length binary data.
REG_DWORD	32-bit number.
REG_DWORD_LITTLE_ENDIAN	32-bit number, with low byte first. This is equivalent to REG_DWORD.
REG_DWORD_BIG_ENDIAN	32-bit number, with high byte first.
REG_LINK	Unicode symbolic link.

Table 4-1 Registry Value Types

Value Type	Description
REG_MULTI_SZ	Array of Unicode NULL-terminated strings.
REG_RESOURCE_LIST	Hardware resource description.
REG_FULL_RESOURCE_DESCRIPTOR	Hardware resource description.
REG_RESOURCE_REQUIREMENTS_LIST	Resource requirements.
REG_QWORD	64-bit number.
REG_QWORD_LITTLE_ENDIAN	64-bit number, with low byte first. This is equivalent to REG_QWORD.
REG_QWORD_BIG_ENDIAN	64-bit number, with high byte first.

The REG_LINK type is particularly interesting because it lets a key transparently point to another key or value. When you traverse the registry through a link, the path searching continues at the target of the link. For example, if \Root1\Link has a REG_LINK value of \Root2\RegKey, and RegKey contains the value RegValue, two paths identify RegValue: \Root1\Link\RegValue and \Root2\RegKey\RegValue. As explained in the next section, Windows prominently uses registry links: three of the six registry root keys are links to subkeys within the three nonlink root keys. Links aren't saved; they must be dynamically created after each reboot.

Registry Logical Structure

You can chart the organization of the registry via the data stored within it. There are six root keys (and you can't add new root keys or delete existing ones) that store information, as shown in Table 4-2.

Table 4-2 The Six Root Keys

Root Key	Description
HKEY_CURRENT_USER	Stores data associated with the currently logged-on user
HKEY_USERS	Stores information about all the accounts on the machine
HKEY_CLASSES_ROOT	Stores file association and Component Object Model (COM) object registration information
HKEY_LOCAL_MACHINE	Stores system-related information
HKEY_PERFORMANCE_DATA	Stores performance information
HKEY_CURRENT_CONFIG	Stores some information about the current hardware profile

Why do root-key names begin with an H? Because the root-key names represent Windows handles (H) to keys (KEY). As mentioned in Chapter 1, HKLM is an abbreviation used for HKEY_LOCAL_MACHINE. Table 4-3 lists all the root keys and their abbreviations. The following sections explain in detail the contents and purpose of each of these six root keys. Again, see the "Technical Reference to the Windows 2000 Registry" help file in the Windows 2000 resource kits or the registry section of the Windows Server 2003 Deployment Kit for details on the contents of these keys.

Table 4-3 Registry Root Keys

Root Key	Abbreviation	Description	Link
HKEY_CURRENT_USER	HKCU	Points to the user profile of the currently logged-on user	Subkey under HKEY_USERS corresponding to currently logged-on user
HKEY_USERS	HKU	Contains subkeys for all loaded user profiles	Not a link
HKEY_CLASSES_ROOT	HKCR	Contains file association and COM registration information	HKLM\SOFTWARE\Classes
HKEY_LOCAL_MACHINE	HKLM	Placeholder—contains other keys	Not a link
HKEY_CURRENT_CONFIG	HKCC	Current hardware profile	HKLM\SYSTEM\CurrentControlSet\Hardware Profiles\Current
HKEY_PERFORMANCE_DATA	HKPD	Performance counters	Not a link

HKEY_CURRENT_USER

The HKCU root key contains data regarding the preferences and software configuration of the locally logged-on user. It points to the currently logged-on user's user profile, located on the hard disk at \Documents and Settings\<username>\Ntuser.dat. (See the section "Registry Internals" later in this chapter to find out how root keys are mapped to files on the hard disk.) Whenever a user profile is loaded (such as at logon time or when a service process runs under the context of a specific username), HKCU is created as a link to the user's key under HKEY_USERS. Table 4-4 lists some of the subkeys under HKCU.

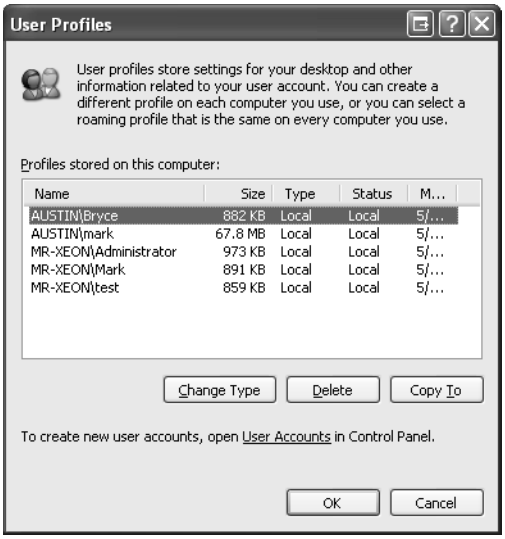
Table 4-4 HKEY_CURRENT_USER Subkeys

Subkey	Description
AppEvents	Sound/event associations
Console	Command window settings (for example, width, height, and colors)
Control Panel	Screen saver, desktop scheme, keyboard, and mouse settings as well as accessibility and regional settings
Environment	Environment variable definitions
Keyboard Layout	Keyboard layout setting (for example, U.S. or U.K.)
Network	Network drive mappings and settings
Printers	Printer connection settings
Software	User-specific software preferences
UNICODE Program Groups	User-specific start menu group definitions
Windows 3.1 Migration Status	File status data for systems that upgrade from Windows 3.x to Windows 2000 and higher

HKEY_USERS

HKU contains a subkey for each loaded user profile and user class registration database on the system. It also contains a subkey named HKU\.DEFAULT that is linked to the profile for the system (which is used by processes running under the local system account and is described in more detail in the section “Services” later in this chapter). This is the profile used by Winlogon, for example, so that changes to the desktop background settings in that profile will be implemented on the logon screen. When a user logs on to a system for the first time and her account does not depend on a roaming domain profile (that is, the user’s profile is obtained from a central network location at the direction of a domain controller), the system creates a profile for her account that’s based on the profile stored in C:\Documents and Settings\Default User.

The location under which the system stores profiles is defined by the registry value HKLM\Software\Microsoft\Windows NT\CurrentVersion\ProfileList\ProfilesDirectory, which is by default set to %SystemDrive%\Documents and Settings. The ProfileList key also stores the list of profiles present on a system. Information for each profile resides under a subkey that has a name reflecting the Security Identifier (SID) of the account to which the profile corresponds. (See Chapter 8 for more information on SIDs.) Data stored in a profile’s key includes the time of the last load of the profile in the *ProfileLoadTimeLow* and *ProfileLoadTimeHigh* values, the binary representation of the account SID in the *Sid* value, and the path to the profile’s on-disk hive (which is described later in this chapter in the “Hives” section) in the ProfileImagePath directory. Windows XP and Windows Server 2003 show the list of profiles stored on a system in the User Profiles management dialog box, shown in Figure 4-1, that you access by clicking Settings in the User Profiles section of the Advanced Tab on the System Control Panel applet.



User profiles store settings for your desktop and other information related to your user account. You can create a different profile on each computer you use, or you can select a roaming profile that is the same on every computer you use.

Profiles stored on this computer:

Name	Size	Type	Status	M...
AUSTIN\Bryce	882 KB	Local	Local	5/...
AUSTIN\mark	67.8 MB	Local	Local	5/...
MR-XEON\Administrator	973 KB	Local	Local	5/...
MR-XEON\Mark	891 KB	Local	Local	5/...
MR-XEON\test	859 KB	Local	Local	5/...

To create new user accounts, open User Accounts in Control Panel.

Figure 4-1 The User Profiles management dialog box

EXPERIMENT: Watching Profile Loading and Unloading

You can see a profile load into the registry and then unload by using the Runas command to launch a process in an account that's not currently logged on to the machine. While the new process is running, run Regedit and note the loaded profile key under HKEY_USERS. After terminating the process, perform a refresh in Regedit by pressing the F5 key and the profile should no longer be present.

HKEY_CLASSES_ROOT

HKCR consists of two types of information: file extension associations and COM class registrations. A key exists for every registered filename extension. Most keys contain a REG_SZ value that points to another key in HKCR containing the association information for the class of files that extension represents. For example, HKCR\.xls would point to information on Microsoft Excel files in a key such as HKCU\Excel.Sheet.8. Other keys contain configuration details for COM objects registered on the system.

The data under HKEY_CLASSES_ROOT comes from two sources:

- The per-user class registration data in HKCU\SOFTWARE\Classes (mapped to the file on hard disk \Documents and Settings\<username>\Local Settings\Application Data\Microsoft\Windows\Usrclass.dat)
- Systemwide class registration data in HKLM\SOFTWARE\Classes

The reason that there is a separation of per-user registration data from systemwide registration data is so that roaming profiles can contain these customizations. It also closes a security hole: a nonprivileged user cannot change or delete keys in the systemwide version HKEY_CLASSES_ROOT, and thus cannot affect the operation of applications on the system. Nonprivileged users and applications can read systemwide data and can add new keys and values to systemwide data (which are mirrored in their per-user data), but they can modify existing keys and values in their private data only.

HKEY_LOCAL_MACHINE

HKLM is the root key that contains all the systemwide configuration subkeys: HARDWARE, SAM, SECURITY, SOFTWARE, and SYSTEM.

The HKLM\HARDWARE subkey maintains descriptions of the system's hardware and all hardware device-to-driver mappings. The Device Manager tool (which is available by running System from Control Panel, clicking the Hardware tab, and then clicking Device Manager) lets you view registry hardware information that it obtains by simply reading values out of the HARDWARE key.

EXPERIMENT: Fun with the Hardware Key

You can fool your coworkers or friends into thinking that you have the latest and greatest processor by modifying the value of the *ProcessorNameString* value under HKLM\HARDWARE\DESCRIPTION\System\CentralProcessor\0. The System applet of the control panel displays the *ProcessorNameString* value on the General page. Changes you make to other values in that key, such as the ~MHz, do not have any affect on what the System applet displays, however, because the system caches many of the values for use by functions that applications use to query the system's processor capabilities.

HKLM\SAM holds local account and group information, such as user passwords, group definitions, and domain associations. Windows Server systems that are operating as domain controllers store domain accounts and groups in Active Directory, a database that stores domainwide settings and information. (Active Directory isn't described in this book.) By default, the security descriptor on the SAM key is configured so that even the administrator account doesn't have access.

HKLM\SECURITY stores systemwide security policies and user-rights assignments. HKLM\SAM is linked into the SECURITY subkey under HKLM\SECURITY\SAM. By default, you can't view the contents of HKLM\SECURITY or HKLM\SAM\SAM because the security settings of those keys allow access only by the system account. (System accounts are discussed in greater detail later in this chapter.) You can change the security descriptor to allow read access to administrators, or you can use PsExec to run Regedit in the local system account (as shown in the related experiment for how to do that) if you want to peer inside. However, that glimpse won't be very revealing because the data is undocumented and the passwords are encrypted with one-way mapping—that is, you can't determine a password from its encrypted form.

HKLM\SOFTWARE is where Windows stores systemwide configuration information not needed to boot the system. Also, third-party applications store their systemwide settings here, such as paths to application files and directories, and licensing and expiration date information.

HKLM\SYSTEM contains the systemwide configuration information needed to boot the system, such as which device drivers to load and which services to start. Because this information is critical to starting the system, Windows also maintains a copy of part of this information, called the *last known good control set*, under this key. The maintenance of a copy allows an administrator to select a previously working control set in the case that configuration changes made to the current control set prevent the system from booting. For details on when Windows declares the current control set "good," see the section "Accepting the Boot and Last Known Good."

HKEY_CURRENT_CONFIG

HKEY_CURRENT_CONFIG is just a link to the current hardware profile, stored under HKLM\SYSTEM\CurrentControlSet\Hardware Profiles\Current. Hardware profiles allow the administrator to configure variations to the base system driver settings. Although the underlying profile might change from boot to boot, applications can always reference the currently active profile through this key. Hardware profile management is managed through the Hardware Profiles dialog box that you access by clicking Settings in the Hardware Profiles section on the Hardware page of the Control Panel's System applet. During the boot process, Ntldr will prompt you to specify which profile it should use if there is more than one.

HKEY_PERFORMANCE_DATA

The registry is the mechanism to access performance counter values on Windows, whether those are from operating system components or server applications. One of the side benefits of providing access to the performance counters via the registry is that remote performance monitoring works "for free" because the registry is easily accessible remotely through the normal registry APIs.

You can access the registry performance counter information directly by opening a special key named HKEY_PERFORMANCE_DATA and querying values beneath it. You won't find this key by looking in the Registry Editor; this key is available only programmatically through the Windows registry functions, such as *RegQueryValueEx*. Performance information isn't actually stored in the registry; the registry functions use this key to locate the information from performance data providers.

You can also access performance counter information by using the Performance Data Helper (PDH) functions available through the Performance Data Helper API (Pdh.dll). Figure 4-2 shows the components involved in accessing performance counter information.

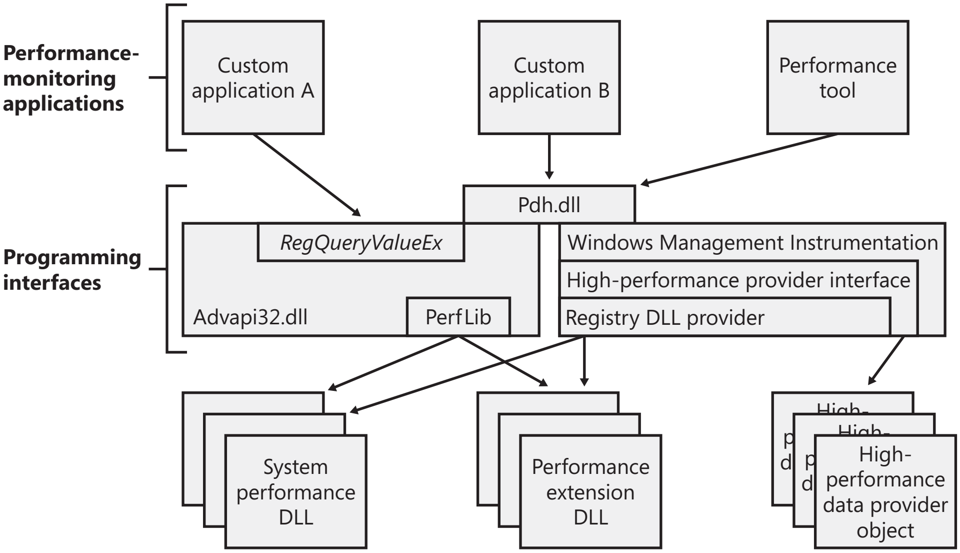

Performance-monitoring applications
Custom application A
Custom application B
Performance tool
Programming interfaces
Pdh.dll
RegQueryValueEx
Advapi32.dll
PerfLib
Windows Management Instrumentation
High-performance provider interface
Registry DLL provider
System performance DLL
Performance extension DLL
High-performance data provider object

Figure 4-2 Registry performance counter architecture

Troubleshooting Registry Problems

Because the system and applications depend so heavily on configuration settings to guide their behavior, system and application failures can result from changing registry data or security. When the system or an application fails to read settings that it assumes it will always be able to access, it can misbehave by crashing, displaying error messages that hide the root cause, or by not executing with limited functionality. It's virtually impossible to know what registry keys or values are misconfigured without understanding how the system or the application that's failing is accessing the registry. In such situations, the Regmon utility from *www.sysinternals.com* might provide the answer.

Regmon lets you monitor registry activity as it occurs. For each registry access, Regmon shows you the process that performed the access and the time, type, and result of the access. This information is useful for seeing how applications and the system rely on the registry, discovering where applications and the system store configuration settings and troubleshooting problems related to applications having missing registry keys or values. Regmon includes advanced filtering and highlighting so that you can zoom in on activity related to specific keys or values, or to the activity of particular processes.

Regmon Internals

Regmon relies on a device driver that it extracts from its executable image at run time and then starts. Its first execution requires that the account running it have the Load Driver privilege as well as the Debug privilege; subsequent executions in the same boot session require only the Debug privilege because once loaded, the driver remains resident.

There are actually three drivers stored within the Regmon executable: one for use on Windows 95, Windows 98, and Windows Millennium; one for Windows NT, Windows 2000, and Windows XP; and another for use on Windows Server 2003. The reason that there is a driver specific to Windows Server 2003 is that on Windows NT, Windows 2000, and Windows XP the only way for a driver to monitor all registry activity is through system-call hooking and because on Windows Server 2003 a driver can use the *registry callback mechanism* to monitor registry activity. (Windows 95, Windows 98, and Windows Millennium support a different registry monitoring mechanism.)

Recall from the "System Service Dispatching" section of Chapter 3 that system service function addresses are stored in a system service dispatch table in the kernel. A driver can hook a system service by saving the address of a function from the array and replacing the array entry with the address of its hook function. After performing theses steps, any invocations of the hooked system service get diverted to the hooking driver's function, which can examine or modify the parameters to the function and, optionally, execute the original system service function. If it calls the original function, the driver can also examine the result of the operation and examine data the function returns, such as data associated with registry values. Figure 4-3 shows how Regmon intercepts registry functions in kernel mode.

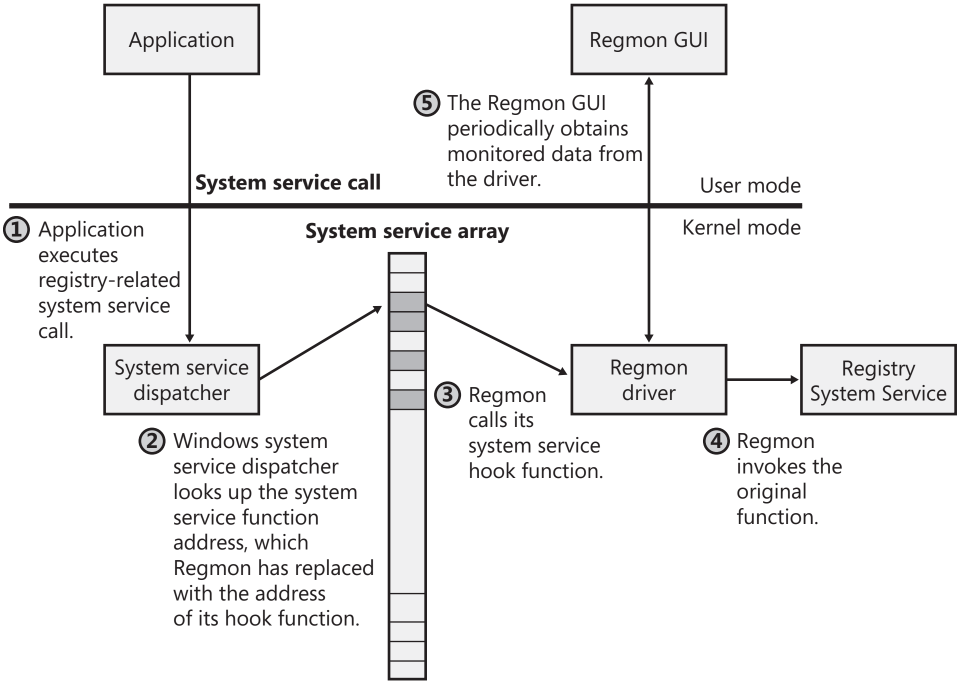

Application
Regmon GUI
5 The Regmon GUI periodically obtains monitored data from the driver.
System service call
User mode
Kernel mode
System service array
1 Application executes registry-related system service call.
System service dispatcher
Regmon driver
Registry System Service
2 Windows system service dispatcher looks up the system service function address, which Regmon has replaced with the address of its hook function.
3 Regmon calls its system service hook function.
4 Regmon invokes the original function.

Figure 4-3 Regmon's use of system service hooking

The registry callback mechanism was introduced in Windows XP; however, Regmon still uses system call hooking when run on Windows XP because the callback mechanism on Windows XP does not report all registry activity. When a driver uses the callback mechanism, it registers a callback function with the configuration manager. The configuration manager executes the driver's callback functions at certain points during the execution of registry system services so that the driver has full visibility and control over registry accesses. Antivirus products that scan registry data for viruses or prevent unauthorized processes from modifying the registry are other users of the callback mechanism.

EXPERIMENT: Viewing Registry Activity on an Idle System

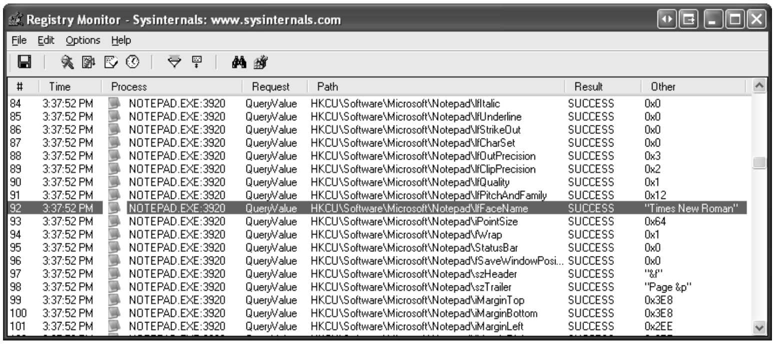
Because the registry implements the *RegNotifyChangeKey* function that applications can use to request notification of registry changes without polling for them, when you Regmon on a system that's idle you should not see repetitive accesses to the same registry keys or values. Any such activity identifies a poorly written application that unnecessarily negatively affects a system's overall performance.

Run Regmon, and after several seconds examine the output log to see whether you can spot polling behavior. Right-click on an output line associated with polling, and choose Process Properties from the context menu to view details about the process performing the activity.

EXPERIMENT: Using Regmon to Locate Application Registry Settings

In some troubleshooting scenarios, you might need to determine where in the registry the system or an application stores particular settings. This experiment has you use Regmon to discover the location of Notepad's settings. Notepad, like most Windows applications, saves user preferences–such as word-wrap mode, font and font size, and window position–across executions. By having Regmon watching when Notepad reads or writes its settings, you can identify the registry key in which the settings are stored. Here are the steps for doing this:

1. Have Notepad save a setting that you can easily search for in a Regmon trace. You can do this by running Notepad, setting the font to Times New Roman, and then exiting Notepad.
2. Run Regmon. Open the highlighting filter dialog box and enter **notepad.exe** in the Include filter. This will have Regmon log only activity that has notepad.exe in either the Process or Path columns.
3. Run Notepad again, and after it has launched stop Regmon's event capture by toggling Capture Events in the Regmon File menu.
4. Scroll to the top line of the resultant log and select it.
5. Press Ctrl+F to open a Find dialog box, and search for **times new**. Regmon should highlight a line like the one shown in the following graphic that represents Notepad reading the font value from the Registry. Other operations in the immediate vicinity should relate to other Notepad settings.



#	Time	Process	Request	Path	Result	Other
84	3:37:52 PM	NOTEPAD.EXE:3920	QueryValue	HKCU\Software\Microsoft\Notepad\lfItalic	SUCCESS	0x0
85	3:37:52 PM	NOTEPAD.EXE:3920	QueryValue	HKCU\Software\Microsoft\Notepad\lfUnderline	SUCCESS	0x0
86	3:37:52 PM	NOTEPAD.EXE:3920	QueryValue	HKCU\Software\Microsoft\Notepad\lfStrikeOut	SUCCESS	0x0
87	3:37:52 PM	NOTEPAD.EXE:3920	QueryValue	HKCU\Software\Microsoft\Notepad\lfCharSet	SUCCESS	0x0
88	3:37:52 PM	NOTEPAD.EXE:3920	QueryValue	HKCU\Software\Microsoft\Notepad\lfOutPrecision	SUCCESS	0x3
89	3:37:52 PM	NOTEPAD.EXE:3920	QueryValue	HKCU\Software\Microsoft\Notepad\lfClipPrecision	SUCCESS	0x2
90	3:37:52 PM	NOTEPAD.EXE:3920	QueryValue	HKCU\Software\Microsoft\Notepad\lfQuality	SUCCESS	0x1
91	3:37:52 PM	NOTEPAD.EXE:3920	QueryValue	HKCU\Software\Microsoft\Notepad\lfPitchAndFamily	SUCCESS	0x12
92	3:37:52 PM	NOTEPAD.EXE:3920	QueryValue	HKCU\Software\Microsoft\Notepad\lfFaceName	SUCCESS	"Times New Roman"
93	3:37:52 PM	NOTEPAD.EXE:3920	QueryValue	HKCU\Software\Microsoft\Notepad\iPointSize	SUCCESS	0x64
94	3:37:52 PM	NOTEPAD.EXE:3920	QueryValue	HKCU\Software\Microsoft\Notepad\fWrap	SUCCESS	0x1
95	3:37:52 PM	NOTEPAD.EXE:3920	QueryValue	HKCU\Software\Microsoft\Notepad\StatusBar	SUCCESS	0x0
96	3:37:52 PM	NOTEPAD.EXE:3920	QueryValue	HKCU\Software\Microsoft\Notepad\fSaveWindowPosi...	SUCCESS	0x0
97	3:37:52 PM	NOTEPAD.EXE:3920	QueryValue	HKCU\Software\Microsoft\Notepad\szHeader	SUCCESS	"&f"
98	3:37:52 PM	NOTEPAD.EXE:3920	QueryValue	HKCU\Software\Microsoft\Notepad\szTrailer	SUCCESS	"Page &p"
99	3:37:52 PM	NOTEPAD.EXE:3920	QueryValue	HKCU\Software\Microsoft\Notepad\iMarginTop	SUCCESS	0x3E8
100	3:37:52 PM	NOTEPAD.EXE:3920	QueryValue	HKCU\Software\Microsoft\Notepad\iMarginBottom	SUCCESS	0x3E8
101	3:37:52 PM	NOTEPAD.EXE:3920	QueryValue	HKCU\Software\Microsoft\Notepad\iMarginLeft	SUCCESS	0x2EE

6. Finally, double-click the highlighted line. Regmon will execute Regedit (if it's not already running) and cause it to navigate to and select the Notepad referenced registry value.

Regmon Troubleshooting Techniques

Two basic Regmon troubleshooting techniques are effective for discovering the cause of registry-related application or system problems:

- Look at the last thing in the Regmon trace that the application did before it failed. This action might point to the problem.
- Compare a Regmon trace of the failing application with a trace from a working system.

To follow the first approach, run Regmon and then run the application. At the point the failure occurs, go back to Regmon and stop the logging (by pressing Ctrl+E). Then go to the end of the log and find the last operations performed by the application before it failed (or crashed, hung, or whatever). Starting with the last line, work your way backward, examining the files, registry keys, or both that were referenced—often this will help pinpoint the problem.

Use the second approach when the application fails on one system but works on another. Capture a Regmon trace of the application on the working and failing systems, and save the output to a log file. Then open the good and bad log files with Microsoft Excel (accepting the defaults on the Import wizard), and delete the first three columns. (If you don't delete the first three columns, the comparison will show every line as different because the first three columns contain information that is different from run to run, such as the time and the process ID.) Finally, compare the resulting log files. (You can do this by using WinDiff, which on Windows XP is included in the free support tools on the Windows XP CD, and for Windows 2000 it is included in the Resource Kit.)

Entries in a Regmon trace that have values of "NOTFOUND" or "ACCESS DENIED" in the Result column are ones that you should investigate. NOTFOUND is reported when an application attempts to read from a registry key or value that doesn't exist. In many cases, a missing key or value is innocuous because a process that fails to read a setting from the registry simply falls back on default values. In some cases, however, applications expect to find values for which there is no default and will fail if they are missing.

Access-denied errors are a common source of registry-related application failures and occur when an application doesn't have permission to access a key the way that it wants. Applications that do not validate registry operation results or perform proper error recovery will fail.

A common result string that might appear suspicious is BUFROVERFLOW. It does not indicate a buffer-overflow exploit in the application that receives it. Instead, it's used by the configuration manager to inform an application that the buffer it specified to store a registry value is too small to hold the value. Application developers often take advantage of this behavior to determine how large a buffer to allocate to store a value. They first perform a registry query with a 0-length buffer that returns a buffer-overflow error and the length of the data it attempted to read. The application then allocates a buffer of the indicated size and rereads the value. You should therefore see operations that return BUFROVERFLOW repeat with a successful result.

In one example of Regmon being used to troubleshoot a real problem, it saved a user from doing a complete reinstall of his Windows XP system. The symptom was that Internet Explorer would hang on startup if the user did not first manually dial the Internet connection. This Internet connection was set as the default connection for the system, so starting Internet Explorer should have caused an automatic dial-up to the Internet (because Internet Explorer was set to display a default home page upon startup).

An examination of a Regmon log of Internet Explorer startup activity, going backward from the point in the log where Internet Explorer hung, showed a query to a key under HKCU\Software\Microsoft\RAS Phonebook. The user reported that he had previously uninstalled the dialer program associated with the key and manually created the dial-up connection. Because the dial-up connection name did not match that of the uninstalled dialer program, it appeared that the key had not been deleted by the dialer's uninstall program and that it was causing Internet Explorer to hang. After the key was deleted, Internet Explorer functioned as expected.

Logging Activity in Unprivileged Accounts or During Logon/Logoff

A common application-failure scenario is that an application works when run in an account that has Administrative group membership but not when run in the account of an unprivileged user. As described earlier, executing Regmon requires security privileges that are not normally assigned to standard user accounts, but you can capture a trace of applications executing in the logon session of an unprivileged user by using the Runas command to execute Regmon in an administrative account.

If a registry problem relates to account logon or logoff, you'll also have to take special steps to be able to use Regmon to capture a trace of those phases of a logon session. Applications that are run in the local system account are not terminated when a user logs off, and you can take advantage of that fact to have Regmon run through a logoff and subsequent logon. You can launch Regmon in the local system account either by using the *At* command that's built into Windows and specifying the */interactive* flag, or by using the PsExec utility from *www.sysinternals.com*, like this:

psexec –i –s –d c:\regmon.exe

The -i switch directs PsExec to have Regmon's window appear on the interactive console, the -s switch has PsExec run Regmon in the local system account, and the -d switch has PsExec launch Regmon and exit without waiting for Regmon to terminate. When you execute this command, the instance of Regmon that executes will survive logoff and reappear on the desktop when you log back on, having captured the registry activity of both actions.

Another way to monitor registry activity during the logon, logoff, boot, or shut down process is to use the Regmon log boot feature, which you can enable by selecting Log Boot in the Options menu. The next time you boot the system, the Regmon device driver logs registry activity from early in the boot to the \Windows\Regmon.log. It will continue logging to that file until disk space runs out, the system shuts down, or you run Regmon. A log file storing a registry trace of startup, logon, logoff, and shut down on a Windows XP system will typically be between 50 and 150 MB in size.

Registry Internals

In this section, you'll find out how the configuration manager—the executive subsystem that implements the registry—organizes the registry's on-disk files. We'll examine how the configuration manager manages the registry as applications and other operating system components read and change registry keys and values. We'll also discuss the mechanisms by which the configuration manager tries to ensure that the registry is always in a recoverable state, even if the system crashes while the registry is being modified.

Hives

On disk, the registry isn't simply one large file but rather a set of discrete files called *hives*. Each hive contains a registry tree, which has a key that serves as the root or starting point of the tree. Subkeys and their values reside beneath the root. You might think that the root keys displayed by the Registry Editor tools correlate to the root keys in the hives, but such is not the case. Table 4-5 lists registry hives and their on-disk filenames. The pathnames of all hives except for user profiles are coded into the configuration manager. As the configuration manager loads hives, including system profiles, it notes each hive's path in the values under the HKLM\SYSTEM\CurrentControlSet\Control\hivelist subkey, removing the path if the hive is unloaded. (User profiles are unloaded when not referenced.) It creates the root keys, linking these hives together to build the registry structure you're familiar with and that the Registry Editor displays.

Table 4-5 On-Disk Files Corresponding to Paths in the Registry

Hive Registry Path	Hive File Path
HKEY_LOCAL_MACHINE\SYSTEM	\Windows\System32\Config\System
HKEY_LOCAL_MACHINE\SAM	\Windows\System32\Config\Sam
HKEY_LOCAL_MACHINE\SECURITY	\Windows\System32\Config\Security
HKEY_LOCAL_MACHINE\SOFTWARE	\Windows\System32\Config\Software
HKEY_LOCAL_MACHINE\HARDWARE	Volatile hive
HKEY_LOCAL_MACHINE\SYSTEM\Clone	Volatile hive (on Windows 2000 only)
HKEY_USERS\<security ID of username>	\Documents and Settings\<username>\Ntuser.dat
HKEY_USERS\<security ID of username>_Classes	\Documents and Settings\<username>\Local Settings\Application Data\Microsoft\Windows\Usrclass.dat
HKEY_USERS\.DEFAULT	\Windows\System32\Config\Default

You'll notice that some of the hives listed in Table 4-5 are volatile and don't have associated files. The system creates and manages these hives entirely in memory; the hives are therefore temporary. The system creates volatile hives every time it boots. An example of a volatile hive is the HKLM\HARDWARE hive, which stores information about physical devices and the devices' assigned resources. Resource assignment and hardware detection occur every time the system boots, so not storing this data on disk is logical.

EXPERIMENT: Manually Loading and Unloading Hives

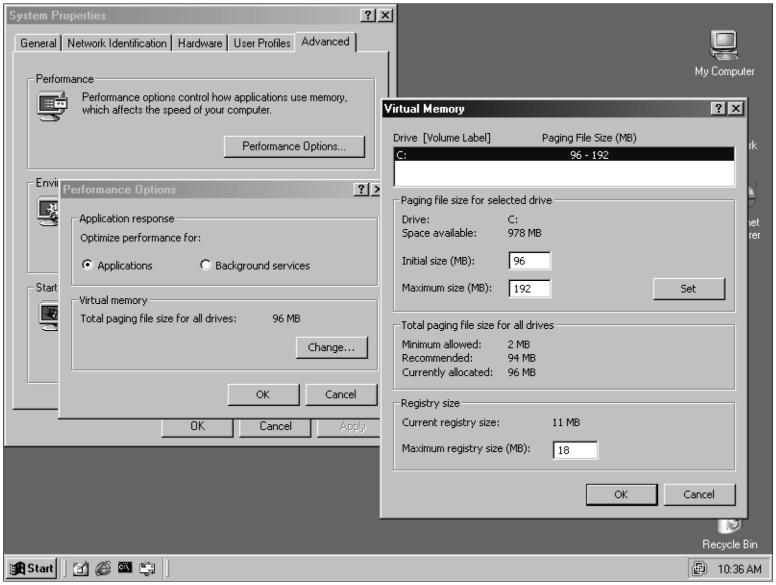
Regedt32 on Windows 2000 and Regedit on Windows XP and Windows Server 2003 have the ability to load hives that you can access through its File menu. This capability can be useful in troubleshooting scenarios where you want to view or edit a hive from an unbootable system or a backup medium. In this experiment, you'll use Regedt32 (if you're running Windows 2000) or Regedit (if you're running Windows XP and Windows Server 2003) to load a version of the HKLM\SYSTEM hive that Windows Setup creates and stores in \Windows\Repair during the install process.

1. Hives can be loaded only underneath HKLM or HKU, so open Regedit or Regedt32, select HKLM, and choose Load Hive from the Regedit File menu or the Regedt32 Registry menu.
2. Navigate to the \Windows\Repair directory in the Load Hive dialog box, select System.bak, and open it. When prompted, enter **Test** as the name of the key under which it will load.
3. Open the newly created HKLM\Test key, and explore the contents of the hive.
4. Open HKLM\System\CurrentControlSet\Control\Hivelist, and locate the entry \Registry\Machine\Test, which demonstrates how the configuration manager lists loaded hives in the HiveList key.
5. Select HKLM\Test, and choose Unload Hive from the Regedit File menu or the Regedt32 Registry menu to unload the hive.

Hive Size Limits

In some cases, hive sizes are limited. For example, Windows places a limit on the size of the HKLM\SYSTEM hive. It does so because Ntldr reads the entire HKLM\SYSTEM hive into physical memory near the start of the boot process when virtual memory paging is not enabled. Ntldr also loads Ntoskrnl and boot device drivers into physical memory, so it must constrain the amount of physical memory assigned to HKLM\SYSTEM. (See Chapter 6 for more information on the role Ntldr plays during the startup process.) On Windows 2000, Ntldr places a fixed upper limit on its size of 12 MB, but on Windows XP and Windows Server 2003 it is more flexible, allowing the hive to be up to 200 MB or one fourth the amount of physical memory on the system, whichever is lower.

On Windows 2000, there is also a limit on the combined sizes of all loaded registry hives. Windows 2000 uses a type of kernel memory called *paged pool* to hold registry hives in memory, and therefore, the total amount of loaded registry data is constrained by the amount of paged pool that's available. The amount of paged pool the memory manager creates during its initialization is based on a number of factors, such as the amount of physical memory on the system. On a system where the memory manager creates the largest amount of paged pool possible, the registry size limit is 376 MB. Because a system will not operate smoothly if there is not enough paged pool left over for other uses, Windows 2000 won't let registry data grow to more than 80 percent of paged pool and also honors a user-configurable registry quota if it's less than that amount. Click the Change button in the Virtual Memory section of the Performance Options dialog box that you reach on the Advanced page of the Control Panel's System applet to view or modify the registry quota setting, which you can see in Figure 4-4.



Performance options control how applications use memory, which affects the speed of your computer.

Optimize performance for:

● Applications ○ Background services

Total paging file size for all drives: 96 MB

C: 96 - 192

Drive: C:
Space available: 978 MB
Initial size (MB): 96
Maximum size (MB): 192

Minimum allowed: 2 MB
Recommended: 94 MB
Currently allocated: 96 MB

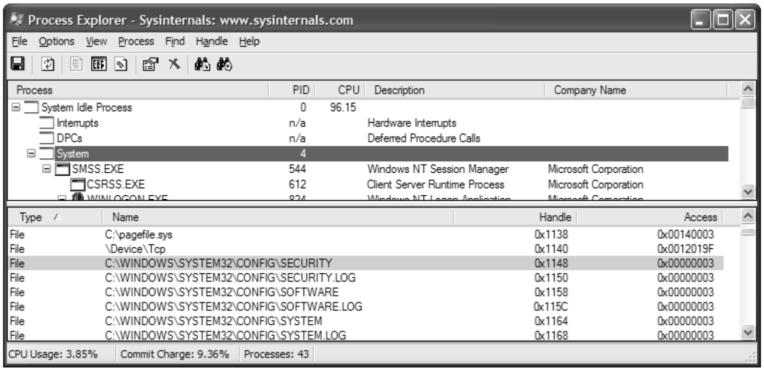
Current registry size: 11 MB
Maximum registry size (MB): 18

Figure 4-4 Windows 2000 registry quota setting

The upper limit on the total size of loaded registry hives can create a limit on the number of concurrently logged-in users on a Windows 2000 system running Terminal Services, because each user's profile contributes to the loaded hive size. On Windows XP and Windows Server 2003, the configuration manager therefore does not use paged pool and instead relies on the memory manager's memory-mapping functions to map into system memory only the portions of registry hives that it's accessing at any given point in time. There is no registry quota on Windows XP or Windows Server 2003, and the total size of loaded hives does not constrain the scalability of Terminal Services.

EXPERIMENT: Looking at Hive Handles

The configuration manager opens hives by using the kernel handle table (described in Chapter 3) so that it can access hives from any process context. Using the kernel handle table is an efficient alternative to approaches that involve using drivers or executive components to access from the system process only handles that must be protected from user processes. You can use the Process Explorer utility, available from *www.sysinternals.com*, to see the hive handles. On Windows 2000, the object manager reports kernel handle table handles as being opened in the System Idle process, and on Windows XP and Windows Server 2003 it reports them as being opened in the System process. Select the appropriate process for the Windows version that you are running, and select Handles from the Lower Pane View menu entry in the View menu. Sort by handle type, and scroll until you see the hive files, as shown in the following graphic.



Process	PID	CPU	Description	Company Name
System Idle Process	0	96.15		
Interrupts	n/a		Hardware Interrupts	
DPCs	n/a		Deferred Procedure Calls	
System	4			
SMSS.EXE	544		Windows NT Session Manager	Microsoft Corporation
CSRSS.EXE	612		Client Server Runtime Process	Microsoft Corporation
WINLOGON.EXE	824		Windows NT Logon Application	Microsoft Corporation

Type	Name	Handle	Access
File	C:\pagefile.sys	0x1138	0x00140003
File	\Device\Tcp	0x1140	0x0012019F
File	C:\WINDOWS\SYSTEM32\CONFIG\SECURITY	0x1148	0x00000003
File	C:\WINDOWS\SYSTEM32\CONFIG\SECURITY.LOG	0x1150	0x00000003
File	C:\WINDOWS\SYSTEM32\CONFIG\SOFTWARE	0x1158	0x00000003
File	C:\WINDOWS\SYSTEM32\CONFIG\SOFTWARE.LOG	0x115C	0x00000003
File	C:\WINDOWS\SYSTEM32\CONFIG\SYSTEM	0x1164	0x00000003
File	C:\WINDOWS\SYSTEM32\CONFIG\SYSTEM.LOG	0x1168	0x00000003

CPU Usage: 3.85% Commit Charge: 9.36% Processes: 43

A special type of key known as a *symbolic link* makes it possible for the configuration manager to link hives to organize the registry. A symbolic link is a key that redirects the configuration manager to another key. Thus, the key HKLM\SAM is a symbolic link to the key at the root of the SAM hive.

Hive Structure

The configuration manager logically divides a hive into allocation units called *blocks* in much the same way that a file system divides a disk into clusters. By definition, the registry block size is 4096 bytes (4 KB). When new data expands a hive, the hive always expands in block-granular increments. The first block of a hive is the *base block*. The base block includes global information about the hive, including a signature—*regf*—that identifies the file as a hive, updated sequence numbers, a time stamp that shows the last time a write operation was initiated on the hive, the hive format version number, a checksum, and the hive file's internal filename (for example, \Device\HarddiskVolume1\WINDOWS\SYSTEM32\CONFIG\SAM). We'll clarify the significance of the updated sequence numbers and time stamp when we describe how data is written to a hive file. The hive format version number specifies the data format within the hive. The configuration manager uses hive format version 1.3 on Windows 2000. On Windows XP and Windows Server 2003, it uses format version 1.3 for all hives except for System and Software for roaming profile compatibility with Windows 2000. For System and Software hives, it uses version 1.5 because of the new format's optimizations for large values and searching.

Windows organizes the registry data that a hive stores in containers called *cells*. A cell can hold a key, a value, a security descriptor, a list of subkeys, or a list of key values. A field at the beginning of a cell's data describes the data's type. Table 4-6 describes each cell data type in detail. A cell's header is a field that specifies the cell's size. When a cell joins a hive and the hive must expand to contain the cell, the system creates an allocation unit called a *bin*. A bin is the size of the new cell rounded up to the next block boundary. The system considers any space between the end of the cell and the end of the bin to be free space that it can allocate to other cells. Bins also have headers that contain a signature, *hbin*, and a field that records the offset into the hive file of the bin and the bin's size.

Table 4-6 **Cell Data Types**

Data Type	Description
Key cell	A cell that contains a registry key, also called a *key node*. A key cell contains a signature (*kn* for a key, *kl* for a symbolic link), the time stamp of the most recent update to the key, the cell index of the key's parent key cell, the cell index of the subkey-list cell that identifies the key's subkeys, a cell index for the key's security descriptor cell, a cell index for a string key that specifies the class name of the key, and the name of the key (for example, CurrentControlSet).
Value cell	A cell that contains information about a key's value. This cell includes a signature (*kv*), the value's type (for example, REG_ DWORD or REG_BINARY), and the value's name (for example, Boot-Execute). A value cell also contains the cell index of the cell that contains the value's data.
Subkey-list cell	A cell composed of a list of cell indexes for key cells that are all subkeys of a common parent key.
Value-list cell	A cell composed of a list of cell indexes for value cells that are all values of a common parent key.
Security-descriptor cell	A cell that contains a security descriptor. Security-descriptor cells include a signature (ks) at the head of the cell and a reference count that records the number of key nodes that share the security descriptor. Multiple key cells can share security-descriptor cells.

By using bins, instead of cells, to track active parts of the registry, Windows minimizes some management chores. For example, the system usually allocates and deallocates bins less frequently than it does cells, which lets the configuration manager manage memory more efficiently. When the configuration manager reads a registry hive into memory, it can choose to read only bins that contain cells (that is, active bins) and to ignore empty bins. When the system adds and deletes cells in a hive, the hive can contain empty bins interspersed with active bins. This situation is similar to disk fragmentation, which occurs when the system creates and deletes files on the disk. When a bin becomes empty, the configuration manager joins to the empty bin any adjacent empty bins to form as large a contiguous empty bin as possible. The configuration manager also joins adjacent deleted cells to form larger free cells. (The configuration manager shrinks a hive only when bins at the end of the hive become free. You can compact the registry by backing it up and restoring it using the Windows *RegSaveKey* and *RegReplaceKey* functions, which are used by the Windows Backup utility.)

The links that create the structure of a hive are called *cell indexes*. A cell index is the offset of a cell into the hive file. Thus, a cell index is like a pointer from one cell to another cell that the configuration manager interprets relative to the start of a hive. For example, as you saw in Table 4-6, a cell that describes a key contains a field specifying the cell index of its parent key; a cell index for a subkey specifies the cell that describes the subkeys that are subordinate to the specified subkey. A subkey-list cell contains a list of cell indexes that refer to the subkey's key cells. Therefore, if you want to locate, for example, the key cell of subkey A, whose parent is key B, you must first locate the cell containing key B's subkey list using the subkey-list cell index in key B's cell. Then you locate each of key B's subkey cells by using the list of cell indexes in the subkey-list cell. For each subkey cell, you check to see whether the subkey's name, which a key cell stores, matches the one you want to locate, in this case, subkey A.

The distinction between cells, bins, and blocks can be confusing, so let's look at an example of a simple registry hive layout to help clarify the differences. The sample registry hive file in Figure 4-5 contains a base block and two bins. The first bin is empty, and the second bin contains several cells. Logically, the hive has only two keys: the root key Root, and a subkey of Root, Sub Key. Root has two values, *Val 1* and *Val 2*. A subkey-list cell locates the root key's subkey, and a value-list cell locates the root key's values. The free spaces in the second bin are empty cells. Figure 4-5 doesn't show the security cells for the two keys, which would be present in a hive.

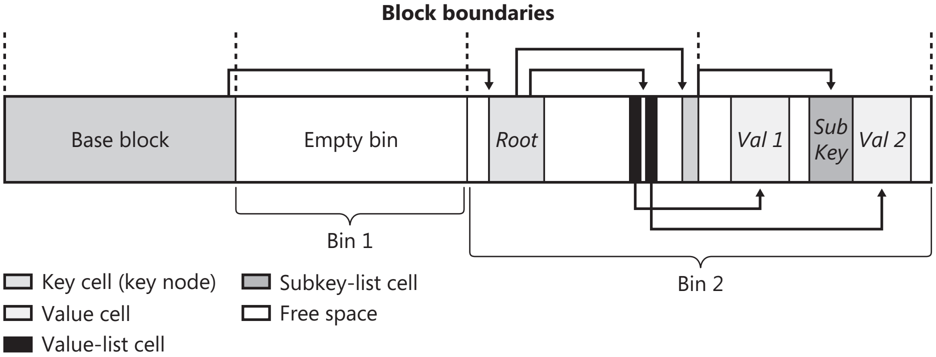
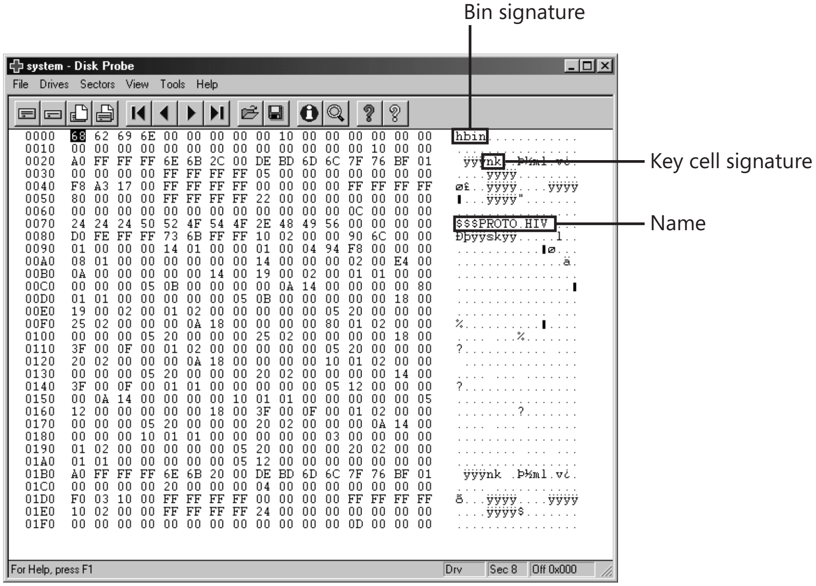


Figure 4-5 Internal structure of a registry hive

Figure 4-6 shows an example of the Disk Probe utility (Dskprobe.exe) examining the first bin in a SYSTEM hive. Notice the bin's signature, *hbin*, at the top right side of the image. Look beneath the bin signature and you'll see the signature *nk*. This signature is the signature of a key cell (*kn*). The signature displays backward because of the way x86 computers store data. The cell is the SYSTEM hive's root cell, which the configuration manager has named internally $$$PROTO.HIV, as specified by the name that follows the *nk* signature.



```
0000  68 62 69 6E 00 00 00 00 00 10 00 00 00 00 00 00   hbin............
0010  00 00 00 00 00 00 00 00 00 00 00 00 00 10 00 00   ................
0020  A0 FF FF FF 6E 6B 2C 00 DE BD 6D 6C 7F 76 BF 01    ÿÿÿnk,.Þ½ml.v¿.
0030  00 00 00 00 FF FF FF FF 05 00 00 00 00 00 00 00   ....ÿÿÿÿ........
0040  F8 A3 17 00 FF FF FF FF 00 00 00 00 FF FF FF FF   ø£..ÿÿÿÿ....ÿÿÿÿ
0050  80 00 00 00 FF FF FF FF 22 00 00 00 00 00 00 00   ▌...ÿÿÿÿ".......
0060  00 00 00 00 00 00 00 00 00 00 00 00 0C 00 00 00   ................
0070  24 24 24 50 52 4F 54 4F 2E 48 49 56 00 00 00 00   $$$PROTO.HIV....
0080  D0 FE FF FF 73 6B FF FF 10 02 00 00 90 6C 00 00   Ðþÿÿskÿÿ.....l..
0090  01 00 00 00 14 01 00 00 01 00 04 94 F8 00 00 00   ...........▌ø...
00A0  08 01 00 00 00 00 00 00 14 00 00 00 02 00 E4 00   ..............ä.
00B0  0A 00 00 00 00 00 14 00 19 00 02 00 01 01 00 00   ................
00C0  00 00 00 05 0B 00 00 00 00 0A 14 00 00 00 00 80   ...............▌
00D0  01 01 00 00 00 00 00 05 0B 00 00 00 00 00 18 00   ................
00E0  19 00 02 00 01 02 00 00 00 00 00 05 20 00 00 00   ................
00F0  25 02 00 00 00 0A 18 00 00 00 00 80 01 02 00 00   %..........▌....
0100  00 00 00 05 20 00 00 00 25 02 00 00 00 00 18 00   .........%......
0110  3F 00 0F 00 01 02 00 00 00 00 00 05 20 00 00 00   ?...............
0120  20 02 00 00 00 0A 18 00 00 00 00 10 01 02 00 00    ...............
0130  00 00 00 05 20 00 00 00 20 02 00 00 00 00 14 00   ................
0140  3F 00 0F 00 01 01 00 00 00 00 00 05 12 00 00 00   ?...............
0150  00 0A 14 00 00 00 00 10 01 01 00 00 00 00 00 05   ................
0160  12 00 00 00 00 00 18 00 3F 00 0F 00 01 02 00 00   ........?.......
0170  00 00 00 05 20 00 00 00 20 02 00 00 00 0A 14 00   ................
0180  00 00 00 10 01 01 00 00 00 00 00 03 00 00 00 00   ................
0190  01 02 00 00 00 00 00 05 20 00 00 00 20 02 00 00   ................
01A0  01 01 00 00 00 00 00 05 12 00 00 00 00 00 00 00   ................
01B0  A0 FF FF FF 6E 6B 20 00 DE BD 6D 6C 7F 76 BF 01    ÿÿÿnk .Þ½ml.v¿.
01C0  00 00 00 00 20 00 00 00 04 00 00 00 00 00 00 00   ................
01D0  F0 03 10 00 FF FF FF FF 00 00 00 00 FF FF FF FF   ð...ÿÿÿÿ....ÿÿÿÿ
01E0  10 02 00 00 FF FF FF FF 24 00 00 00 00 00 00 00   ....ÿÿÿÿ$.......
01F0  00 00 00 00 00 00 00 00 00 00 00 00 0D 00 00 00   ................
```

Figure 4-6 Binary contents of first bin in the SYSTEM hive

To optimize searches for both values and subkeys, the configuration manager sorts subkey-list cells alphabetically. The configuration manager can then perform a binary search when it looks for a subkey within a list of subkeys. The configuration manager examines the subkey in the middle of the list, and if the name of the subkey the configuration manager is looking for is alphabetically before the name of the middle subkey, the configuration manager knows that the subkey is in the first half of the subkey list; otherwise, the subkey is in the second half of the subkey list. This splitting process continues until the configuration manager locates the subkey or finds no match. Value-list cells aren't sorted, however, so new values are always added to the end of the list.

Cell Maps

The configuration manager doesn't access a hive's image on disk every time a registry access occurs. Windows 2000 keeps a version of every hive in the kernel's address space. When a hive initializes, the configuration manager determines the size of the hive file, allocates enough memory from the kernel's paged pool to store it, and reads the hive file into memory. (For more information on paged pool, see Chapter 7.) Because all loaded registry hives are read into paged pool, that registry data is typically the largest consumer of the paged pool in Windows 2000. (To check paged pool allocation, use the Poolmon utility, described in the "Experiment: Monitoring Pool Usage" sidebar in Chapter 7.)

In Windows XP and Windows Server 2003, the configuration manager maps portions of a hive into memory as it needs to access them. It uses the cache manager's file mapping functions to map in 16-KB views into the hive files. (See Chapter 10 for more information on the cache manager.) To prevent hive mapping from consuming all the cache manager's address range, the configuration manager tries to keep no more than 256 views of a hive mapped at any given point in time by unmapping least-recently used (LRU) views when it reaches that limit. The configuration manager still uses the paged pool to store various data structures (including the LRU list of views), but its use of the paged pool is a fraction of what it is in Windows 2000.

Note On Windows XP and Windows Server 2003, the configuration manager will store a block in the paged pool instead of mapping it if the block exceeds 256 KB in size.

If hives never grew, the configuration manager could perform all its registry management on the in-memory version of a hive as if the hive were a file. Given a cell index, the configuration manager could calculate the location in memory of a cell simply by adding the cell index, which is a hive file offset, to the base of the in-memory hive image. Early in the system boot, this process is exactly what Ntldr does with the SYSTEM hive: Ntldr reads the entire SYSTEM hive into memory as a read-only hive and adds the cell indexes to the base of the in-memory hive image to locate cells. Unfortunately, hives grow as they take on new keys and values, which means the system must allocate paged pool memory to store the new bins that contain added keys and values. Thus, the paged pool that keeps the registry data in memory isn't necessarily contiguous.

EXPERIMENT: Viewing Hive Paged Pool Usage

There are no administrative-level tools that show you the amount of paged pool that registry hives, including user profiles, are consuming on Windows 2000. However, the *!reg dumppool* kernel debugger command shows you not only how many pages of the paged pool each loaded hive consumes but also how many of the pages store volatile and nonvolatile data. The command prints the total hive memory usage at the end of the output. (The command shows only the last 32 characters of a hive's name.)

```
kd> !reg dumppool

dumping hive at e20d66a8 (a\Microsoft\Windows\UsrClass.dat)
  Stable Length = 1000
  1/1 pages present
  Volatile Length = 0

dumping hive at e215ee88 (ettings\Administrator\ntuser.dat)
  Stable Length = f2000
  242/242 pages present
  Volatile Length = 2000
  2/2 pages present

dumping hive at e13fa188 (\SystemRoot\System32\Config\SAM)
  Stable Length = 5000
  5/5 pages present
  Volatile Length = 0

...
```

EXPERIMENT: Viewing Hive Memory Usage

In Windows XP and Windows Server 2003, you can view statistics on hive memory usage, including its stable (on-disk) size and nonvolatile size, the number of active views, and the number of views that are locked into memory, using the *!reg hivelist* command (note that the line output wraps):

```
--------------------------------------------------------------------------------------
-----------------------
| HiveAddr |Stable Length|Stable Map|Volatile Length|Volatile Map|MappedViews|PinnedVi
ews|U(Cnt)| BaseBlock | FileName
--------------------------------------------------------------------------------------
-----------------------
| e22f8b68 |       5000  | e22f8bc4 |       1000    |  e22f8ca0  |        2  |
0  |     0| e2353000  | \Microsoft
\Windows\UsrClass.dat
| e28c3008 |     3fe000  | e1e84000 |       c000    |  e28c3140  |      116  |
0  |     0| e1e48000  | ttings\Adm
inistrator\ntuser.dat
| e23ec008 |       1000  | e23ec064 |          0    |  00000000  |        1  |
0  |     0| e23ee000  | \Microsoft
\Windows\UsrClass.dat
| e23ed760 |      37000  | e23ed7bc |       1000    |  e23ed898  |       14  |
0  |     0| e23ef000  | ettings\Lo
calService\ntuser.dat
...
```

In the preceding output, the Administrator account's profile hive (the full path of which, \Documents and Settings\Administrator\ntuser.dat, is truncated in the output) has 116 mapped views and is approximately 4 MB in size (0x3f000 in decimal). The *!reg viewlist* command will dump the mapped views of the hive you specify. Here's the output of that command when executed for the UsrClass.dat hive that was printed as the first hive of the *!reg hivelist* command's output:

```
kd> !reg viewlist e22f8b68

   0  Pinned Views ; PinViewListHead = e22f8da0 e22f8da0

   2  Mapped Views ; LRUViewListHead = e1cf4448 e1c5d440
--------------------------------------------------------------------------------------
------------------------
| ViewAddr |FileOffset|   Size   |ViewAddress|   Bcb    |    LRUViewList     |    PinV
iewList     | UseCount |
--------------------------------------------------------------------------------------
------------------------
| e1cf4448 |        0 |     4000 | c9a40000  | 8a4bb0e9 | e1c5d440  e22f8d98 | e1cf445
0  e1cf4450 |        0 |
| e1c5d440 |     4000 |     2000 | c9a44000  | 8a4bb0e9 | e22f8d98  e1cf4448 | e1c5d44
8  e1c5d448 |        0 |
--------------------------------------------------------------------------------------
------------------------
```

The output shows the addresses of the two views that the *hivelist* command reported for the hive in the ViewAddress column. Using the debugger's *db* command to dump the contents of memory at the address of the first view reveals that it maps the base block of the hive, recognizable with its *regf* signature:

```
kd> db c9a40000
c9a40000  72 65 67 66 d5 01 00 00-d5 01 00 00 cc 20 43 c7  regf......... C.
c9a40010  3d 40 c4 01 01 00 00 00-03 00 00 00 00 00 00 00  =@..............
c9a40020  01 00 00 00 20 00 00 00-00 50 00 00 01 00 00 00  .... ....P......
c9a40030  5c 00 4d 00 69 00 63 00-72 00 6f 00 73 00 6f 00  \.M.i.c.r.o.s.o.
c9a40040  66 00 74 00 5c 00 57 00-69 00 6e 00 64 00 6f 00  f.t.\.W.i.n.d.o.
c9a40050  77 00 73 00 5c 00 55 00-73 00 72 00 43 00 6c 00  w.s.\.U.s.r.C.l.
c9a40060  61 00 73 00 73 00 2e 00-64 00 61 00 74 00 00 00  a.s.s...d.a.t...
c9a40070  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
```

To deal with noncontiguous memory addresses referencing hive data in memory, the configuration manager adopts a strategy similar to what the Windows memory manager uses to map virtual memory addresses to physical memory addresses. The configuration manager employs a two-level scheme, which Figure 4-7 illustrates, that takes as input a cell index (that is, a hive file offset) and returns as output both the address in memory of the block the cell index resides in and the address in memory of the block the cell resides in. Remember that a bin can contain one or more blocks and that hives grow in bins, so Windows always represents a bin with a contiguous region of memory. Therefore, all blocks within a bin occur within the same cache manager view (in Windows XP and Windows Server 2003) or portion of a paged pool (in Windows 2000).

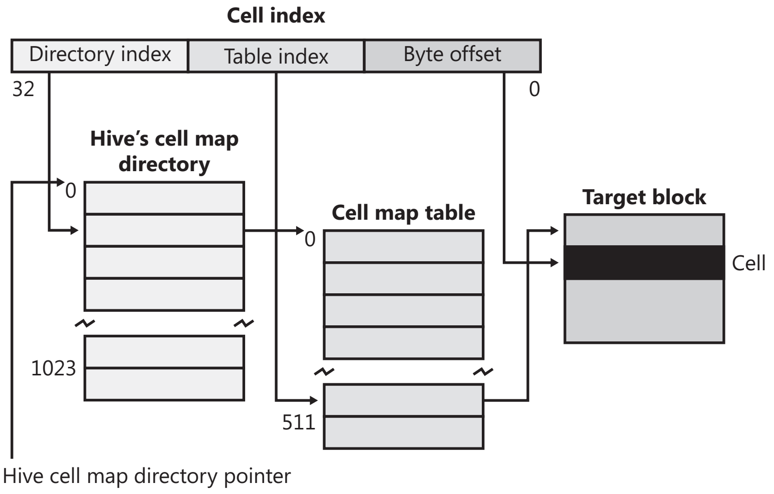


Figure 4-7 Structure of a cell index

To implement the mapping, the configuration manager divides a cell index logically into fields, in the same way that the memory manager divides a virtual address into fields. Windows interprets a cell index's first field as an index into a hive's cell map directory. The cell map directory contains 1024 entries, each of which refers to a cell map table that contains 512 map entries. An entry in this cell map table is specified by the second field in the cell index. That entry locates the bin and block memory addresses of the cell. In Windows XP and Windows Server 2003, not all bins are necessarily mapped into memory, and if a cell lookup yields an address of 0, the configuration manager maps the bin into memory, unmapping another on the mapping LRU list it maintains, if necessary.

In the final step of the translation process, the configuration manager interprets the last field of the cell index as an offset into the identified block to precisely locate a cell in memory. When a hive initializes, the configuration manager dynamically creates the mapping tables, designating a map entry for each block in the hive, and it adds and deletes tables from the cell directory as the changing size of the hive requires.

The Registry Namespace and Operation

The configuration manager defines a *key object* object type to integrate the registry's namespace with the kernel's general namespace. The configuration manager inserts a key object named *Registry* into the root of the Windows namespace, which serves as the entry point to the registry. Regedit shows key names in the form HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet, but the Windows subsystem translates such names into their object namespace form (for example, \Registry\Machine\System\CurrentControlSet). When the Windows object manager parses this name, it encounters the key object by the name of Registry first and hands the rest of the name to the configuration manager. The configuration manager takes over the name parsing, looking through its internal hive tree to find the desired key or value. Before we describe the flow of control for a typical registry operation, we need to discuss key objects and *key control blocks*. Whenever an application opens or creates a registry key, the object manager gives a handle with which to reference the key to the application. The handle corresponds to a key object that the configuration manager allocates with the help of the object manager. By using the object manager's object support, the configuration manager takes advantage of the security and reference-counting functionality that the object manager provides.

For each open registry key, the configuration manager also allocates a key control block. A key control block stores the full pathname of the key, includes the cell index of the key node that the control block refers to, and contains a flag that notes whether the configuration manager needs to delete the key cell that the key control block refers to when the last handle for the key closes. Windows places all key control blocks into a hash table to enable quick searches for existing key control blocks by name. A key object points to its corresponding key control block, so if two applications open the same registry key, each will receive a key object, and both key objects will point to a common key control block.

When an application opens an existing registry key, the flow of control starts with the application specifying the name of the key in a registry API that invokes the object manager's name-parsing routine. The object manager, upon encountering the configuration manager's registry key object in the namespace, hands the pathname to the configuration manager. The configuration manager uses the in-memory hive data structures to search through keys and subkeys to find the specified key. If the configuration manager finds the key cell, the configuration manager searches the key control block tree to determine whether the key is open (by the same or another application). The search routine is optimized to always start from the closest ancestor with a key control block already opened. For example, if an application opens \Registry\Machine\Key1\Subkey2, and \Registry\Machine is already opened, the parse routine uses the key control block of \Registry\Machine as a starting point. If the key is open, the configuration manager increments the existing key control block's reference count. If the key isn't open, the configuration manager allocates a new key control block and inserts it into the tree. Then the configuration manager allocates a key object, points the key object at the key control block, and returns control to the object manager, which returns a handle to the application.

When an application creates a new registry key, the configuration manager first finds the key cell for the new key's parent. The configuration manager then searches the list of free cells for the hive in which the new key will reside to determine whether cells exist that are large enough to hold the new key cell. If there aren't any free cells large enough, the configuration manager allocates a new bin and uses it for the cell, placing any space at the end of the bin on the free cell list. The new key cell fills with pertinent information—including the key's name—and the configuration manager adds the key cell to the subkey list of the parent key's subkey-list cell. Finally, the system stores the cell index of the parent cell in the new subkey's key cell.

The configuration manager uses a key control block's reference count to determine when to delete the key control block. When all the handles that refer to a key in a key control block close, the reference count becomes 0, which denotes that the key control block is no longer necessary. If an application that calls an API to delete the key sets the delete flag, the configuration manager can delete the associated key from the key's hive because it knows that no application is keeping the key open.

EXPERIMENT: Viewing Key Control Blocks

You can use the kernel debugger to list all the key control blocks allocated on a system with the command *!reg openkeys*. Alternatively, if you want to view the key control block for a particular open key, use *!reg findkcb*:

```
kd> !reg findkcb \registry\machine\software\microsoft

Found KCB = e1034d40 :: \REGISTRY\MACHINE\SOFTWARE\MICROSOFT
```

You can then examine a reported key control block with the *!reg kcb* command:

```
kd> !reg kcb e1034d40

Key              : \REGISTRY\MACHINE\SOFTWARE\MICROSOFT
RefCount         : 1f
Flags            : CompressedName, Stable
```

```
ExtFlags         :
Parent           : 0xe1997368
KeyHive          : 0xe1c8a768
KeyCell          : 0x64e598 [cell index]
TotalLevels      : 4
DelayedCloseIndex: 2048
MaxNameLen       : 0x3c
MaxValueNameLen  : 0x0
MaxValueDataLen  : 0x0
LastWriteTime    : 0x 1c42501:0x7eb6d470
KeyBodyListHead  : 0xe1034d70 0xe1034d70
SubKeyCount      : 137
ValueCache.Count : 0
KCBLock          : 0xe1034d40
KeyLock          : 0xe1034d40
```

The Flags field indicates that the name is stored in compressed form and the SubKeyCount field shows that the key has 137 subkeys.

Stable Storage

To make sure that a nonvolatile registry hive (one with an on-disk file) is always in a recoverable state, the configuration manager uses *log hives*. Each nonvolatile hive has an associated log hive, which is a hidden file with the same base name as the hive and a *.log* extension. For example, if you look in your \Windows\System32\Config directory (and you have the Show Hidden Files And Folders folder option selected), you'll see System.log, Sam.log, and other .log files. When a hive initializes, the configuration manager allocates a bit array in which each bit represents a 512-byte portion, or *sector*, of the hive. This array is called the *dirty sector array* because an *on* bit in the array means that the system has modified the corresponding sector in the hive in memory and must write the sector back to the hive file. (An *off* bit means that the corresponding sector is up to date with the in-memory hive's contents.)

When the creation of a new key or value or the modification of an existing key or value takes place, the configuration manager notes the sectors of the hive that change in the hive's dirty sector array. Then the configuration manager schedules a lazy write operation, or a *hive sync*. The hive lazy writer system thread wakes up 5 seconds after the request to synchronize the hive and writes dirty hive sectors for all hives from memory to the hive files on disk. Thus, the system flushes, at the same time, all the registry modifications that take place between the time a hive sync is requested and the time the hive sync occurs. When a hive sync takes place, the next hive sync will occur no sooner than 5 seconds later.

Note On Windows Server 2003, you can change the default 5-second delay the hive lazy writer thread uses up by setting the registry value HKLM\System\CurrentControlSet\Session Manager\Configuration Manager\RegistryLazyFlushInterval.

If the lazy writer simply wrote all a hive's dirty sectors to the hive file and the system crashed in midoperation, the hive file would be in an inconsistent (corrupted) and unrecoverable state. To prevent such an occurrence, the lazy writer first dumps the hive's dirty sector array and all the dirty sectors to the hive's log file, increasing the log file's size if necessary. The lazy writer then updates a sequence number in the hive's base block and writes the dirty sectors to the hive. When the lazy writer is finished, it updates a second sequence number in the base block. Thus, if the system crashes during the write operations to the hive, at the next reboot the configuration manager will notice that the two sequence numbers in the hive's base block don't match. The configuration manager can update the hive with the dirty sectors in the hive's log file to roll the hive forward. The hive is then up to date and consistent.

To further protect the integrity of the crucial SYSTEM hive in Windows 2000, the configuration manager maintains a mirror of the SYSTEM hive on disk. If you look at the nonhidden files in a Windows 2000 \Windows\System32\Config directory, you'll see System.alt. System.alt is the *alternate hive*. Whenever a hive sync flushes dirty sectors to the SYSTEM hive, the hive sync also updates the System.alt hive. If the configuration manager detects that the SYSTEM hive is corrupt when the system boots, the configuration manager attempts to load the hive's alternate. If that hive is usable, it then uses that alternate to update the original SYSTEM hive.

Windows XP and Windows Server 2003 do not maintain a System.alt hive because NTLDR on those versions of Windows knows how to process the System.log file to bring up to date a System hive that's become inconsistent during a shut down or crash. Windows Server 2003 has other enhancements for tolerating corruption of the registry. Prior to Windows Server 2003, the configuration manager crashes the system if it reads a base block, bin, or cell that contains data that fails basic consistency checks. The configuration manager in Windows Server 2003 is more tolerant of such problems, and if the corruption isn't too severe, it will reinitialize corrupted data structures, possibly deleting subkeys in the process, and continue operation. If it has to resort to self-healing operation, it pops up a system error dialog box notifying the user.

Note When you look at the hidden files on \Windows\System32\Config, you'll also see a file named System.sav. System.Sav is the version of the SYSTEM hive that served as the initial copy of the System hive and is what Windows Setup copied from the install media.

Registry Optimizations

The configuration manager makes a few noteworthy performance optimizations. First, virtually every registry key has a security descriptor that protects access to the key. Storing a unique security-descriptor copy for every key in a hive would be highly inefficient, however, because the same security settings often apply to entire subtrees of the registry. When the system applies security to a key in Windows 2000, the configuration manager first checks the security descriptors associated with the key's parent key and then checks all the parent's subkeys. If any of those security descriptors match the security descriptor the system is applying to the key, the configuration manager simply shares the existing descriptors with the key, employing a reference count to track how many keys share the same descriptor. In Windows XP and Windows Server 2003, the configuration manager checks a pool of the unique security descriptors used within the same hive as the key to which new security is being applied, and it shares any existing descriptor for the key, ensuring that there is at most one copy of every unique security descriptor in a hive.

The configuration manager also optimizes the way it stores key and value names in a hive. Although the registry is fully Unicode-capable and specifies all names using the Unicode convention, if a name contains only ASCII characters, the configuration manager stores the name in ASCII form in the hive. When the configuration manager reads the name (such as when performing name lookups), it converts the name into Unicode form in memory. Storing the name in ASCII form can significantly reduce the size of a hive.

To minimize memory usage, key control blocks don't store full key registry pathnames. Instead, they reference only a key's name. For example, a key control block that refers to \Registry\System\Control would refer to the name *Control* rather than to the full path. A further memory optimization is that the configuration manager uses key name control blocks to store key names, and all key control blocks for keys with the same name share the same key name control block. To optimize performance, the configuration manager stores the key control block names in a hash table for quick lookups.

To provide fast access to key control blocks, the configuration manager stores frequently accessed key control blocks in the cache table, which is configured as a hash table. When the configuration manager needs to look up a key control block, it first checks the cache table. Finally, the configuration manager has another cache, the delayed close table, that stores key control blocks that applications close, so that an application can quickly reopen a key it has recently closed. The configuration manager removes the oldest key control blocks from the delayed close table as it adds the most recently closed blocks to the table.

Services

Almost every operating system has a mechanism to start processes at system startup time that provide services not tied to an interactive user. In Windows, such processes are called *services* or *Windows services*, because they rely on the Windows API to interact with the system. Services are similar to UNIX daemon processes and often implement the server side of client/server applications. An example of a Windows service might be a Web server because it must be running regardless of whether anyone is logged on to the computer and it must start running when the system starts so that an administrator doesn't have to remember, or even be present, to start it.

Windows services consist of three components: a service application, a service control program (SCP), and the service control manager (SCM). First, we'll describe service applications, service accounts, and the operations of the SCM. Then we'll explain how auto-start services are started during the system boot. We'll also cover the steps the SCM takes when a service fails during its startup and the way the SCM shuts down services.

Service Applications

Service applications, such as Web servers, consist of at least one executable that runs as a Windows service. A user wanting to start, stop, or configure a service uses an SCP. Although Windows supplies built-in SCPs that provide general start, stop, pause, and continue functionality, some service applications include their own SCP that allows administrators to specify configuration settings particular to the service they manage.

Service applications are simply Windows executables (GUI or console) with additional code to receive commands from the SCM as well as to communicate the application's status back to the SCM. Because most services don't have a user interface, they are built as console programs.

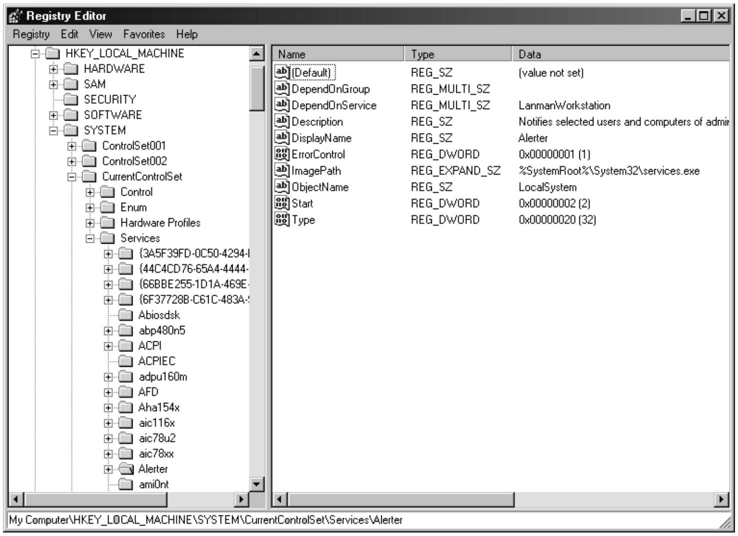
When you install an application that includes a service, the application's setup program must register the service with the system. To register the service, the setup program calls the Windows *CreateService* function, a services-related function implemented in Advapi32.dll (\Windows\System32\Advapi32.dll). Advapi32, the "Advanced API" DLL, implements all the client-side SCM APIs.

When a setup program registers a service by calling *CreateService*, a message is sent to the SCM on the machine where the service will reside. The SCM then creates a registry key for the service under HKLM\SYSTEM\CurrentControlSet\Services. The Services key is the nonvolatile representation of the SCM's database. The individual keys for each service define the path of the executable image that contains the service as well as parameters and configuration options.

After creating a service, an installation or management application can start the service via the *StartService* function. Because some service-based applications also must initialize during the boot process to function, it's not unusual for a setup program to register a service as an autostart service, ask the user to reboot the system to complete an installation, and let the SCM start the service as the system boots.

When a program calls *CreateService*, it must specify a number of parameters describing the service's characteristics. The characteristics include the service's type (whether it's a service that runs in its own process rather than a service that shares a process with other services), the location of the service's executable image file, an optional display name, an optional account name and password used to start the service in a particular account's security context, a start type that indicates whether the service starts automatically when the system boots or manually under the direction of an SCP, an error code that indicates how the system should react if the service detects an error when starting, and, if the service starts automatically, optional information that specifies when the service starts relative to other services.

The SCM stores each characteristic as a value in the service's registry key. Figure 4-8 shows an example of a service registry key.



Name	Type	Data
(Default)	REG_SZ	(value not set)
DependOnGroup	REG_MULTI_SZ	
DependOnService	REG_MULTI_SZ	LanmanWorkstation
Description	REG_SZ	Notifies selected users and computers of admir
DisplayName	REG_SZ	Alerter
ErrorControl	REG_DWORD	0x00000001 (1)
ImagePath	REG_EXPAND_SZ	%SystemRoot%\System32\services.exe
ObjectName	REG_SZ	LocalSystem
Start	REG_DWORD	0x00000002 (2)
Type	REG_DWORD	0x00000020 (32)

My Computer\HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\Alerter

Figure 4-8 Example of a service registry key

Table 4-7 lists all the service characteristics, many of which also apply to device drivers. (Not every characteristic applies to every type of service or device driver.) If a service needs to store configuration information that is private to the service, the convention is to create a subkey named Parameters under its service key and then store the configuration information in values under that subkey. The service then can retrieve the values by using standard registry functions.

> **Note** The SCM does not access a service's Parameters subkey until the service is deleted, at which time the SCM deletes the service's entire key, including subkeys like Parameters.

Table 4-7 Service and Driver Registry Parameters

Value Setting	Value Name	Value Setting Description
Start	SERVICE_BOOT_START (0)	Ntldr or Osloader preloads the driver so that it is in memory during the boot. These drivers are initialized just prior to SERVICE_ SYSTEM_START drivers.
	SERVICE_SYSTEM_START (1)	The driver loads and initializes during kernel initialization after SERVICE_ BOOT_START drivers have initialized.
	SERVICE_AUTO_START (2)	The SCM starts the driver or service after the SCM process, Services.exe, starts.
	SERVICE_DEMAND_START (3)	The SCM starts the driver or service on demand.

Table 4-7 **Service and Driver Registry Parameters**

Value Setting	Value Name	Value Setting Description
	SERVICE_DISABLED (4)	The driver or service doesn't load or initialize.
ErrorControl	SERVICE_ERROR_IGNORE (0)	Any error the driver or service returns is ignored and no warning is logged or displayed.
	SERVICE_ERROR_NORMAL (1)	If the driver or service reports an error, a warning displays.
	SERVICE_ERROR_SEVERE (2)	If the driver or service returns an error and last known good isn't being used, reboot into last known good; otherwise, continue the boot.
	SERVICE_ERROR_CRITICAL (3)	If the driver or service returns an error and last known good isn't being used, reboot into last known good; otherwise, stop the boot with a blue screen crash.
Type	SERVICE_KERNEL_DRIVER (1)	Device driver.
	SERVICE_FILE_SYSTEM_DRIVER (2)	Kernel-mode file system driver.
	SERVICE_ADAPTER (4)	Obsolete.
	SERVICE_RECOGNIZER_DRIVER (8)	File system recognizer driver.
	SERVICE_WIN32_OWN_PROCESS (16)	The service runs in a process that hosts only one service.
	SERVICE_WIN32_SHARE_PROCESS (32)	The service runs in a process that hosts multiple services.
	SERVICE_INTERACTIVE_PROCESS (256)	The service is allowed to display windows on the console and receive user input.
Group	Group name	The driver or service initializes when its group is initialized.
Tag	Tag number	The specified location in a group initialization order. This parameter doesn't apply to services.
ImagePath	Path to service or driver executable file	If *ImagePath* isn't specified, the I/O manager looks for drivers in \Windows\System32\Drivers and the SCM uses Windows functions that search for the image using the PATH environment variable.
DependOnGroup	Group name	The driver or service won't load unless a driver or service from the specified group loads.

Table 4-7 Service and Driver Registry Parameters

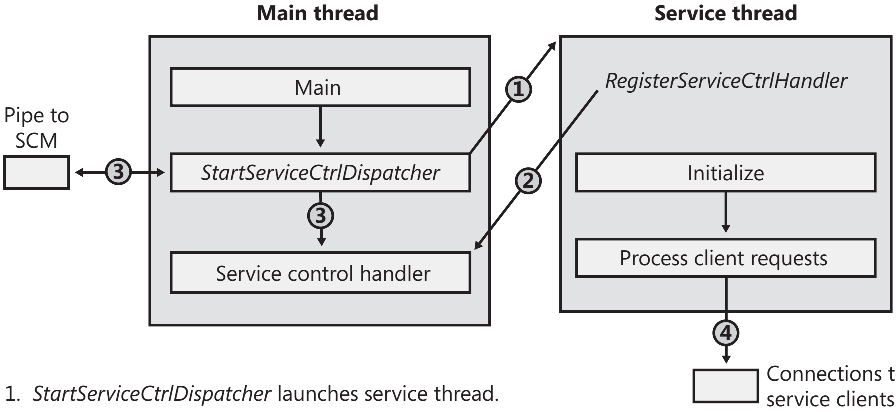
Value Setting	Value Name	Value Setting Description
DependOnService	Service name	The service won't load until after the specified service loads. This parameter doesn't apply to device drivers other than those with a start type of SERVICE_AUTO_START.
ObjectName	Usually LocalSystem, but can be an account name, such as .\Administrator	Specifies the account in which the service will run. If ObjectName isn't specified, LocalSystem is the account used. This parameter doesn't apply to device drivers.
DisplayName	Name of service	The service application shows services by this name. If no name is specified, the name of the service's registry key becomes its name.
Description	Description of service	Up to 32767-byte description of the service.
FailureActions	Description of actions the SCM should take when service process exits unexpectedly	Failure actions include restarting the service process, rebooting the system, and running a specified program. This value doesn't apply to drivers.
FailureCommand	Program command line	The SCM reads this value only if FailureActions specifies that a program should execute upon service failure. This value doesn't apply to drivers.
Security	Security descriptor	This value contains the security descriptor that defines who has what access to the service object created internally by the SCM.

Notice that Type values include three that apply to device drivers: device driver, file system driver, and file system recognizer. These are used by Windows device drivers, which also store their parameters as registry data in the Services registry key. The SCM is responsible for starting drivers with a Start value of SERVICE_AUTO_START or SERVICE_DEMAND_START, so it's natural for the SCM database to include drivers. Services use the other types, SERVICE_WIN32_OWN_PROCESS and SERVICE_WIN32_SHARE_PROCESS, which are mutually exclusive. An executable that hosts more than one service specifies the SERVICE_WIN32_SHARE_PROCESS type. An advantage to having a process run more than one service is that the system resources that would otherwise be required to run them in distinct processes are saved. A potential disadvantage is that if one of the services of a collection running in the same process causes an error that terminates the process, all the services of that process terminate. Also, another limitation is that all the services must run under the same account.

When the SCM starts a service process, the process immediately invokes the *StartServiceCtrlDispatcher* function. *StartServiceCtrlDispatcher* accepts a list of entry points into services, one entry point for each service in the process. Each entry point is identified by the name of the service the entry point corresponds to. After making a named pipe communications connection to the SCM, *StartServiceCtrlDispatcher* sits in a loop waiting for commands to come through the pipe from the SCM. The SCM sends a service-start command each time it starts a service the process owns. For each start command it receives, the *StartServiceCtrlDispatcher* function creates a thread, called a service thread, to invoke the starting service's entry point and implement the command loop for the service. *StartServiceCtrlDispatcher* waits indefinitely for commands from the SCM and returns control to the process's main function only when all the process's services have stopped, allowing the service process to clean up resources before exiting.

A service entry point's first action is to call the *RegisterServiceCtrlHandler* function. This function receives and stores a pointer to a function, called the *control handler*, which the service implements to handle various commands it receives from the SCM. *RegisterServiceCtrlHandler* doesn't communicate with the SCM, but it stores the function in local process memory for the *StartServiceCtrlDispatcher* function. The service entry point continues initializing the service, which can include allocating memory, creating communications end points, and reading private configuration data from the registry. A convention most services follow is to store their parameters under a subkey of their service registry key, named Parameters. While the entry point is initializing the service, it might periodically send status messages, using the *SetServiceStatus* function, to the SCM indicating how the service's startup is progressing. After the entry point finishes initialization, a service thread usually sits in a loop waiting for requests from client applications. For example, a Web server would initialize a TCP listen socket and wait for inbound HTTP connection requests.

A service process's main thread, which executes in the *StartServiceCtrlDispatcher* function, receives SCM commands directed at services in the process and invokes the target service's control handler function (stored by *RegisterServiceCtrlHandler*). SCM commands include stop, pause, resume, interrogate, and shutdown, or application-defined commands. Figure 4-9 shows the internal organization of a service process. Pictured are the two threads that make up a process hosting one service: the main thread and the service thread.



1. *StartServiceCtrlDispatcher* launches service thread.
2. Service thread registers control handler.
3. *StartServiceCtrlDispatcher* calls handlers in response to SCM commands.
4. Service thread processes client requests.

Figure 4-9 Inside a service process

SrvAny Tool

If you have a program that you want to run as a service, you need to modify the startup code to conform to the requirements for services outlined in this section. If you don't have the source code, you can use the SrvAny tool in the Windows resource kits. SrvAny enables you to run any application as a service. It reads the path of the service file that it must load from the Parameters subkey of the service's registry key. When SrvAny starts, it notifies the SCM that it is hosting a particular service, and when it receives a start command, it launches the service executable as a child process. The child process receives a copy of the SrvAny process's access token and a reference to the same window station, so the executable runs within the same security account and with the same interactivity setting as you specified when configuring the SrvAny process. SrvAny services don't have the share-process Type value, so each application you install as a service with SrvAny runs in a separate process with a different instance of the SrvAny host program.

Service Accounts

The security context of a service is an important consideration for service developers as well as for system administrators because it dictates what resources the process can access. Unless a service installation program or administrator specifies otherwise, most services run in the security context of the *local system account* (displayed sometimes as SYSTEM and other times as LocalSystem). Windows XP introduced two variants on the local system account, the *network service* and *local service* accounts. The new accounts have fewer capabilities than the local system account from a security standpoint, and any built-in Windows service that does not require the power of the local system account runs in the appropriate alternate service account. The following subsections describe the special characteristics of these accounts.

The Local System Account

The local system account is the same account in which core Windows user-mode operating system components run, including the Session Manager (\Windows\System32\Smss.exe), the Windows subsystem process (Csrss.exe), the local security authority subsystem (\Windows\System32\Lsass.exe), and the Winlogon process (\Windows\System32\Winlogon.exe).

From a security perspective, the local system account is extremely powerful—more powerful than any local or domain account when it comes to security ability on a local system. This account has the following characteristics:

- It is a member of the local administrators group. Table 4-8 shows the groups to which the local system account belongs. (See Chapter 8 for information on how group membership is used in object access checks.)
- It has the right to enable virtually every privilege (even privileges not normally granted to the local administrator account, such as creating security tokens). See Table 4-9 for the list of privileges assigned to the local system account. (Chapter 8 describes the use of each privilege.)
- Most files and registry keys grant full access to the local system account. (Even if they don't grant full access, a process running under the local system account can exercise the take-ownership privilege to gain access.)
- Processes running under the local system account run with the default user profile (HKU\.DEFAULT). Therefore, they can't access configuration information stored in the user profiles of other accounts.
- When a system is a member of a Windows domain, the local system account includes the machine security identifier (SID) for the computer on which a service process is running. Therefore, a service running in the local system account will be automatically authenticated on other machines in the same forest by using its computer account. (A *forest* is a grouping of domains.)
- Unless the machine account is specifically granted access to resources (such as network shares, named pipes, and so on), a process can access network resources that allow null sessions—that is, connections that require no credentials. You can specify the shares and pipes on a particular computer that permit null sessions in the NullSessionPipes and NullSessionShares registry values under HKLM\SYSTEM\CurrentControlSet\Services\lanmanserver\parameters.

Table 4-8 **Service Account Group Membership**

Local System	Network Service	Local Service
Everyone	Everyone	Everyone
Authenticated Users	Authenticated Users	Authenticated Users
Administrators	Users	Users
	Local	Local
	Network Service	Local Service
	Service	Service

Table 4-9 **Service Account Privileges**

Local System	Network Service	Local Service
SeAssignPrimaryTokenPrivilege	SeAssignPrimaryTokenPrivilege	SeAssignPrimaryTokenPrivilege
SeAuditPrivilege	SeAuditPrivilege	SeAuditPrivilege
SeBackupPrivilege	SeChangeNotifyPrivilege	SeChangeNotifyPrivilege
SeChangeNotifyPrivilege	SeIncreaseQuotaPrivilege	SeIncreaseQuotaPrivilege
SeCreateGlobalPrivilege	Privileges assigned to the Everyone, Authenticated Users, and Users groups	Privileges assigned to the Everyone, Authenticated Users, and Users groups
SeCreatePagefilePrivilege		
SeCreatePermanentPrivilege		
SeCreateTokenPrivilege*		
SeDebugPrivilege		
SeImpersonatePrivilege		
SeIncreaseBasePriorityPrivilege		
SeIncreaseQuotaPrivilege		
SeLoadDriverPrivilege		
SeLockMemoryPrivilege		
SeManageVolumePrivilege		
SeProfileSingleProcessPrivilege		
SeRestorePrivilege		
SeSecurityPrivilege		
SeShutdownPrivilege		
SeSystemEnvironmentPrivilege		
SeSystemTimePrivilege		
SeTakeOwnershipPrivilege		
SeTcbPrivilege		
SeUndockPrivilege		

* The local system account on Windows Server 2003 does not include this privilege.

The Network Service Account

The network service account is intended for use by services that wish to authenticate to other machines on the network using the computer account, as does the local system account, but do not have the need for membership in the administrators group or the use of many of the privileges assigned to the local system account. Because the network service account does not belong to the administrators group, services running in the network service account by default have access to far fewer registry keys and file system folders and files than the services running the local system account. Further, the assignment of few privileges limits the scope of a compromised network service process. For example, a process running in the network service account cannot load a device driver or open arbitrary processes.

Another difference between the network service and local system accounts is that processes running in the network service account use the network service account's profile. The registry component of the network service profile loads under HKU\S-1-5-20, and the files and directories that make up the component reside in \Documents and Settings\NetworkService.

A service that runs in the network service account in Windows XP and Windows Server 2003 is the DNS client, which is responsible for resolving DNS names and for locating domain controllers.

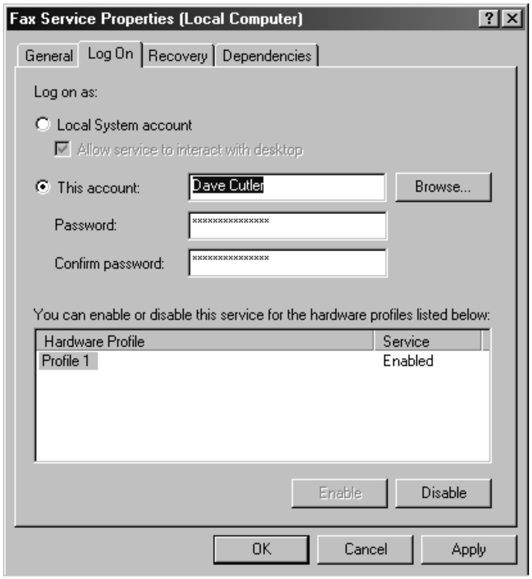
The Local Service Account

The local service account is virtually identical to the network service account with the important difference that it can access only network resources that allow anonymous access. Table 4-9 shows that it has the same privileges as the local service account, and Table 4-8 shows that it belongs to the same groups with the exception that it belongs to the Network Service group instead of the Local Service group. The profile used by processes running in the local service loads into HKU\S-1-5-19 and is stored in \Documents and Settings\LocalService.

Examples of services that Windows XP and Windows Server 2003 run in the local service account include the Remote Registry Service that allows remote access to the local system's registry, the Alerter service that receives network-broadcast administrative alerts messages, and the LmHosts service that performs NetBIOS name resolution.

Running Services in Alternate Accounts

Because of the restrictions just outlined, some services need to run with the security credentials of a user account. You can configure a service to run in an alternate account when the service is created or by specifying an account and password that the service should run under with the Windows Services MMC snap-in. In the Services snap-in, right-click on a service and select Properties, click the Log On tab, and select the This Account option, as shown in Figure 4-10.



Log on as:

○ Local System account

☑ Allow service to interact with desktop

● This account: Dave Cutler

Password: ***************

Confirm password: ***************

You can enable or disable this service for the hardware profiles listed below:

Hardware Profile	Service
Profile 1	Enabled

Figure 4-10 Service account settings

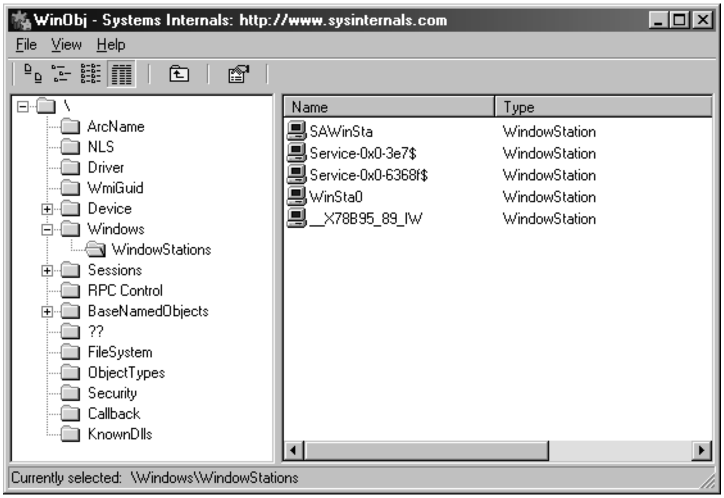
Interactive Services

Another restriction for services running under the local system, local service, and network service accounts is that they can't (without using a special flag on the *MessageBox* function, discussed in a moment) display dialog boxes or windows on the interactive user's desktop. This limitation isn't the direct result of running under these accounts but rather a consequence of the way the Windows subsystem assigns service processes to window stations.

The Windows subsystem associates every Windows process with a *window station*. A window station contains desktops, and desktops contain windows. Only one window station can be visible on a console and receive user mouse and keyboard input. In a Terminal Services environment, one window station per session is visible, but services all run as part of the console session. Windows names the visible window station WinSta0, and all interactive processes access WinSta0.

Unless otherwise directed, the Windows subsystem associates services running in the local system account with a nonvisible window station named Service-0x0-3e7$ that all noninteractive services share. The number in the name, *3e7*, represents the logon session identifier Lsass assigns to the logon session the SCM uses for noninteractive services running in the local system account.

Services configured to run under a user account (that is, not the local system account) are run in a different nonvisible window station named with the LSASS logon identifier assigned for the service's logon session. Figure 4-11 shows a sample display from the Winobj tool, available from *www.sysinternals.com*, viewing the object manager directory in which Windows places window station objects. Visible are the interactive window station (WinSta0), the noninteractive system service window station (Service-0x0-3e7$), and a noninteractive window station assigned to a service process logged on as a user (Service-0x0-6368f$).



Name	Type
SAWinSta	WindowStation
Service-0x0-3e7$	WindowStation
Service-0x0-6368f$	WindowStation
WinSta0	WindowStation
__X78B95_89_IW	WindowStation

Currently selected: \Windows\WindowStations

Figure 4-11 List of window stations

Regardless of whether services are running in a user account, the local system account, or the local or network service accounts, services that aren't running on the visible window station can't receive input from a user or display windows on the console. In fact, if a service were to pop up a normal dialog box on the window station, the service would appear hung because no user would be able to see the dialog box, which of course would prevent the user from providing keyboard or mouse input to dismiss it and allow the service to continue executing. (The one exception is if the special flag MB_SERVICE_NOTIFICATION or MB_DEFAULT_DESKTOP_ ONLY is set on the *MessageBox* call—if MB_SERVICE_NOTIFICATION is specified, the message box will always be displayed on the interactive window station, even if the service wasn't configured with permission to interact with the user; if MB_DEFAULT_DESKTOP_ONLY is specified, the message box is displayed on the default desktop of the interactive window station.)

In rare cases, a service can have a valid reason to interact with the user via dialog boxes or windows. To configure a service with the right to interact with the user, the SERVICE_ INTERACTIVE_PROCESS modifier must be present in the service's registry key's Type parameter. (Note that services configured to run under a user account can't be marked as interactive.) When the SCM starts a service marked as interactive, it launches the service's process in the local system account's security context but connects the service with WinSta0 instead of the noninteractive service window station. This connection to WinSta0 allows the service to display dialog boxes and windows on the console and allows those windows to respond to user input.

Note Microsoft discourages running interactive services, especially in the local system account, because of the inherent security vulnerability it creates. Windows presented by an interactive service are susceptible to the receipt of windows messages that a malicious process running on the desktop of an unprivileged user can use to cause buffer overflows in the service process and subvert the service process to elevate the security privileges of the malicious process.

The Service Control Manager

The SCM's executable file is \Windows\System32\Services.exe, and like most service processes, it runs as a Windows console program. The Winlogon process starts the SCM early during the system boot. (Refer to Chapter 5 for details on the boot process.) The SCM's startup function, *SvcCtrlMain*, orchestrates the launching of services that are configured for automatic startup. *SvcCtrlMain* executes shortly after the screen switches to a blank desktop but generally before Winlogon has loaded the graphical identification and authentication interface (GINA) that presents a logon dialog box.

SvcCtrlMain first creates a synchronization event named SvcCtrlEvent_ A3752DX that it initializes as nonsignaled. Only after the SCM completes steps necessary to prepare it to receive commands from SCPs does the SCM set the event to a signaled state. The function that an SCP uses to establish a dialog with the SCM is *OpenSCManager*. *OpenSCManager* prevents an SCP from trying to contact the SCM before the SCM has initialized by waiting for SvcCtrlEvent_A3752DX to become signaled.

Next, *SvcCtrlMain* gets down to business and calls *ScCreateServiceDB*, the function that builds the SCM's internal service database. *ScCreateServiceDB* reads and stores the contents of HKLM\SYSTEM\CurrentControlSet\Control\ServiceGroupOrder\List, a REG_MULTI_SZ value that lists the names and order of the defined service groups. A service's registry key contains an optional Group value if that service or device driver needs to control its startup ordering with respect to services from other groups. For example, the Windows networking stack is built from the bottom up, so networking services must specify Group values that place them later in the startup sequence than networking device drivers. SCM internally creates a group list that preserves the ordering of the groups it reads from the registry. Groups include (but are not limited to) NDIS, TDI, Primary Disk, Keyboard Port, and Keyboard Class. Add-on and third-party applications can even define their own groups and add them to the list. Microsoft Transaction Server, for example, adds a group named MS Transactions.

ScCreateServiceDB then scans the contents of HKLM\SYSTEM\CurrentControlSet\Services, creating an entry in the service database for each key it encounters. A database entry includes all the service-related parameters defined for a service as well as fields that track the service's status. The SCM adds entries for device drivers as well as for services because the SCM starts services and drivers marked as auto-start and detects startup failures for drivers marked boot-start and system-start. It also provides a means for applications to query the status of drivers.

The I/O manager loads drivers marked boot-start and system-start before any user-mode processes execute, and therefore any drivers having these start types load before the SCM starts.

ScCreateServiceDB reads a service's Group value to determine its membership in a group and associates this value with the group's entry in the group list created earlier. The function also reads and records in the database the service's group and service dependencies by querying its DependOnGroup and DependOnService registry values. Figure 4-12 shows how the SCM organizes the service entry and group order lists. Notice that the service list is alphabetically sorted. The reason this list is sorted alphabetically is that the SCM creates the list from the Services registry key, and Windows stores registry keys alphabetically.

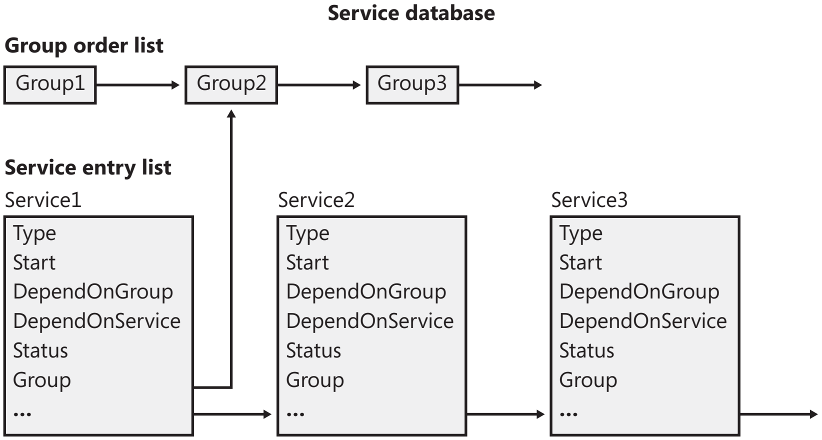
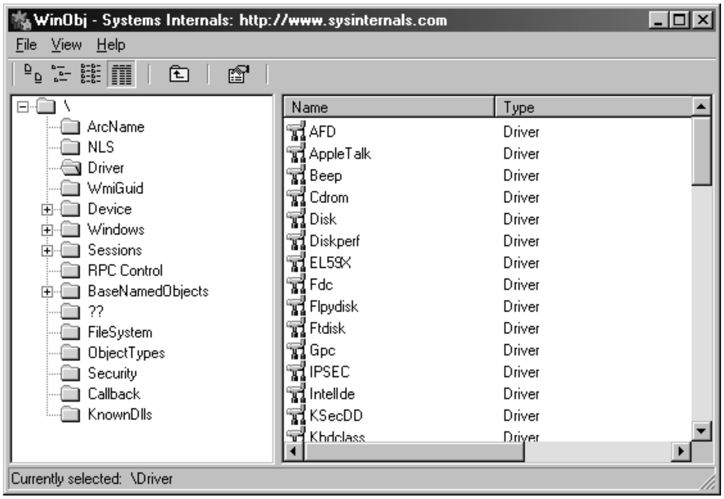


Figure 4-12 Organization of a service database

During service startup, the SCM might need to call on LSASS (for example, to log on a service in a user account), so the SCM waits for LSASS to signal the LSA_RPC_SERVER_ACTIVE synchronization event, which it does when it finishes initializing. Winlogon also starts the LSASS process, so the initialization of LSASS is concurrent with that of the SCM, and the order in which LSASS and the SCM complete initialization can vary. Then *SvcCtrlMain* calls *ScGetBootAndSystemDriverState* to scan the service database looking for boot-start and system-start device driver entries. *ScGetBootAndSystemDriverState* determines whether or not a driver successfully started by looking up its name in the object manager namespace directory named \Driver. When a device driver successfully loads, the I/O manager inserts the driver's object in the namespace under this directory, so if its name isn't present, it hasn't loaded. Figure 4-13 shows Winobj displaying the contents of the Driver directory. If a driver isn't loaded, the SCM looks for its name in the list of drivers returned by the *PnP_DeviceList* function. *PnP_DeviceList* supplies the drivers included in the system's current hardware profile. *SvcCtrlMain* notes the names of drivers that haven't started and that are part of the current profile in a list named ScFailedDrivers.

Before starting the auto-start services, the SCM performs a few more steps. It creates its remote procedure call (RPC) named pipe, which is named \Pipe\Ntsvcs, and then RPC launches a thread to listen on the pipe for incoming messages from SCPs. The SCM then signals its initialization-complete event, SvcCtrlEvent_A3752DX. Registering a console application shutdown event handler and registering with the Windows subsystem process via *RegisterServiceProcess* prepares the SCM for system shutdown.



Name	Type
AFD	Driver
AppleTalk	Driver
Beep	Driver
Cdrom	Driver
Disk	Driver
Diskperf	Driver
EL59X	Driver
Fdc	Driver
Flpydisk	Driver
Ftdisk	Driver
Gpc	Driver
IPSEC	Driver
Intellde	Driver
KSecDD	Driver
Kbdclass	Driver

Currently selected: \Driver

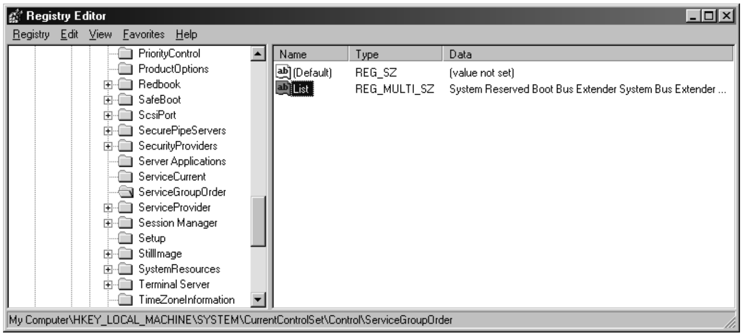
Figure 4-13 List of driver objects

Network Drive Letters

In addition to its role as an interface to services, the SCM has another totally unrelated responsibility: it notifies GUI applications in a system whenever the system creates or deletes a network drive-letter connection. The SCM waits for the Multiple Provider Router (MPR) to signal a named event, \BaseNamedObjects\ScNetDrvMsg, which MPR signals whenever an application assigns a drive letter to a remote network share or deletes a remote-share drive-letter assignment. (See Chapter 13 for more information on MPR.) When MPR signals the event, the SCM calls the *GetDriveType* Windows function to query the list of connected network drive letters. If the list changes across the event signal, the SCM sends a Windows broadcast message of type WM_DEVICECHANGE. The SCM uses either DBT_ DEVICEREMOVECOMPLETE or DBT_DEVICEARRIVAL as the message's subtype. This message is primarily intended for Windows Explorer so that it can update any open My Computer windows to show the presence or absence of a network drive letter.

Service Startup

SvcCtrlMain invokes the SCM function *ScAutoStartServices* to start all services that have a Start value designating auto-start. *ScAutoStartServices* also starts auto-start device drivers. To avoid confusion, you should assume that the term *services* means *services and drivers* unless indicated otherwise. The algorithm in *ScAutoStartServices* for starting services in the correct order proceeds in phases, whereby a phase corresponds to a group and phases proceed in the sequence defined by the group ordering stored in the HKLM\SYSTEM\CurrentControlSet\Control\ServiceGroupOrder\List registry value. The List value, shown in Figure 4-14, includes the names of groups in the order that the SCM should start them. Thus, assigning a service to a group has no effect other than to fine-tune its startup with respect to other services belonging to different groups.



Name	Type	Data
(Default)	REG_SZ	(value not set)
List	REG_MULTI_SZ	System Reserved Boot Bus Extender System Bus Extender ...

My Computer\HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\ServiceGroupOrder

Figure 4-14 ServiceGroupOrder registry key

When a phase starts, *ScAutoStartServices* marks all the service entries belonging to the phase's group for startup. Then *ScAutoStartServices* loops through the marked services seeing whether it can start each one. Part of the check it makes consists of determining whether the service has a dependency on another group, as specified by the existence of the DependOnGroup value in the service's registry key. If a dependency exists, the group on which the service is dependent must have already initialized, and at least one service of that group must have successfully started. If the service depends on a group that starts later than the service's group in the group startup sequence, the SCM notes a "circular dependency" error for the service. If *ScAutoStartServices* is considering a Windows service or an auto-start device driver, it next checks to see whether the service depends on one or more other services, and if so, if those services have already started. Service dependencies are indicated with the DependOnService registry value in a service's registry key. If a service depends on other services that belong to groups that come later in the ServiceGroupOrder\List, the SCM also generates a "circular dependency" error and doesn't start the service. If the service depends on any services from the same group that haven't yet started, the service is skipped.

When the dependencies of a service have been satisfied, *ScAutoStartServices* makes a final check to see whether the service is part of the current boot configuration before starting the service. When the system is booted in safe mode, the SCM ensures that the service is either identified by name or by group in the appropriate safe boot registry key. There are two safe boot keys, Minimal and Network, under HKLM\SYSTEM\CurrentControlSet\Control\SafeBoot, and the one that the SCM checks depends on what safe mode the user booted. If the user chose Safe Mode or Safe Mode With Command Prompt at the special boot menu (which you can access by pressing F8 when prompted in the boot process), the SCM references the

Minimal key; if the user chose Safe Mode With Networking, the SCM refers to Network. The existence of a string value named Option under the SafeBoot key indicates not only that the system booted in safe mode but also the type of safe mode the user selected. For more information about safe boots, see the section "Safe Mode" in Chapter 5.

Once the SCM decides to start a service, it calls *ScStartService*, which takes different steps for services than for device drivers. When *ScStartService* starts a Windows service, it first determines the name of the file that runs the service's process by reading the ImagePath value from the service's registry key. It then examines the service's Type value, and if that value is SERVICE_WINDOWS_SHARE_PROCESS (0x20), the SCM ensures that the process the service runs in, if already started, is logged on using the same account as specified for the service being started. A service's ObjectName registry value stores the user account in which the service should run. A service with no ObjectName or an ObjectName of LocalSystem runs in the local system account.

The SCM verifies that the service's process hasn't already been started in a different account by checking to see whether the service's ImagePath value has an entry in an internal SCM database called the *image database*. If the image database doesn't have an entry for the ImagePath value, the SCM creates one. When the SCM creates a new entry, it stores the logon account name used for the service and the data from the service's ImagePath value. The SCM requires services to have an ImagePath value. If a service doesn't have an ImagePath value, the SCM reports an error stating that it couldn't find the service's path and isn't able to start the service. If the SCM locates an existing image database entry with matching ImagePath data, the SCM ensures that the user account information for the service it's starting is the same as the information stored in the database entry—a process can be logged on as only one account, so the SCM reports an error when a service specifies a different account name than another service that has already started in the same process.

The SCM calls *ScLogonAndStartImage* to log on a service if the service's configuration specifies and to start the service's process. The SCM logs on services that don't run in the system account by calling the LSASS function *LsaLogonUser*. *LsaLogonUser* normally requires a password, but the SCM indicates to LSASS that the password is stored as a service's LSASS "secret" under the key HKLM\SECURITY\Policy\Secrets in the registry. (Keep in mind that the contents of the SECURITY aren't typically visible because its default security settings permit access only from the system account.) When the SCM calls *LsaLogonUser*, it specifies a service logon as the logon type, so LSASS looks up the password in the Secrets subkey that has a name in the form _SC_<service name>.

The SCM directs LSASS to store a logon password as a secret using the *LsaStorePrivateData* function when an SCP configures a service's logon information. When a logon is successful, *LsaLogonUser* returns a handle to an access token to the caller. Windows uses access tokens to represent a user's security context, and the SCM later associates the access token with the process that implements the service.

After a successful logon, the SCM loads the account's profile information, if it's not already loaded, by calling the UserEnv DLL's (\Windows\System32\Userenv.dll) *LoadUserProfile* function. The value HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\ProfileList\<user profile key>\ProfileImagePath contains the location on disk of a registry hive that *LoadUserProfile* loads into the registry, making the information in the hive the HKEY_CURRENT_USER key for the service.

An interactive service must open the WinSta0 window station, but before *ScLogonAndStartImage* allows an interactive service to access WinSta0 it checks to see whether the value HKLM\SYSTEM\CurrentControlSet\Control\Windows\NoInteractiveServices is set. Administrators set this value to prevent services marked as interactive from displaying windows on the console. This option is desirable in unattended server environments in which no user is present to respond to popups from interactive services.

As its next step, *ScLogonAndStartImage* proceeds to launch the service's process, if the process hasn't already been started (for another service, for example). The SCM starts the process in a suspended state with the *CreateProcessAsUser* Windows function. The SCM next creates a named pipe through which it communicates with the service process, and it assigns the pipe the name \Pipe\Net\NtControlPipeX, where *X* is a number that increments each time the SCM creates a pipe. The SCM resumes the service process via the *ResumeThread* function and waits for the service to connect to its SCM pipe. If it exists, the registry value HKLM\SYSTEM\CurrentControlSet\Control\ServicesPipeTimeout determines the length of time that the SCM waits for a service to call *StartServiceCtrlDispatcher* and connect before it gives up, terminates the process, and concludes that the service failed to start. If ServicesPipeTimeout doesn't exist, the SCM uses a default timeout of 30 seconds. The SCM uses the same timeout value for all its service communications.

When a service connects to the SCM through the pipe, the SCM sends the service a start command. If the service fails to respond positively to the start command within the timeout period, the SCM gives up and moves on to start the next service. When a service doesn't respond to a start request, the SCM doesn't terminate the process, as it does when a service doesn't call *StartServiceCtrlDispatcher* within the timeout; instead, it notes an error in the system Event Log that indicates the service failed to start in a timely manner.

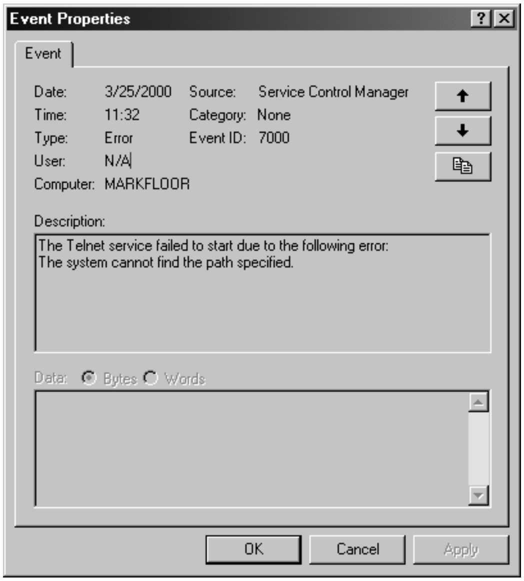
If the service the SCM starts with a call to *ScStartService* has a Type registry value of SERVICE_KERNEL_DRIVER or SERVICE_FILE_SYSTEM_ DRIVER, the service is really a device driver, and so *ScStartService* calls *ScLoadDeviceDriver* to load the driver. *ScLoadDeviceDriver* enables the load driver security privilege for the SCM process and then invokes the kernel service *NtLoadDriver*, passing in the data in the ImagePath value of the driver's registry key. Unlike services, drivers don't need to specify an ImagePath value, and if the value is absent, the SCM builds an image path by appending the driver's name to the string \Windows\System32\Drivers\.

ScAutoStartServices continues looping through the services belonging to a group until all the services have either started or generated dependency errors. This looping is the SCM's way of automatically ordering services within a group according to their DependOnService dependencies. The SCM will start the services that other services depend on in earlier loops, skipping the dependent services until subsequent loops. Note that the SCM ignores Tag values for Windows services, which you might come across in subkeys under the HKLM\SYSTEM\CurrentControlSet\Services key; the I/O manager honors Tag values to order device driver startup within a group for boot and system-start drivers.

Once the SCM completes phases for all the groups listed in the ServiceGroupOrder\List value, it performs a phase for services belonging to groups not listed in the value and a final phase for services without a group. When it's finished starting all auto-start services and drivers, the SCM signals the event \BaseNamedObjects\SC_AutoStartComplete.

Startup Errors

If a driver or a service reports an error in response to the SCM's startup command, the ErrorControl value of the service's registry key determines how the SCM reacts. If the ErrorControl value is SERVICE_ERROR_IGNORE (0) or the ErrorControl value isn't specified, the SCM simply ignores the error and continues processing service startups. If the ErrorControl value is SERVICE_ERROR_NORMAL (1), the SCM writes an event to the system Event Log that says, "The <service name> service failed to start due to the following error:". The SCM includes the textual representation of the Windows error code that the service returned to the SCM as the reason for the startup failure in the Event Log record. Figure 4-15 shows the Event Log entry that reports a service startup error.



Date: 3/25/2000 Source: Service Control Manager
Time: 11:32 Category: None
Type: Error Event ID: 7000
User: N/A
Computer: MARKFLOOR

Description:
The Telnet service failed to start due to the following error:
The system cannot find the path specified.

Data: ● Bytes ○ Words

Figure 4-15 Service startup failure Event Log entry

If a service with an ErrorControl value of SERVICE_ERROR_SEVERE (2) or SERVICE_ERROR_CRITICAL (3) reports a startup error, the SCM logs a record to the Event Log and then calls the internal function *ScRevertToLastKnownGood*. This function switches the system's registry configuration to a version, named last known good, with which the system last booted successfully. Then it restarts the system using the *NtShutdownSystem* system service, which is implemented in the executive. If the system is already booting with the last known good configuration, the system just reboots.

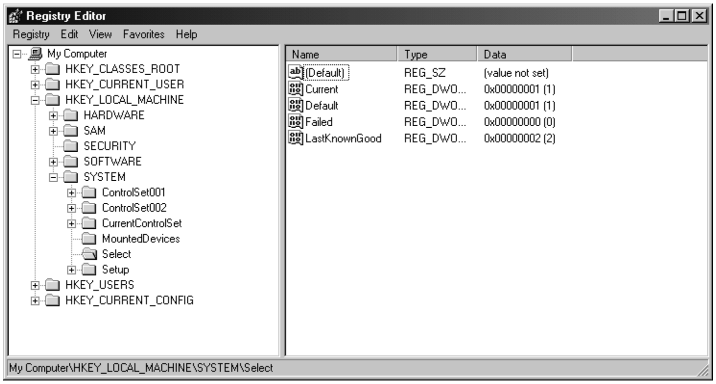
Accepting the Boot and Last Known Good

Besides starting services, the system charges the SCM with determining when the system's registry configuration, HKLM\SYSTEM\CurrentControlSet, should be saved as the last known good control set. The CurrentControlSet key contains the Services key as a subkey, so CurrentControlSet includes the registry representation of the SCM database. It also contains the Control key, which stores many kernel-mode and user-mode subsystem configuration settings. By default, a successful boot consists of a successful startup of auto-start services and a successful user logon. A boot fails if the system halts because a device driver crashes the system during the boot or if an auto-start service with an ErrorControl value of SERVICE_ERROR_SEVERE or SERVICE_ERROR_CRITICAL reports a startup error.

The SCM obviously knows when it has completed a successful startup of the auto-start services, but Winlogon (\Windows\System32\Winlogon.exe) must notify it when there is a successful logon. Winlogon invokes the *NotifyBootConfigStatus* function when a user logs on, and *NotifyBootConfigStatus* sends a message to the SCM. Following the successful start of the auto-start services or the receipt of the message from *NotifyBootConfigStatus* (whichever comes last), the SCM calls the system function *NtInitializeRegistry* to save the current registry startup configuration.

Third-party software developers can supersede Winlogon's definition of a successful logon with their own definition. For example, a system running Microsoft SQL Server might not consider a boot successful until after SQL Server is able to accept and process transactions. Developers impose their definition of a successful boot by writing a boot-verification program and installing the program by pointing to its location on disk with the value stored in the registry key HKLM\SYSTEM\CurrentControlSet\Control\BootVerificationProgram. In addition, a boot-verification program's installation must disable Winlogon's call to *NotifyBootConfigStatus* by setting HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon\ReportBootOk to 0. When a boot-verification program is installed, the SCM launches it after finishing auto-start services and waits for the program's call to *NotifyBootConfigStatus* before saving the last known good control set.

Windows maintains several copies of CurrentControlSet, and CurrentControlSet is really a symbolic registry link that points to one of the copies. The control sets have names in the form HKLM\SYSTEM\ControlSet*nnn*, where *nnn* is a number such as 001 or 002. The HKLM\SYSTEM\Select key contains values that identify the role of each control set. For example, if CurrentControlSet points to ControlSet001, the Current value under Select has a value of 1. The LastKnownGood value under Select contains the number of the last known good control set, which is the control set last used to boot successfully. Another value that might be on your system under the Select key is Failed, which points to the last control set for which the boot was deemed unsuccessful and aborted in favor of an attempt at booting with the last known good control set. Figure 4-16 displays a system's control sets and Select values.



Name	Type	Data
(Default)	REG_SZ	(value not set)
Current	REG_DWO...	0x00000001 (1)
Default	REG_DWO...	0x00000001 (1)
Failed	REG_DWO...	0x00000000 (0)
LastKnownGood	REG_DWO...	0x00000002 (2)

My Computer\HKEY_LOCAL_MACHINE\SYSTEM\Select

Figure 4-16 Control set selection key

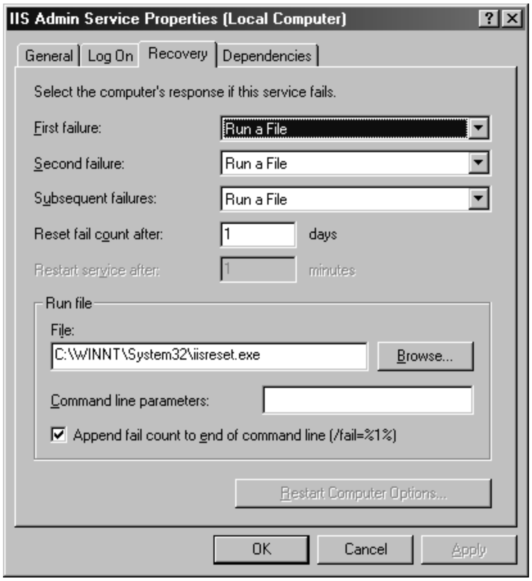
NtInitializeRegistry takes the contents of the last known good control set and synchronizes it with that of the CurrentControlSet key's tree. If this was the system's first successful boot, the last known good won't exist and the system will create a new control set for it. If the last known good tree exists, the system simply updates it with differences between it and CurrentControlSet.

Last known good is helpful in situations in which a change to CurrentControlSet, such as the modification of a system performance-tuning value under HKLM\SYSTEM\Control or the addition of a service or device driver, causes the subsequent boot to fail. Users can press F8 early in the boot process to bring up a menu that lets them direct the boot to use the last known good control set, rolling the system's registry configuration back to the way it was the last time the system booted successfully. Chapter 5 describes in more detail the use of Last Known Good and other recovery mechanisms for troubleshooting system startup problems.

Service Failures

A service can have optional FailureActions and FailureCommand values in its registry key that the SCM records during the service's startup. The SCM registers with the system so that the system signals the SCM when a service process exits. When a service process terminates unexpectedly, the SCM determines which services ran in the process and takes the recovery steps specified by their failure-related registry values.

Actions that a service can configure for the SCM include restarting the service, running a program, and rebooting the computer. Furthermore, a service can specify the failure actions that take place the first time the service process fails, the second time, and subsequent times, and it can indicate a delay period that the SCM waits before restarting the service if the service asks to be restarted. The service failure action of the IIS Admin Service results in the SCM running the IISReset application, which performs cleanup work and then restarts the service. You can easily manage the recovery actions for a service with the Recovery tab of the service's Properties dialog box in the Services MMC snap-in, as shown in Figure 4-17.


IIS Admin Service Properties (Local Computer)
?
General
Log On
Recovery
Dependencies
Browse...
Restart Computer Options...
OK
Cancel
Apply

Select the computer's response if this service fails.

First failure: Run a File

Second failure: Run a File

Subsequent failures: Run a File

Reset fail count after: 1 days

Restart service after: 1 minutes

Run file

File: C:\WINNT\System32\iisreset.exe

Command line parameters:

☑ Append fail count to end of command line (/fail=%1%)

Figure 4-17 Service recovery options

Service Shutdown

When Winlogon calls the Windows *ExitWindowsEx* function, *ExitWindowsEx* sends a message to Csrss, the Windows subsystem process, to invoke Csrss's shutdown routine. Csrss loops through the active processes and notifies them that the system is shutting down. For every system process except the SCM, Csrss waits up to the number of seconds specified by HKU\.DEFAULT\Control Panel\Desktop\WaitToKillAppTimeout (which defaults to 20 seconds) for the process to exit before moving on to the next process. When Csrss encounters the SCM process, it also notifies it that the system is shutting down but employs a timeout specific to the SCM. Csrss recognizes the SCM using the process ID Csrss saved when the SCM registered with Csrss using the *RegisterServicesProcess* function during system initialization. The SCM's timeout differs from that of other processes because Csrss knows that the SCM communicates with services that need to perform cleanup when they shut down, and so an administrator might need to tune only the SCM's timeout. The SCM's timeout value resides in the HKLM\SYSTEM\CurrentControlSet\Control\WaitToKillServiceTimeout registry value, and it defaults to 20 seconds.

The SCM's shutdown handler is responsible for sending shutdown notifications to all the services that requested shutdown notification when they initialized with the SCM. The SCM function *ScShutdownAllServices* loops through the SCM services database searching for services desiring shutdown notification and sends each one a shutdown command. For each service to which it sends a shutdown command, the SCM records the value of the service's *wait hint*, a value that a service also specifies when it registers with the SCM. The SCM keeps track of the largest wait hint it receives. After sending the shutdown messages, the SCM waits either until one of the services it notified of shutdown exits or until the time specified by the largest wait hint passes.

If the wait hint expires without a service exiting, the SCM determines whether one or more of the services it was waiting on to exit have sent a message to the SCM telling the SCM that the service is progressing in its shutdown process. If at least one service made progress, the SCM waits again for the duration of the wait hint. The SCM continues executing this wait loop until either all the services have exited or none of the services upon which it's waiting has notified it of progress within the wait hint timeout period.

While the SCM is busy telling services to shut down and waiting for them to exit, Csrss waits for the SCM to exit. If Csrss's wait ends without the SCM having exited (the WaitToKillServiceTimeout time expires), Csrss simply moves on, continuing the shutdown process. Thus, services that fail to shut down in a timely manner are simply left running, along with the SCM, as the system shuts down. Unfortunately, there's no way for administrators to know whether they should raise the WaitToKillServiceTimeout value on systems where services aren't getting a chance to shut down completely before the system shuts down. See "Shutdown" in Chapter 5 for more information on the shutdown process.

Shared Service Processes

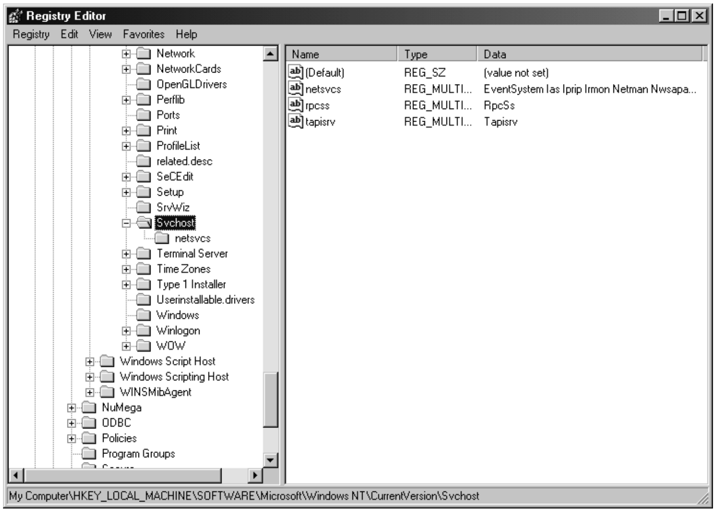
Running every service in its own process instead of having services share a process whenever possible wastes system resources. However, sharing processes means that if any of the services in the process has a bug that causes the process to exit, all the services in that process terminate.

Of the Windows built-in services, some run in their own process and some share a process with other services. For example, the SCM process hosts the Event Log service and the user-mode Plug and Play service, and the LSASS process contains security-related services—such as the Security Accounts Manager (SamSs) service, the Net Logon (Netlogon) service, and the IPSec Policy Agent (PolicyAgent) service.

There is also a "generic" process named Service Host (SvcHost - \Windows\System32\Svchost.exe) to contain multiple services. Multiple instances of SvcHost can be running in different processes. Services that run in SvcHost processes include Telephony

(TapiSrv), Remote Procedure Call (RpcSs), and Remote Access Connection Manager (RasMan). Windows implements services that run in SvcHost as DLLs and includes an ImagePath definition of the form "%SystemRoot%\System32\svchost.exe -k netsvcs" in the service's registry key. The service's registry key must also have a registry value named ServiceDll under a Parameters subkey that points to the service's DLL file.

All services that share a common SvcHost process specify the same parameter ("-k netsvcs" in the example in the preceding paragraph) so that they have a single entry in the SCM's image database. When the SCM encounters the first service that has a SvcHost ImagePath with a particular parameter during service startup, it creates a new image database entry and launches a SvcHost process with the parameter. The new SvcHost process takes the parameter and looks for a value having the same name as the parameter under HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Svchost. SvcHost reads the contents of the value, interpreting it as a list of service names, and notifies the SCM that it's hosting those services when SvcHost registers with the SCM. Figure 4-18 presents an example Svchost registry key that shows that a SvcHost process started with the "-k netsvcs" parameter is prepared to host a number of different network-related services.



Name	Type	Data
(Default)	REG_SZ	(value not set)
netsvcs	REG_MULTI...	EventSystem Ias Iprip Irmon Netman Nwsapa...
rpcss	REG_MULTI...	RpcSs
tapisrv	REG_MULTI...	Tapisrv

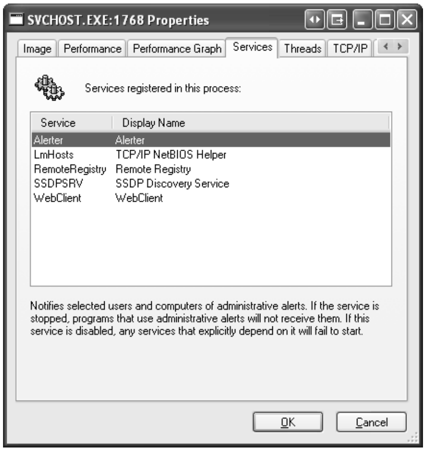
My Computer\HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Svchost

Figure 4-18 Svchost registry key

When the SCM encounters a SvcHost service during service startup with an ImagePath matching an entry it already has in the image database, it doesn't launch a second process but instead just sends a start command for the service to the SvcHost it already started for that ImagePath value. The existing SvcHost process reads the ServiceDll parameter in the service's registry key and loads the DLL into its process to start the service.

EXPERIMENT: Viewing Services Running Inside Processes

The Process Explorer utility that you can download from *www.sysinternals.com* shows detailed information about the services running with processes. Run Process Explorer and view Services tabs on the process properties dialog box for the following processes: Services.exe, Lsass.exe, and Svchost.exe. Several instances of SvcHost will be running on your system, and you can see the account in which each is running by adding the Username column to the Process Explorer display or by looking at the Username field on the Image tab of a process's Process Properties dialog box. The following figure shows the list of services running within a SvcHost executing in the local service account:



Services registered in this process:

Service	Display Name
Alerter	Alerter
LmHosts	TCP/IP NetBIOS Helper
RemoteRegistry	Remote Registry
SSDPSRV	SSDP Discovery Service
WebClient	WebClient

Notifies selected users and computers of administrative alerts. If the service is stopped, programs that use administrative alerts will not receive them. If this service is disabled, any services that explicitly depend on it will fail to start.

The information displayed includes the service name, display name, and service description, if it has one, which Process Explorer shows beneath the service list when you select a service.

You can also use the tlist.exe tool from the Windows Support Tools or Tasklist, which ships with Windows XP and Windows Server 2003, to view the list of services running within processes from a command prompt. The syntax to see services with Tlist is:

```
tlist /s
```

The syntax for tasklist is:

```
tasklist /svc
```

Note that these utilities do not show service display names or descriptions, only service names.

Service Control Programs

Service control programs are standard Windows applications that use XSCM service management functions, including *CreateService*, *OpenService*, *StartService*, *ControlService*, *QueryServiceStatus,* and *DeleteService*. To use the SCM functions, an SCP must first open a communications channel to the SCM by calling the *OpenSCManager* function. At the time of the open call, the SCP must specify what types of actions it wants to perform. For example, if an SCP simply wants to enumerate and display the services present in the SCM's database, it requests enumerate-service access in its call to *OpenSCManager*. During its initialization, the SCM creates an internal object that represents the SCM database and uses the Windows security functions to protect the object with a security descriptor that specifies what accounts can open the object with what access permissions. For example, the security descriptor indicates that the Authenticated Users group can open the SCM object with enumerate-service access. However, only administrators can open the object with the access required to create or delete a service.

As it does for the SCM database, the SCM implements security for services themselves. When an SCP creates a service by using the *CreateService* function, it specifies a security descriptor that the SCM associates internally with the service's entry in the service database. The SCM stores the security descriptor in the service's registry key as the Security value, and it reads that value when it scans the registry's Services key during initialization so that the security settings persist across reboots. In the same way that an SCP must specify what types of access it wants to the SCM database in its call to *OpenSCManager*, an SCP must tell the SCM what access it wants to a service in a call to *OpenService*. Accesses that an SCP can request include the ability to query a service's status and to configure, stop, and start a service.

The SCP you're probably most familiar with is the Services MMC snap-in that's included in Windows, which resides in \Windows\System32\Filemgmt.dll. Windows XP and Windows Server 2003 include Sc.exe (Service Controller tool), a command-line service control program that's available for Windows 2000 in the Windows 2000 resource kits.

SCPs sometimes layer service policy on top of what the SCM implements. A good example is the timeout that the Services MMC snap-in implements when a service is started manually. The snap-in presents a progress bar that represents the progress of a service's startup. Whereas the SCM waits indefinitely for a service to respond to a start command, the Services snap-in waits only 2 minutes before the progress bar reaches 100 percent and the snap-in announces that the service didn't start in a timely manner. Services indirectly interact with SCPs by setting their configuration status to reflect their progress as they respond to SCM commands such as the start command. SCPs query the status with the *QueryServiceStatus* function. They can tell when a service actively updates the status versus when a service appears to be hung, and the SCM can take appropriate actions in notifying a user about what the service is doing.

Windows Management Instrumentation

Windows NT has always had integrated performance and system-event monitoring tools. Applications and the system typically use the Event Manager to report errors and diagnostic messages. The Event Viewer utility lets administrators view event output from either the local computer or another computer on the network. Similarly, the performance counter mechanism lets applications and operating system components report performance-related statistics to performance-monitoring applications such as the Performance Monitor.

Although the Windows NT 4 event-monitoring and performance-monitoring features met their design goals, they had limitations. For example, the programming interfaces differ from one another, and this variation increases the complexity of applications that use both event and performance monitoring to collect data. Perhaps the biggest drawback to the monitoring facilities in Windows NT 4 is that they have little or no extensibility and that neither event logging nor performance data collection provides the two-way interaction necessary in a management API. Applications must provide data in predefined formats. The Performance API provides no way for an application to receive notification of performance-related events, and applications that request notification of Event Manager events can't restrict notification to specific event types or sources. Finally, clients of either collection facility can't communicate with event-data or performance-data providers through the Event Manager or Performance API.

To address these limitations as well as to provide management capabilities for other types of data sources, Windows has a new management mechanism, Windows Management Instrumentation (WMI). WMI is an implementation of Web-Based Enterprise Management (WBEM), a standard that the Distributed Management Task Force (DMTF—an industry consortium) defines. The WBEM standard encompasses the design of an extensible enterprise data-collection and data-management facility that has the flexibility and extensibility required to manage local and remote systems that comprise arbitrary components. WMI support was added to Windows NT 4 in Service Pack 4. It is also supported in Windows 95 OSR2, Windows 98 and Windows Millennium. Although most of this section applies to all the Windows platforms that support WMI, implementation details are specific to Windows 2000, Windows XP, and Windows Server 2003.

WMI Architecture

WMI consists of four main components, as shown in Figure 4-19: management applications, WMI infrastructure, providers, and managed objects. Management applications are Windows applications that access and display or process data that the applications obtain about managed objects. A simple example of a management application is a Performance tool replacement that relies on WMI rather than the Performance API to obtain performance information. A more complex example is an enterprise-management tool that lets administrators perform automated inventories of the software and hardware configuration of every computer in their enterprise.

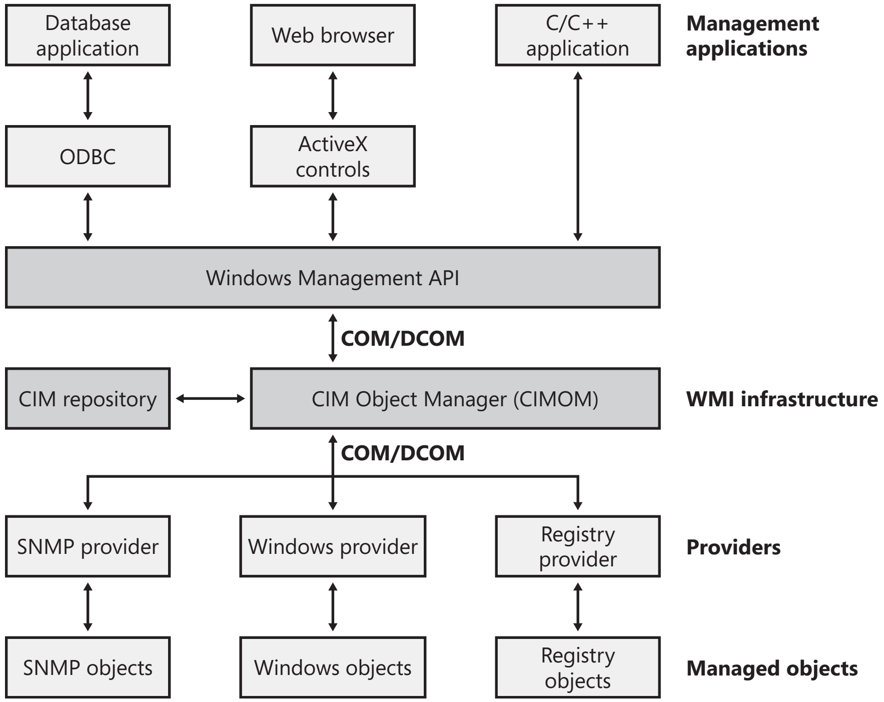


Figure 4-19 WMI architecture

Developers typically must target management applications to collect data from and manage specific objects. An object might represent one component, such as a network adapter device, or a collection of components, such as a computer. (The computer object might contain the network adapter object.) Providers need to define and export the representation of the objects that management applications are interested in. For example, the vendor of a network adapter might want to add adapter-specific properties to the network adapter WMI support that Windows includes, querying and setting the adapter's state and behavior as the management applications direct. In some cases (for example, for device drivers), Microsoft supplies a provider that has its own API to help developers leverage the provider's implementation for their own managed objects with minimal coding effort.

The WMI infrastructure, the heart of which is the Common Information Model (CIM) Object Manager (CIMOM), is the glue that binds management applications and providers. (CIM is described later in this chapter.) The infrastructure also serves as the object-class store and, in many cases, as the storage manager for persistent object properties. WMI implements the store, or repository, as an on-disk database named the CIMOM Object Repository. As part of its infrastructure, WMI supports several APIs through which management applications access object data and providers supply data and class definitions.

Windows programs use the WMI COM API, the primary management API, to directly interact with WMI. Other APIs layer on top of the COM API and include an Open Database Connectivity (ODBC) adapter for the Microsoft Access database application. A database developer uses the WMI ODBC adapter to embed references to object data in the developer's database. Then the developer can easily generate reports with database queries that contain WMI-based data. WMI ActiveX controls support another layered API. Web developers use the ActiveX controls to construct Web-based interfaces to WMI data. Another management API is the WMI scripting API, for use in script-based applications and Microsoft Visual Basic programs. WMI scripting support exists for all Microsoft programming language technologies.

As they are for management applications, WMI COM interfaces constitute the primary API for providers. However, unlike management applications, which are COM clients, providers are COM or Distributed COM (DCOM) servers (that is, the providers implement COM objects that WMI interacts with). Possible embodiments of a WMI provider include DLLs that load into WMI's manager process or stand-alone Windows applications or Windows services. Microsoft includes a number of built-in providers that present data from well-known sources, such as the Performance API, the registry, the Event Manager, Active Directory, SNMP, and Windows Driver Model (WDM) device drivers. The WMI SDK lets developers develop third-party WMI providers.

Providers

At the core of WBEM is the DMTF-designed CIM specification. The CIM specifies how management systems represent, from a systems management perspective, anything from a computer to an application or device on a computer. Provider developers use the CIM to represent the components that make up the parts of an application for which the developers want to enable management. Developers use the Managed Object Format (MOF) language to implement a CIM representation.

In addition to defining classes that represent objects, a provider must interface WMI to the objects. WMI classifies providers according to the interface features the providers supply. Table 4-10 lists WMI provider classifications. Note that a provider can implement one or more features; therefore, a provider can be, for example, both a class and an event provider. To clarify the feature definitions in Table 4-10, let's look at a provider that implements several of those features. The Event Log provider supports several objects, including an Event Log Computer, an Event Log Record, and an Event Log File. The Event Log is an Instance provider because it can define multiple instances for several of its classes. One class for which the Event Log provider defines multiple instances is the Event Log File class (Win32_NTEventlogFile); the Event Log provider defines an instance of this class for each of the system's event logs (that is, System Event Log, Application Event Log, and Security Event Log).

Table 4-10 **Provider Classifications**

Classification	Description
Class	Can supply, modify, delete, and enumerate a provider-specific class. Can also support query processing. Active Directory is a rare example of a service that is a class provider.
Instance	Can supply, modify, delete, and enumerate instances of system and provider-specific classes. An instance represents a managed object. Can also support query processing.
Property	Can supply and modify individual object property values.
Method	Supplies methods for a provider-specific class.
Event	Generates event notifications.
Event consumer	Maps a physical consumer to a logical consumer to support event notification.

The Event Log provider defines the instance data and lets management applications enumerate the records. To let management applications use WMI to back up and restore the Event Log files, the Event Log provider implements backup and restore methods for Event Log File objects. Doing so makes the Event Log provider a Method provider. Finally, a management application can register to receive notification whenever a new record writes to one of the Event Logs. Thus, the Event Log provider serves as an Event provider when it uses WMI event notification to tell WMI that Event Log records have arrived.

The Common Information Model and the Managed Object Format Language

The CIM follows in the steps of object-oriented languages such as C++ and Java, in which a modeler designs representations as classes. Working with classes lets developers use the powerful modeling techniques of inheritance and composition. Subclasses can inherit the attributes of a parent class, and they can add their own characteristics and override the characteristics they inherit from the parent class. A class that inherits properties from another class *derives* from that class. Classes also *compose*: a developer can build a class that includes other classes.

The DMTF provides multiple classes as part of the WBEM standard. These classes are CIM's basic language and represent objects that apply to all areas of management. The classes are part of the CIM *core model*. An example of a core class is CIM_ManagedSystemElement. This class contains a few basic properties that identify physical components such as hardware devices, and logical components such as processes and files. The properties include a caption, description, installation date, and status. Thus, the CIM_LogicalElement and CIM_PhysicalElement classes inherit the attributes of the CIM_ManagedSystemElement class. These two classes are also part of the CIM core model. The WBEM standard calls these classes *abstract* classes because they exist solely as classes that other classes inherit (that is, no object instances of an abstract class exist). You can therefore think of abstract classes as templates that define properties for use in other classes.

A second category of classes represents objects that are specific to management areas but independent of a particular implementation. These classes constitute the *common model* and are considered an extension of the core model. An example of a common-model class is the CIM_FileSystem class, which inherits the attributes of CIM_LogicalElement. Because virtually every operating system—including Windows, Linux, and other varieties of UNIX—rely on file-system-based structured storage, the CIM_FileSystem class is an appropriate constituent of the common model.

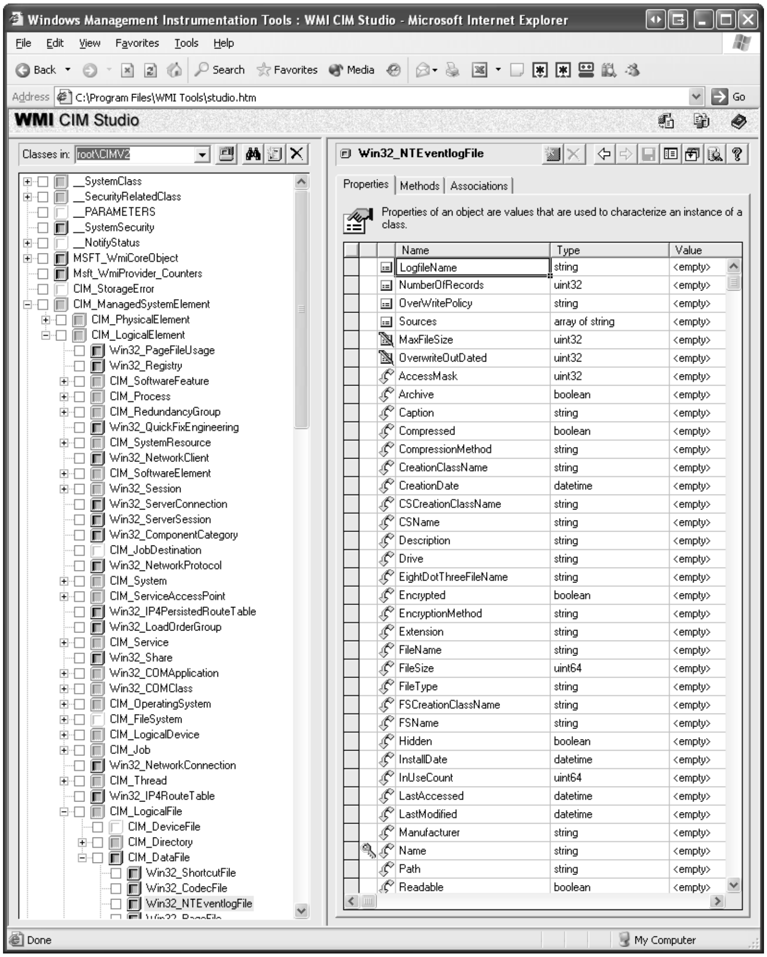
The final class category, the *extended model*, comprises technology-specific additions to the common model. Windows defines a large set of these classes to represent objects specific to the Windows environment. Because all operating systems store data in files, the CIM common model includes the CIM_LogicalFile class. The CIM_DataFile class inherits the CIM_LogicalFile class, and Windows adds the Win32_PageFile and Win32_ShortcutFile file classes for those Windows file types.

The Event Log provider makes extensive use of inheritance. Figure 4-20 shows a view of the WMI CIM Studio, a class browser that ships with the WMI Administrative Tools that you can obtain from the Microsoft download center at the Microsoft Web site. You can see where the Event Log provider relies on inheritance in the provider's Win32_NTEventlogFile class, which derives from CIM_DataFile. Event Log files are data files that have additional Event Log–specific attributes such as a log file name (LogfileName) and a count of the number of records that the file contains (NumberOfRecords). The tree that the class browser shows reveals that Win32_NTEventlogFile is based on several levels of inheritance, in which CIM_DataFile derives from CIM_LogicalFile, which derives from CIM_LogicalElement, and CIM_LogicalElement derives from CIM_ManagedSystemElement.

As stated earlier, WMI provider developers write their classes in the MOF language. The following output shows the definition of the Event Log provider's Win32_NTEventlogFile, which is selected in Figure 4-20. Notice the correlation between the properties that the right panel in Figure 4-20 lists and those properties' definitions in the MOF file below. CIM Studio uses yellow arrows to tag those properties that a class inherits. Thus, you don't see those properties specified in Win32_NTEventlogFile's definition.

```
dynamic: ToInstance, provider("MS_NT_EVENTLOG_PROVIDER"), Locale(1033), UUID("{8502C57B-
5FBB-11D2-AAC1-006008C78BC7}")]
class Win32_NTEventlogFile : CIM_DataFile
{
[read] string LogfileName;
[read, write] uint32 MaxFileSize;
[read] uint32 NumberOfRecords;
[read, volatile, ValueMap{"0", "1..365", "4294967295"}] string OverWritePolicy;
[read, write, Units("Days"), Range("0-365 | 4294967295")] uint32 OverwriteOutDated;
[read] string Sources[];
```

```
[implemented, Privileges{"SeSecurityPrivilege",
“SeBackupPrivilege"}] uint32 ClearEventlog([in] string ArchiveFileName);
[implemented, Privileges{"SeSecurityPrivilege",
“SeBackupPrivilege"}] uint32 BackupEventlog([in] string ArchiveFileName);
};
```



Properties of an object are values that are used to characterize an instance of a class.

Name	Type	Value
LogfileName	string	<empty>
NumberOfRecords	uint32	<empty>
OverWritePolicy	string	<empty>
Sources	array of string	<empty>
MaxFileSize	uint32	<empty>
OverwriteOutDated	uint32	<empty>
AccessMask	uint32	<empty>
Archive	boolean	<empty>
Caption	string	<empty>
Compressed	boolean	<empty>
CompressionMethod	string	<empty>
CreationClassName	string	<empty>
CreationDate	datetime	<empty>
CSCreationClassName	string	<empty>
CSName	string	<empty>
Description	string	<empty>
Drive	string	<empty>
EightDotThreeFileName	string	<empty>
Encrypted	boolean	<empty>
EncryptionMethod	string	<empty>
Extension	string	<empty>
FileName	string	<empty>
FileSize	uint64	<empty>
FileType	string	<empty>
FSCreationClassName	string	<empty>
FSName	string	<empty>
Hidden	boolean	<empty>
InstallDate	datetime	<empty>
InUseCount	uint64	<empty>
LastAccessed	datetime	<empty>
LastModified	datetime	<empty>
Manufacturer	string	<empty>
Name	string	<empty>
Path	string	<empty>
Readable	boolean	<empty>

Figure 4-20 WMI CIM Studio

One term worth reviewing is *dynamic*, which is a descriptive designator for the Win32_NTEventlogFile class that the MOF file in the preceding output shows. *Dynamic* means that the WMI infrastructure asks the WMI provider for the values of properties associated with an object of that class whenever a management application queries the object's properties. A *static* class is one in the WMI repository; the WMI infrastructure refers to the repository to obtain the values instead of asking a provider for the values. Because updating the repository is a relatively expensive operation, dynamic providers are more efficient for objects that have properties that change frequently.

EXPERIMENT: Viewing the MOF Definitions of WMI Classes

You can view the MOF definition for any WMI class by using the WbemTest tool that comes with Windows. In this experiment, we'll look at the MOF definition for the Win32_NTEventLogFile class:

1. Run Wbemtest from the Start menu's Run dialog box.
2. Click the Connect button, change the Namespace to root\cimv2, and connect.
3. Select Enum Classes, select the Recursive option button, and then click OK.
4. Find Win32_NTEventLogFile in the list classes, and double-click it to see its class properties.
5. Click the Show MOF button to open a window that displays the MOF text.

After constructing classes in MOF, WMI developers can supply the class definitions to WMI in several ways. WDM provider developers compile a MOF file into a binary MOF (BMF) file—a more compact binary representation than a MOF file—and give the BMF files to the WDM infrastructure. Another way is for the provider to compile the MOF and use WMI COM APIs to give the definitions to the WMI infrastructure. Finally, a provider can use the MOF Compiler (Mofcomp.exe) tool to give the WMI infrastructure a classes-compiled representation directly.

The WMI Namespace

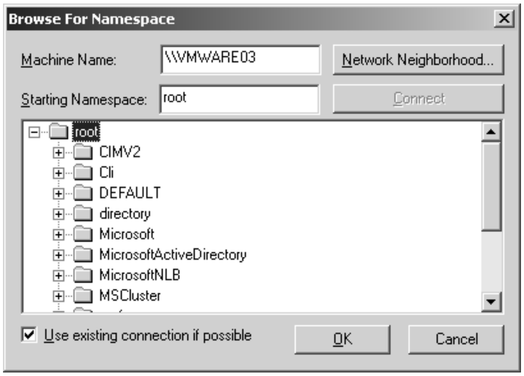
Classes define the properties of objects, and objects are class instances on a system. WMI uses a namespace that contains several subnamespaces that WMI arranges hierarchically to organize objects. A management application must connect to a namespace before the application can access objects within the namespace.

WMI names the namespace root directory *root*. All WMI installations have four predefined namespaces that reside beneath root: CIMV2, Default, Security, and WMI. Some of these namespaces have other namespaces within them. For example, CIMV2 includes the Applications and ms_409 namespaces as subnamespaces. Providers sometimes define their own namespaces; you can see the WMI namespace (which the Windows device driver WMI provider defines) beneath root in Windows.

EXPERIMENT: Viewing WMI Namespaces

You can see what namespaces are defined on a system with WMI CIM Studio. WMI CIM Studio presents a connection dialog box when you run it that includes a namespace browsing button to the right of the namespace edit box. Opening the browser and selecting a namespace has WMI CIM Studio connect to that namespace.

Windows Server 2003 defines over a dozen namespaces beneath root, some of which are visible here:



Unlike a file system namespace, which comprises a hierarchy of directories and files, a WMI namespace is only one level deep. Instead of using names as a file system does, WMI uses object properties that it defines as *keys* to identify the objects. Management applications specify class names with key names to locate specific objects within a namespace. Thus, each instance of a class must be uniquely identifiable by its key values. For example, the Event Log provider uses the Win32_NTLogEvent class to represent records in an Event Log. This class has two keys: Logfile, a string; and RecordNumber, an unsigned integer. A management application that queries WMI for instances of Event Log records obtains them from the provider key pairs that identify records. The application refers to a record using the syntax that you see in this sample object pathname:

```
\\DARYL\root\CIMV2:Win32_NTLogEvent.Logfile="Application",
                                        RecordNumber="1”
```

The first component in the name (\\DARYL) identifies the computer on which the object is located, and the second component (\root\CIMV2) is the namespace in which the object resides. The class name follows the colon, and key names and their associated values follow the period. A comma separates the key values.

WMI provides interfaces that let applications enumerate all the objects in a particular class or to make queries that return instances of a class that match a query criteria.

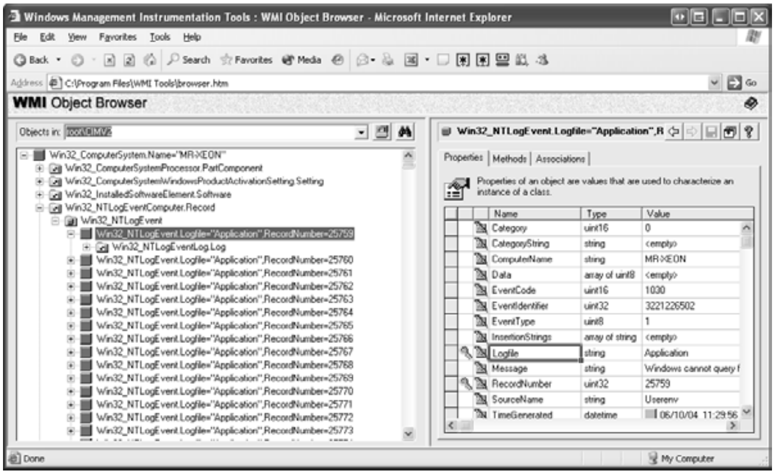
Class Association

Many object types are related to one another in some way. For example, a computer object has a processor, software, an operating system, active processes, and so on. WMI lets providers construct an *association class* to represent a logical connection between two different classes. Association classes associate one class with another, so the classes have only two properties: a class name and the *Ref* modifier. The following output shows an association in which the

Event Log provider's MOF file associates the Win32_NTLogEvent class with the Win32_ComputerSystem class. Given an object, a management application can query associated objects. In this way, a provider defines a hierarchy of objects.

```
[dynamic: ToInstance, provider("MS_NT_EVENTLOG_PROVIDER"): ToInstance, EnumPrivileges{"SeSec
urityPrivilege"}: ToSubClass, Locale(1033): ToInstance, UUID("{8502C57F-5FBB-11D2-AAC1-
006008C78BC7}"): ToInstance, Association: DisableOverride ToInstance ToSubClass]
class Win32_NTLogEventComputer
{
    [key, read: ToSubClass] Win32_ComputerSystem ref Computer;
    [key, read: ToSubClass] Win32_NTLogEvent ref Record;
};
```

Figure 4-21 shows the WMI Object Browser (another tool that the WMI Administrative Tools includes) displaying the contents of the CIMV2 namespace. Windows system components typically place their objects within the CIMV2 namespace. The Object Browser first locates the Win32_ComputerSystem object instance MR-XEON, which is the object that represents the computer. Then the Object Browser obtains the objects associated with Win32_ComputerSystem and displays them beneath MR-XEON. The Object Browser user interface displays association objects with a double-arrow folder icon. The associated class type's objects display beneath the folder.


Windows Management Instrumentation Tools : WMI Object Browser - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address C:\Program Files\WMI Tools\browser.htm
Go
WMI Object Browser
Objects in: root\CIMV2
Win32_NTLogEvent.Logfile="Application",R
Properties Methods Associations
Done
My Computer

```
Win32_ComputerSystem.Name="MR-XEON"
  Win32_ComputerSystemProcessor.PartComponent
  Win32_ComputerSystemWindowsProductActivationSetting.Setting
  Win32_InstalledSoftwareElement.Software
  Win32_NTLogEventComputer.Record
    Win32_NTLogEvent
      Win32_NTLogEvent.Logfile="Application",RecordNumber=25759
        Win32_NTLogEventLog.Log
      Win32_NTLogEvent.Logfile="Application",RecordNumber=25760
      Win32_NTLogEvent.Logfile="Application",RecordNumber=25761
      Win32_NTLogEvent.Logfile="Application",RecordNumber=25762
      Win32_NTLogEvent.Logfile="Application",RecordNumber=25763
      Win32_NTLogEvent.Logfile="Application",RecordNumber=25764
      Win32_NTLogEvent.Logfile="Application",RecordNumber=25765
      Win32_NTLogEvent.Logfile="Application",RecordNumber=25766
      Win32_NTLogEvent.Logfile="Application",RecordNumber=25767
      Win32_NTLogEvent.Logfile="Application",RecordNumber=25768
      Win32_NTLogEvent.Logfile="Application",RecordNumber=25769
      Win32_NTLogEvent.Logfile="Application",RecordNumber=25770
      Win32_NTLogEvent.Logfile="Application",RecordNumber=25771
      Win32_NTLogEvent.Logfile="Application",RecordNumber=25772
      Win32_NTLogEvent.Logfile="Application",RecordNumber=25773
```

Properties of an object are values that are used to characterize an instance of a class.

Name	Type	Value
Category	uint16	0
CategoryString	string	<empty>
ComputerName	string	MR-XEON
Data	array of uint8	<empty>
EventCode	uint16	1030
EventIdentifier	uint32	3221226502
EventType	uint8	1
InsertionStrings	array of string	<empty>
Logfile	string	Application
Message	string	Windows cannot query f
RecordNumber	uint32	25759
SourceName	string	Userenv
TimeGenerated	datetime	06/10/04 11:29:56

Figure 4-21 WMI Object Browser

You can see in the Object Browser that the Event Log provider's association class Win32_NTLogEventComputer is beneath MR-XEON and that numerous instances of the Win32_NTLogEvent class exist. Refer to the preceding output to verify that the MOF file defines the Win32_NTLogEventComputer class to associate the Win32_ComputerSystem class with the Win32_NTLogEvent class. Selecting an instance of Win32_NTLogEvent in the Object Browser reveals that class's properties under the Properties tab in the right-hand pane. Microsoft intended the Object Browser to help WMI developers examine their objects, but a management application would perform the same operations and display properties or collected information more intelligibly.

EXPERIMENT: Using WMI Scripts to Manage Systems

A powerful aspect of WMI is its support for scripting languages. Microsoft has generated hundreds of scripts that perform common administrative tasks for managing user accounts, files, the registry, processes, and hardware devices. While some scripts ship in the Windows Resource Kits, the Microsoft TechNet Scripting Center Web site serves as the central location for Microsoft scripts. Using a script from the scripting center is as easy as copying its text from your Internet browser, storing it in a file with with a .vbs extension, and running it with the command cscript **script**.vbs, where “script” is the name you gave the script. Cscript is the command-line interface to Windows Script Host (WSH).

Here’s a sample TechNet script that registers to receive events when Win32_Process object instances are created, which occurs whenever a process starts, and prints a line with the name of the process that the object represents:

```
strComputer = "."
Set objWMIService = GetObject("winmgmts:" _
    & "{impersonationLevel=impersonate}!\\" & strComputer & "\root\cimv2")
Set colMonitoredProcesses = objWMIService. _
    ExecNotificationQuery("select * from __instancecreationevent " _
        & " within 1 where TargetInstance isa 'Win32_Process'")
i = 0
Do While i = 0
    Set objLatestProcess = colMonitoredProcesses.NextEvent
    Wscript.Echo objLatestProcess.TargetInstance.Name
Loop
```

The line that invokes *ExecNotificationQuery* does so with a parameter that includes a “select” statement, which highlights WMI’s support for a read-only subset of the ANSI standard Structured Query Language (SQL), known as WQL, to provide a flexible way for WMI consumers to specify the information that they want to extract from WMI providers. Running the sample script with Cscript and then starting Notepad results in the following output:

```
C:\>cscript monproc.vbs
Microsoft (R) Windows Script Host Version 5.6
Copyright (C) Microsoft Corporation 1996-2001. All rights reserved.

NOTEPAD.EXE
```

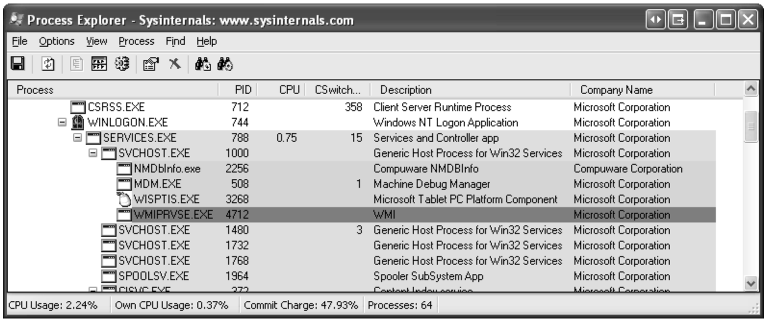
WMI Implementation

In Windows 2000, the WMI service is implemented in \Windows\System32\Winmgmt.exe, which the Windows SCM starts the first time a management application or WMI provider tries to access WMI APIs. In Windows XP and Windows Server 2003, the WMI service runs in a shared Svchost process that executes in the local system account.

In Windows 2000, WMI loads providers as in-process DCOM servers that execute within the Winmgmt service process. If a provider bug crashes the WMI process, the WMI service exits and then restarts in response to the next WMI request. Because the WMI service shares its Svchost process with several other services that would also exit if a WMI provider bug caused the process to exit, in Windows XP and Windows Server 2003, WMI loads providers into the Wmiprvse.exe provider-hosting process. Wmiprvse.exe launches as a child of the RPC service process. WMI executes Wmiprvse in the local system, local service, or network service accounts, depending on the value of the *HostingModel* property of the WMI Win32Provider object instance that represents the provider implementation. A Wmiprvse process exits after the provider is removed from the cache, one minute following the last provider request it receives.

EXPERIMENT: Viewing Wmiprvse Creation

You can see Wmiprvse being created by running Process Explorer from *www.sysinternals.com* and executing Wmic. A Wmiprvse process will appear beneath the Svchost process that hosts the RPC service. If Process Explorer job highlighting is enabled, it will appear with the job highlight color because, to prevent a runaway provider from consuming all virtual memory resources on a system, Wmiprvse executes in a job object that limits the number of child processes it can create and the amount of virtual memory each process and all the processes of the job can allocate. (See Chapter 6 for more information on job objects.)



Process	PID	CPU	CSwitch...	Description	Company Name
CSRSS.EXE	712		358	Client Server Runtime Process	Microsoft Corporation
WINLOGON.EXE	744			Windows NT Logon Application	Microsoft Corporation
SERVICES.EXE	788	0.75	15	Services and Controller app	Microsoft Corporation
SVCHOST.EXE	1000			Generic Host Process for Win32 Services	Microsoft Corporation
NMDbInfo.exe	2256			Compuware NMDBInfo	Compuware Corporation
MDM.EXE	508		1	Machine Debug Manager	Microsoft Corporation
WISPTIS.EXE	3268			Microsoft Tablet PC Platform Component	Microsoft Corporation
WMIPRVSE.EXE	4712			WMI	Microsoft Corporation
SVCHOST.EXE	1480		3	Generic Host Process for Win32 Services	Microsoft Corporation
SVCHOST.EXE	1732			Generic Host Process for Win32 Services	Microsoft Corporation
SVCHOST.EXE	1768			Generic Host Process for Win32 Services	Microsoft Corporation
SPOOLSV.EXE	1964			Spooler SubSystem App	Microsoft Corporation
CISVC.EXE	372			Content Index service	Microsoft Corporation

CPU Usage: 2.24% | Own CPU Usage: 0.37% | Commit Charge: 47.93% | Processes: 64

Most WMI components reside by default in \Windows\System32 and \Windows\System32\Wbem, including Windows MOF files, built-in provider DLLs, and management application WMI DLLs. Look in the \Windows\System32\Wbem directory, and you'll find Ntevt.mof, the Event Log provider MOF file. You'll also find Ntevt.dll, the Event Log provider's DLL, which the WMI service uses.

Directories beneath \Windows\System32\Wbem store the repository, log files, and third-party MOF files. WMI implements the repository—named the CIMOM repository—using a proprietary version of the Microsoft JET database engine. In Windows 2000, the database file stores in \Windows\System32\Wbem\Repository\Cim.rep; in Windows XP and Windows Server 2003, the database resides in \Windows\System32\Wbem\Repository\Fs.

WMI honors numerous registry settings (including various internal performance parameters such as CIMOM backup locations and intervals in Windows 2000) that the service's HKLM\SOFTWARE\Microsoft\WBEM\CIMOM registry key stores.

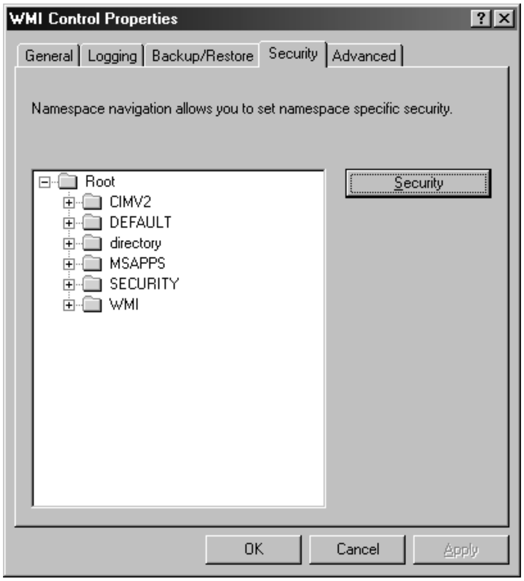
Device drivers use special interfaces to provide data to and accept commands—called the WMI System Control commands—from WMI. These interfaces are part of the WDM, which is explained in Chapter 9. Because the interfaces are cross-platform, they fall under the \root\WMI namespace.

WMIC

Windows XP and Windows Server 2003 include Wmic.exe, a utility that allows you to interact with WMI from a WMI-aware command-line shell. All WMI objects and their properties, including their methods, are accessible through the shell, which makes WMIC an advanced systems management console.

WMI Security

WMI implements security at the namespace level. If a management application successfully connects to a namespace, the application can view and access the properties of all the objects in that namespace. An administrator can use the WMI Control application to control which users can access a namespace. To start the WMI Control application, from the Start menu, select Programs, Administrative Tools, Computer Management. Next, open the Services And Applications branch. Right-click WMI Control, and select Properties to launch the WMI Control Properties dialog box, which Figure 4-22 shows. To configure security for namespaces, click the Security tab, select the namespace, and click Security. The other tabs in the WMI Control Properties dialog box let you modify the performance and backup settings that the registry stores.



Namespace navigation allows you to set namespace specific security.

Figure 4-22 WMI security properties

Conclusion

So far, we’ve examined the overall structure of Windows and the core system mechanisms on which the structure is built, and core management mechanisms. With this foundation laid, we're ready to explore the boot process and the individual executive components in more detail.

Chapter 5

Startup and Shutdown

In this chapter, we'll describe the steps required to boot Microsoft Windows and the options that can affect system startup. Understanding the details of the boot process will help you diagnose problems that can arise during a boot. Then we'll explain the kinds of things that can go wrong during the boot process and how to resolve them. Finally, we'll explain what occurs on an orderly system shutdown.

Boot Process

In describing the Windows boot process, we'll start with the installation of Windows and proceed through the execution of boot support files. Device drivers are a crucial part of the boot process, so we'll explain the way that they control the point in the boot process at which they load and initialize. Then we'll describe how the executive subsystems initialize and how the kernel launches the user-mode portion of Windows by starting the Session Manager process (Smss.exe), the Windows subsystem, and the logon process (Winlogon). Along the way, we'll highlight the points at which various text appears on the screen to help you correlate the internal process with what you see when you watch Windows boot.

The early phases of the boot process differ significantly on x86 and x64 systems versus IA64 systems. The next sections describe the portions of the boot process specific to x86 and x64 and follow with a section describing the IA64-specific portions of the boot process.

x86 and x64 Preboot

The Windows boot process doesn't begin when you power on your computer or press the reset button. It begins when you install Windows on your computer. At some point during the execution of the Windows Setup program, the system's primary hard disk is prepared with code that takes part in the boot process. Before we get into what this code does, let's look at how and where Windows places the code on a disk. Since the early days of MS-DOS, a standard has existed on x86 systems for the way physical hard disks are divided into volumes. Microsoft operating systems split hard disks into discrete areas known as *partitions* and use file systems (such as FAT and NTFS) to format each partition into a volume. A hard disk can contain up to four primary partitions. Because this apportioning scheme would limit a disk to four volumes, a special partition type, called an *extended partition*, further allocates up to four additional partitions within each primary partition. Extended partitions can contain extended partitions, which can contain extended partitions, and so on, making the number of volumes an operating system can place on a disk effectively infinite. Figure 5-1 shows an example of a hard disk layout, and Table 5-1 summarizes the files involved in the x86 and x64 boot process. (You can learn more about Windows partitioning in Chapter 10, which covers storage management.)

Table 5-1 x86 and x64 Boot Process Components

Component	Processor Execution	Responsibilities
Master Boot Record (MBR) code	16-bit real mode	Reads and loads partition boot sectors.
Boot sector	16-bit real mode	Reads the root directory to load Ntldr.
Ntldr	16-bit real mode and 32-bit or 64-bit protected mode; turns on paging	Reads Boot.ini, presents boot menu, and loads Ntoskrnl.exe, Bootvid.dll, Hal.dll, and boot-start device drivers. If a 32-bit installation is booted, switches to 32-bit protected mode; if a 64-bit installation is booted, switches to 64-bit long mode.
Ntdetect.com	16-bit real mode	Performs hardware detection for Ntldr.
Ntbootdd.sys	Protected mode	Device driver used for disk I/O on SCSI and Advanced Technology Attachment (ATA) systems where the BIOS is not used.
Ntoskrnl.exe	Protected mode with paging	Initializes executive subsystems and boot and system-start device drivers, prepares the system for running native applications, and runs Smss.exe.
Hal.dll	Protected mode with paging	Kernel-mode DLL that interfaces Ntoksnrl and drivers to the hardware.
Smss	Native application	Loads Windows subsystem, including Win32k.sys and Csrss.exe, and starts Winlogon process.
Winlogon	Native application	Starts the service control manager (SCM), starts the Local Security Subsystem (LSASS), and presents interactive logon dialog box.
Service control manager (SCM)	Native application	Loads and initializes auto-start device drivers and Windows services.

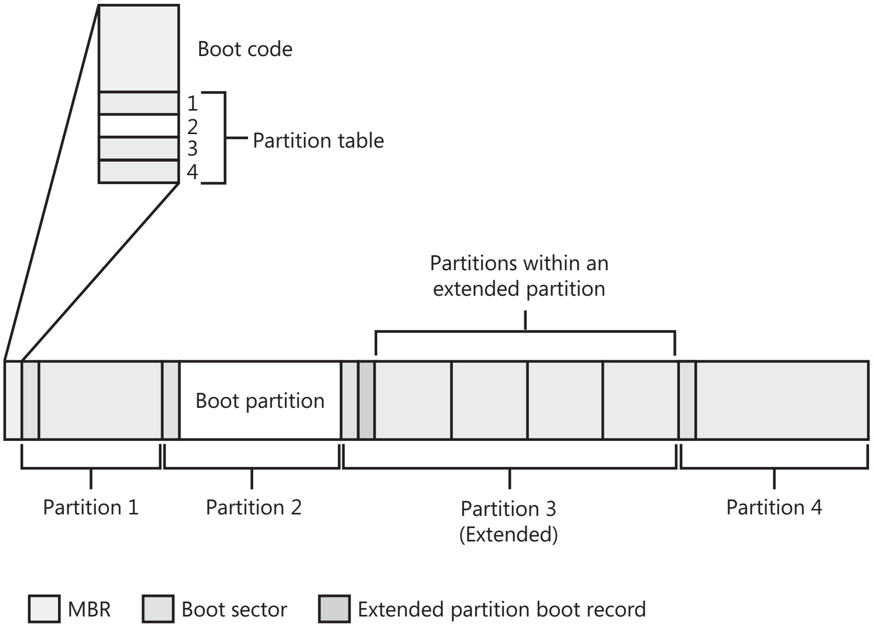


Figure 5-1 Sample hard disk layout

Physical disks are addressed in units known as *sectors*. A hard disk sector on an IBM-compatible PC is typically 512 bytes. Utilities that prepare hard disks for the definition of volumes, including the MS-DOS Fdisk utility or the Windows Setup program, write a sector of data called a Master Boot Record (MBR) to the first sector on a hard disk. (MBR partitioning is described in Chapter 10.) The MBR includes a fixed amount of space that contains executable instructions (called *boot code*) and a table (called a *partition table*) with four entries that define the locations of the primary partitions on the disk. When an IBM-compatible computer boots, the first code it executes is called the BIOS, which is encoded into the computer's ROM. The BIOS selects a boot device, reads that device's MBR into memory, and transfers control to the code in the MBR.

The MBRs written by Microsoft partitioning tools, such as the one integrated into Windows Setup and the Disk Management MMC snap-in, go through a similar process of reading and transferring control. First, an MBR's code scans the primary partition table until it locates a partition containing a flag that signals the partition is bootable. When the MBR finds at least one such flag, it reads the first sector from the flagged partition into memory and transfers control to code within the partition. This type of partition is called a *boot partition*, and the first sector of such a partition is called a *boot sector*. The volume defined for the boot partition is called the *system volume*.

Operating systems generally write boot sectors to disk without a user's involvement. For example, when Windows Setup writes the MBR to a hard disk, it also writes a boot sector to the first bootable partition of the disk. You might have created a MS-DOS boot sector during the installation of MS-DOS, Windows Me, Windows 98, or Windows 95. Windows Setup checks to see whether the boot sector it will overwrite with a Windows boot sector is a valid MS-DOS boot sector. If it is, Windows Setup copies the boot sector's contents to a file named Bootsect.dos in the root directory of the partition.

Before writing to a partition's boot sector, Windows Setup ensures that the partition is formatted with a file system that Windows supports (FAT, FAT32, or NTFS) by formatting the boot partition (and any other partition) with a file system type you specify. If partitions are already formatted, you can instruct Setup to skip this step. After Setup formats the boot partition, Setup copies the files Windows uses to the boot partition (the system volume), including two files that are part of the boot sequence, Ntldr and Ntdetect.com.

Another of Setup's roles is to create a boot menu file, Boot.ini, in the root directory of the system volume. This file contains options for starting the version of Windows that Setup installs and any preexisting Windows installations. If Bootsect.dos contains a valid MS-DOS boot sector, one of the entries Boot.ini creates is to boot into MS-DOS. The following output shows a sample Boot.ini file from a dual-boot computer on which MS-DOS is installed before Windows XP:

```
[boot loader]
timeout=30
default=multi(0)disk(0)rdisk(0)partition(1)\WINDOWS
[operating systems]
multi(0)disk(0)rdisk(0)partition(1) \WINDOWS="Microsoft Windows XP Professional" /fastdetect
C:\="Microsoft Windows"
```

You'll notice in the sample file that the path to the Windows directory is specified in a special syntax that conforms to the Advanced RISC Computing (ARC) naming convention. There are three variants to the syntax used by Windows. The first, shown in the preceding code sample, is the multi() syntax, which instructs Windows to use BIOS INT 13 functions to load system files. Thus, the multi() syntax is present when the disk on which the boot volume is located has a controller that provides INT-13 support. The multi() syntax follows this format:

multi(W)disk(X)rdisk(Y)partition(Z)

W is the disk controller number (also known as the ordinal number) and is typically 0. *X* is always 0 in the multi() syntax. *Y* specifies the physical hard disk attached to controller W. For ATA controllers, this number is typically between 0 and 3. For SCSI controllers, the number is typically between 0 and 15. *Z* indicates the partition number on the physical disk that corresponds to the boot volume. The first partition is assigned the number 1.

The scsi() ARC syntax informs Windows that it should rely on disk I/O services provided by Ntbootdd.sys (described shortly) to access the files on the boot volume. The format of the scsi() syntax is:

scsi(W)disk(X)rdisk(Y)partition(Z)

In this syntax, *W* is the controller number, and *X* is the physical hard disk attached to the controller and is typically between 0 and 15. *Y* specifies the SCSI logical unit number (LUN) of the disk that contains the boot volume and is typically 0. Finally, *Z* is the partition that corresponds to the boot volume with numbering starting at 1.

The final syntax used by Windows is the signature() syntax. This instructs Windows to locate the disk with the signature that matches the first value in parentheses, regardless of the controller number associated with the disk and to use Ntbootdd.sys to access the boot volume. A disk signature is a globally unique identifier (GUID) that Windows Setup extracts from information in the MBR and writes to the disk. The signature() syntax is as follows:

signature(V)disk(X)rdisk(Y)partition(Z)

V is a 32-bit hexadecimal disk signature that identifies the disk. *X* is the physical hard disk with the specific signature, and it can be attached to any controller on the system. *Y* is always 0, and *Z* is the partition number on which the boot volume is located.

Windows uses the signature() syntax in the following cases:

- The boot volume is larger than 7.8 GB in size, and BIOS extended INT-13 functions (those used to access parts of a disk beyond 7.8 GB) cannot access the entire volume.
- The BIOS does not support extended INT-13 functions.

The x86/x64 Boot Sector and Ntldr

Setup must know the partition format before it writes a boot sector because the contents of the boot sector vary depending on the format. For example, if the boot partition is a FAT partition, Windows writes code to the boot sector that understands the FAT file system. But if the partition is in NTFS format, Windows writes NTFS-capable code. The role of the boot-sector code is to give Windows information about the structure and format of a volume and to read in the Ntldr file from the root directory of the volume. Thus, the boot-sector code contains just enough read-only file system code to accomplish this task. After the boot-sector code loads Ntldr into memory, it transfers control to Ntldr's entry point. If the boot-sector code can't find Ntldr in the volume's root directory, it displays the error message "BOOT: Couldn't find NTLDRP" if the boot file system is FAT or "NTLDR is missing" if the file system is NTFS.

Ntldr begins its existence while a system is executing in an x86 operating mode called *real mode*. In real mode, no virtual-to-physical translation of memory addresses occurs, which means that programs that use the memory addresses interpret them as physical addresses and that only the first 1 MB of the computer's physical memory is accessible. Simple MS-DOS programs execute in a real-mode environment. However, the first action Ntldr takes is to switch the system to *protected mode*. Still no virtual-to-physical translation occurs at this point in the boot process, but a full 32 bits of memory becomes accessible. After the system is in protected mode, Ntldr can access all of physical memory. After creating enough page tables to make memory below 16 MB accessible with paging turned on, Ntldr enables paging. Protected mode with paging enabled is the mode in which Windows executes in normal operation.

After Ntldr enables paging, it is fully operational. However, it still relies on functions supplied by the boot code to access IDE-based system and boot disks as well as the display. The boot-code functions briefly switch off paging and switch the processor back to a mode in which services provided by the BIOS can be executed. If the disk containing the boot volume is SCSI-based and is not accessible using BIOS firmware support, Ntldr loads a file named Ntbootdd.sys and uses it instead of the boot-code functions for disk access. Ntbootdd.sys is a copy of the SCSI miniport driver that Windows uses when its fully operation to access the boot disk. (See Chapter 10 for more information on disk drivers.) Ntldr next reads the Boot.ini file from the root directory using built-in file system code. Like the boot sector's code, Ntldr contains read-only NTFS and FAT code; unlike the boot sector's code, however, Ntldr's file system code can read subdirectories.

Ntldr next clears the screen. If there is a valid Hiberfil.sys file in the root of the system volume, it shortcuts the boot process by reading the contents of the file into memory and transferring control to code in the kernel that resumes a hibernated system. That code is responsible for restarting drivers that were active when the system was shut down. Hiberfil.sys will be valid only if the last time the computer was shut down it was hibernated. (See the section "The Power Manager" in Chapter 11 for information on hibernation.)

If there is more than one boot-selection entry in Boot.ini, it presents the user with the boot-selection menu. (If there is only one entry, Ntldr bypasses the menu and proceeds to displaying the startup progress bar.) Selection entries in Boot.ini direct Ntldr to the partition on which the Windows system directory (typically \Windows) of the selected installation resides. This partition might be the same as the boot partition, or it might be another primary partition.

If the Boot.ini entry refers to an MS-DOS installation (that is, by referring to C:\ as the system partition), Ntldr reads the contents of the Bootsect.dos file into memory, switches back to 16-bit real mode, and calls the MBR code in Bootsect.dos. This action causes the Bootsect.dos code to execute as if the MBR had read the code from disk. Code in Bootsect.dos continues an MS-DOS-specific boot, such as is used to boot Microsoft Windows Me, Windows 98, or Windows 95 on a computer on which these operating systems are installed with Windows.

Entries in Boot.ini can include optional arguments that Ntldr and other components involved in the boot process interpret. Table 5-2 contains a complete list of these options and their effects. The Bootcfg.exe tool, introduced in Windows XP, provides a convenient interface to setting a number of the switches. Any options that are included on the Boot.ini save to the Registry value HKLM\System\CurrentControlSet\Control\SystemStartOptions.

Table 5-2 **Boot Options**

Boot Qualifier	Meaning
/3GB	Increases the size of the user process address space from 2 GB to 3 GB (and therefore reduces the size of system space from 2 GB to 1 GB). Giving virtual-memory-intensive applications such as database servers a larger address space can improve their performance. For an application to take advantage of this feature, however, two additional conditions must be met: the system must be running Windows XP, Windows Server 2003, Windows 2000 Advanced Server, or Datacenter Server; and the application .exe must be flagged as a 3-GB-aware application (applies to 32-bit systems only). (See the section "Address Space Layout" in Chapter 7 for more information.)
/BASEVIDEO	Causes Windows to use the standard VGA display driver for GUI-mode operations.
/BAUDRATE=	Enables kernel-mode debugging, and specifies an override for the default baud rate (19200) at which a remote kernel debugger host will connect. Example: /BAUDRATE=115200.
/BOOTLOG	Causes Windows to write a log of the boot to the file %SystemRoot%\Ntbtlog.txt.
/BOOTLOGO	Use this switch to have Windows XP or Windows Server 2003 display an installable splash screen instead of the standard splash screen. First, create a 16-color (any 16 colors) 640x480 bitmap and save it in the Windows directory with the name Boot.bmp. Then add "/bootlogo /noguiboot" to the boot.ini selection.
/BREAK	Causes the hardware abstraction layer (HAL) to stop at a breakpoint at HAL initialization. The first thing the Windows kernel does when it initializes is to initialize the HAL, so this breakpoint is the earliest one possible. The HAL will wait indefinitely at the breakpoint until a kernel-debugger connection is made. If the switch is used without the /DEBUG switch, the system will elicit a blue screen with a STOP code of 0x00000078 (PHASE0_ EXCEPTION).
/BURNMEMORY=	Specifies an amount of memory Windows can't use (similar to the /MAXMEM switch). The value is specified in megabytes. Example: /BURNMEMORY=128 would indicate that Windows can't use 128 MB of the total physical memory on the machine.

Table 5-2 **Boot Options**

Boot Qualifier	Meaning
/CHANNEL=	Used in conjunction with /DEBUGPORT=1394 to specify the IEEE 1394 channel through which kernel debugging communications will flow. This can be any number between 0 and 62 and defaults to 0 if not set.
/CLKLVL	Causes the standard x86 multiprocessor HAL (Halmps.dll) to configure itself for a level-sensitive system clock rather than an edge-triggered clock. *Level-sensitive* and *edge-triggered* are terms used to describe hardware interrupt types.
/CMDCONS	Passed when booting into the Recovery Console (described later in this chapter).
/CRASHDEBUG	Causes the kernel debugger to be loaded when the system boots, but to remain inactive unless a crash occurs. This allows the serial port that the kernel debugger would use to be available for use by the system until the system crashes (vs. /DEBUG, which causes the kernel debugger to use the serial port for the life of the system session).
/DEBUG	Enables kernel-mode debugging.
/DEBUGPORT=	Enables kernel-mode debugging, and specifies an override for the default serial (usually COM2 on systems with at least two serial ports) to which a remote kernel-debugger host is connected. Windows XP and Windows Server 2003 also support debugging through accepted IEEE 1394 ports. Examples: /DEBUGPORT=COM2, /DEBUGPORT=1394.
/EXECUTE	Disables no-execute protection. See the /NOEXECUTE switch for more information.
/FASTDETECT	Default boot option for Windows. Replaces the Windows NT 4 switch /NOSERIALMICE. The reason the qualifier exists (vs. just having NTDETECT perform this operation by default) is so that NTDETECT can support booting Windows NT 4. Windows Plug and Play device drivers perform detection of parallel and serial devices, but Windows NT 4 expects NTDETECT to perform the detection. Thus, specifying /FASTDETECT causes NTDETECT to skip parallel and serial device enumeration (actions that are not required when booting Windows), whereas omitting the switch causes NTDETECT to perform this enumeration (which is required for booting Windows NT 4).
/INTAFFINITY	Directs the standard x86 multiprocessor HAL (Halmps.dll) to set interrupt affinities such that only the highest numbered processor will receive interrupts. Without the switch, the HAL defaults to its normal behavior of letting all processors receive interrupts.

Table 5-2 **Boot Options**

Boot Qualifier	Meaning
/KERNEL= /HAL=	Enables you to override Ntldr's default filename for the kernel image (Ntoskrnl.exe) and/or the HAL (Hal.dll). These options are useful for alternating between a checked kernel environment and a free (retail) kernel environment or even to manually select a different HAL. If you want to boot a checked environment that consists solely of the checked kernel and HAL, which is typically all that is needed to test drivers, follow these steps on a system installed with the free build: **1.** Copy the checked versions of the kernel images from the checked build CD to your \Windows\System32 directory, giving the images different names than the default. For example, if you're on a uniprocessor, copy Ntoskrnl.exe to Ntoschk.exe and Ntkrnlpa.exe to Ntoschkpa.exe. If you're on a multiprocessor, copy Ntkrnlmp.exe to Ntoschk.exe and Ntkrpamp.exe to Ntoschkpa.exe. The kernel filename must be an 8.3-style short name. **2.** Copy the checked version of the appropriate HAL needed for your system from \I386\Driver.cab on the checked build CD to your \Windows\System32 directory, naming it Halchk.dll. To determine which HAL to copy, open \Windows\Repair\Setup.log and search for Hal.dll; you'll find a line like \WINDOWS\system32\ hal.dll="halacpi.dll","1d8a1". The name immediately to the right of the equals sign is the name of the HAL you should copy. The HAL filename must be an 8.3-style short name. **3.** Make a copy of the default line in the system's Boot.ini file. **4.** In the string description of the boot selection, add something that indicates that the new selection will be for a checked build environment (for example, "Windows XP Professional Checked"). **5.** Add the following to the end of the new selection's line: /KERNEL=NTOSCHK.EXE /HAL= HALCHK.DLL Now when the selection menu appears during the boot process, you can select the new entry to boot a checked environment or select the entry you were using to boot the free build.
/LASTKNOWNGOOD	Causes the system to boot as if the LastKnownGood boot option was selected.
/MAXMEM=	Limits Windows to ignore (not use) physical memory beyond the amount indicated. The number is interpreted in megabytes. Example: /MAXMEM=32 would limit the system to using the first 32 MB of physical memory even if more were present.

Table 5-2 **Boot Options**

Boot Qualifier	Meaning
/MAXPROCSPERCLUSTER=	For the standard x86 multiprocessor HAL (Halmps.dll), forces cluster-mode Advanced Programmable Interrupt Controller (APIC) addressing (not supported on systems with an 82489DX external APIC interrupt controller).
/MININT	This option is used by Windows PE (Preinstallation Environment) and causes the Configuration Manager to load the Registry SYSTEM hive as a volatile hive such that changes made to it in memory are not saved back to the hive image.
/NODEBUG	Prevents kernel-mode debugging from being initialized. Overrides the specification of any of the three debug-related switches, /DEBUG, /DEBUGPORT, and /BAUDRATE.
/NOEXECUTE	This option is available only on 32-bit versions of Windows when running on AMD64 processors and only when PAE (explained further in the /PAE switch entry) is also enabled. It enables no-execute protection, which results in the Memory Manager marking pages containing data as no-execute so that they cannot be executed as code. This can be useful for preventing malicious code from exploiting buffer overflow bugs with unexpected program input in order to execute arbitrary code. No-execute protection is always enabled on 64-bit versions of Windows on AMD64 processors. There are 4 modifiers that can be specified to the /NOEXECUTE switch: =OPTIN,=OPTOUT,=ALWAYSON,=ALWAYSOFF. See Chapter 7 for a description of their behavior.
/NOGUIBOOT	Instructs Windows not to initialize the VGA video driver responsible for presenting bitmapped graphics during the boot process. The driver is used to display boot progress information, so disabling it will disable the ability of Windows to show this information.
/NOLOWMEM	Requires that the /PAE switch be present and that the system have more than 4 GB of physical memory. If these conditions are met, the PAE-enabled version of the Windows kernel, Ntkrnlpa.exe, won’t use the first 4 GB of physical memory. Instead, it will load all applications and device drivers, and allocate all memory pools, from above that boundary. This switch is useful only to test device-driver compatibility with large memory systems.
/NOPAE	Forces Ntldr to load the non–Physical Address Extension (PAE) version of the Windows kernel, even if the system is detected as supporting x86 PAEs and has more than 4 GB of physical memory.
/NOSERIALMICE=[COMx | COMx,y,z...]	Obsolete Windows NT 4 qualifier—replaced by the absence of the /FASTDETECT switch. Disables serial mouse detection of the specified COM ports. This switch was used if you had a device other than a mouse attached to a serial port during the startup sequence. Using /NOSERIALMICE without specifying a COM port disables serial mouse detection on all COM ports. See Microsoft Knowledge Base article Q131976 for more information.

Table 5-2 Boot Options

Boot Qualifier	Meaning
/NUMPROC=	Specifies the number of CPUs that can be used on a multiprocessor system. Example: /NUMPROC=2 on a four-way system will prevent Windows from using two of the four processors.
/ONECPU	Causes Windows to use only one CPU on a multiprocessor system.
/PAE	Causes Ntldr to load Ntkrnlpa.exe, which is the version of the x86 kernel that is able to take advantage of x86 PAEs. The PAE version of the kernel presents 64-bit physical addresses to device drivers, so this switch is helpful for testing device driver support for large memory systems.
/PCILOCK	Stops Windows from dynamically assigning IO/IRQ resources to PCI devices and leaves the devices configured by the BIOS. See Microsoft Knowledge Base article Q148501 for more information.
/RDPATH=	Specifies the path to a System Disk Image (SDI) file, which can be on the network, that the system will use to boot from. Often used in conjunction with the /RDIMAGEOFFSET= flag to indicate to NTLDR where in the file the system image starts.
/REDIRECT	Introduced with Windows Server 2003. Used to cause Windows to enable Emergency Management Services (EMS), which reports boot information and accepts system management commands through a serial port. Specify serial port and baudrate used in conjunction with EMS with redirect= and redirectbaudrate= lines in the [boot loader] section of the Boot.ini file.
/SAFEBOOT:	Specifies options for a safe mode boot. You should never have to specify this option manually because Ntldr specifies it for you when you use the F8 menu to perform a safe mode boot. (A safe mode boot is a boot in which Windows loads only drivers and services that are specified by name or group under the Minimal or Network registry keys under HKLM\SYSTEM\CurrentControlSet\Control\SafeBoot.) Following the colon in the option, you must specify one of three additional switches: MINIMAL, NETWORK, or DSREPAIR. The MINIMAL and NETWORK flags correspond to safe mode boot with no network and safe mode boot with network support, respectively. The DSREPAIR (Directory Services Repair) switch causes Windows to boot into a mode in which the Active Directory directory service is offline and the active directory database unopened. This allows an administrator to perform diagnostic, repair, or restore functions on the database. An additional option you can append is (ALTERNATESHELL), which tells Windows to use the program specified by the HKLM\SYSTEM\CurrentControlSet\SafeBoot\AlternateShell value as the graphical shell rather than to use the default, which is Windows Explorer.

Table 5-2 **Boot Options**

Boot Qualifier	Meaning
/SCSIORDINAL:	Directs Windows to the SCSI ID of the controller. (Adding a new SCSI device to a system with an on-board SCSI controller can cause the controller's SCSI ID to change.) See Microsoft Knowledge Base article Q103625 for more information.
/SDIBOOT=	Used in Windows XP Embedded systems to have Windows boot from a RAM disk image stored in the specified System Disk Image (SDI) file.
/SOS	Causes Windows to list the device drivers marked to load at boot time and then to display the system version number (including the build number), amount of physical memory, and number of processors.
/TIMERES=	Sets the resolution of the system timer on the standard x86 multiprocessor HAL (Halmps.dll). The argument is a number interpreted in hundreds of nanoseconds, but the rate is set to the closest resolution the HAL supports that isn't larger than the one requested. The HAL supports the following resolutions: **Hundreds of nanoseconds** — **Milliseconds (ms)** 97660.98 — 195322.00 390633.90 — 781257.80 The default resolution is 7.8 ms. The system timer resolution affects the resolution of waitable timers. Example: /TIMERES=21000 would set the timer to a resolution of 2.0 ms.
/USERVA=	This switch is supported only on Windows XP and Windows Server 2003. Like the /3GB switch, this switch gives applications a larger address space. Specify the amount in MB between 2048 and 3072. This switch has the same application requirements as the /3GB switch and requires that the /3GB switch also be present (applies to 32-bit systems only).
/WIN95	Directs Ntldr to boot the Consumer Windows boot sector stored in Bootsect.w40. This switch is pertinent only on a triple-boot system that has MS-DOS, Consumer Windows, and Windows installed. See Microsoft Knowledge Base article Q157992 for more information.
/WIN95DOS	Directs Ntldr to boot the Consumer Windows boot sector stored in Bootsect.w40. This switch is pertinent only on a triple-boot system that has MS-DOS, Consumer Windows, and Windows installed. See Microsoft Knowledge Base article Q157992 for more information.
/YEAR=	Instructs the Windows core time function to ignore the year that the computer's real-time clock reports and instead use the one indicated. Thus, the year used in the switch affects every piece of software on the system, including the Windows kernel. Example: /YEAR=2001. (This switch was created to assist in Y2K testing.)

If the user doesn't select an entry from the selection menu within the timeout period the Boot.ini file specifies, Ntldr chooses the default selection, which is the top-most entry in boot.ini with a path matching the path specified in the "default=" line. Once the boot selection has been made, Ntldr loads and executes Ntdetect.com, a 16-bit real-mode program that uses a system's BIOS to query the computer for basic device and configuration information. This information includes the following:

- The time and date information stored in the system's CMOS (nonvolatile memory)
- The types of buses (for example, ISA, PCI, EISA, Micro Channel Architecture [MCA]) on the system and identifiers for devices attached to the buses
- The number, size, and type of disk drives on the system
- The types of mouse input devices connected to the system
- The number and type of parallel ports configured on the system
- The types of video adapters present on the system

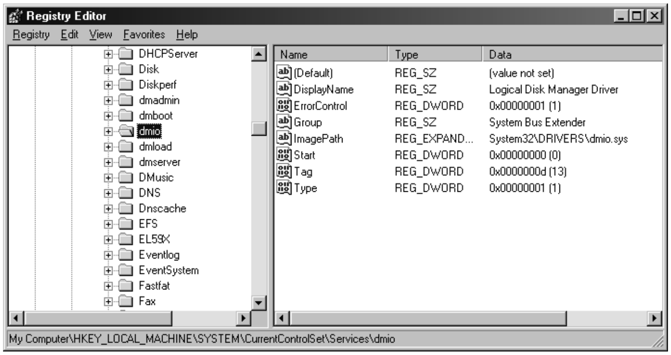
This information is gathered into internal data structures that will be stored under the HKLM\HARDWARE\DESCRIPTION registry key later in the boot.

On Windows 2000, Ntldr then clears the screen and displays the "Starting Windows" progress bar. This progress bar remains empty until Ntldr begins loading boot drivers. (See step 5 in the following list.) Below the progress bar is the message, "For troubleshooting and advanced startup options for Windows, press F8." If the user presses F8, the advanced boot menu is presented, which allows the user to select such options as booting from last known good, safe mode, debug mode, and so on. On Windows XP and Windows Server 2003, Ntldr presents a logo splash screen instead of a progress bar.

If Ntldr is running on an x64 system and the kernel specified by the entry selected in the boot menu is for x64, Ntldr switches the processor to *long mode*, where the native word size is 64-bits. Next, Ntldr begins loading the files from the *boot volume* needed to start the kernel initialization. The boot volume is the volume that corresponds to the partition on which the system directory (usually \Windows) of the installation being booted is located. The steps Ntldr follows here include:

1. Loads the appropriate kernel and HAL images (Ntoskrnl.exe and Hal.dll by default). If Ntldr fails to load either of these files, it prints the message "Windows could not start because the following file was missing or corrupt", followed by the name of the file.
2. Reads in the SYSTEM registry hive, \Windows\System32\Config\System, so that it can determine which device drivers need to be loaded to accomplish the boot. (A hive is a file that contains a registry subtree. You'll find more details about the registry in Chapter 4)
3. Scans the in-memory SYSTEM registry hive and locates all the *boot device drivers*. Boot device drivers are drivers necessary to boot the system. These drivers are indicated in the registry by a start value of SERVICE_BOOT_START (0). Every device driver has a registry subkey under HKLM\SYSTEM\CurrentControlSet\Services. For example,

Services has a subkey named Dmio for the Logical Disk Manager driver, which you can see in Figure 5-2. (For a detailed description of the Services registry entries, see the section “Services” in Chapter 4)



Name	Type	Data
(Default)	REG_SZ	(value not set)
DisplayName	REG_SZ	Logical Disk Manager Driver
ErrorControl	REG_DWORD	0x00000001 (1)
Group	REG_SZ	System Bus Extender
ImagePath	REG_EXPAND...	System32\DRIVERS\dmio.sys
Start	REG_DWORD	0x00000000 (0)
Tag	REG_DWORD	0x0000000d (13)
Type	REG_DWORD	0x00000001 (1)

My Computer\HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\dmio

Figure 5-2 Logical Disk Manager driver service settings

4. Adds the file system driver that’s responsible for implementing the code for the type of partition (FAT, FAT32, or NTFS) on which the installation directory resides to the list of boot drivers to load. Ntldr must load this driver at this time; if it didn’t, the kernel would require the drivers to load themselves, a requirement that would introduce a circular dependency.
5. Loads the boot drivers, which should only be drivers that, like the file system driver for the boot volume, would introduce a circular dependency if the kernel was required to load them. To indicate the progress of the loading, Ntldr updates a progress bar displayed below the text “Starting Windows”. The progress bar moves for each driver loaded. (It assumes there are 80 boot device drivers—each successful load moves the progress bar by 1.25 percent.) If the /SOS switch is specified in the Boot.ini selection, Ntldr doesn’t display the progress bar but instead displays the filenames of each boot driver. Keep in mind that the drivers are loaded but not initialized at this time—they initialize later in the boot sequence.
6. Prepares CPU registers for the execution of Ntoskrnl.exe.

This action is the end of Ntldr’s role in the boot process. At this point, Ntldr calls the main function in Ntoskrnl.exe to perform the rest of the system initialization.

The IA64 Boot Process

Table 5-3 lists the files involved in the IA64 boot process. IA64 systems conform to the Extensible Firmware Interface (EFI) specification as defined by Intel. An EFI-compliant system has firmware that runs boot loader code that’s been programmed into the system’s nonvolatile RAM (NVRAM) by Windows Setup. The boot code reads the IA64-equivalent of the x86 and x64 Boot.ini contents, which are also stored in NVRAM. Both Microsoft EFI tools runnable in the EFI console and Bootcfg.exe, a tool included with Windows, allow for modification of the NVRAM boot selections and switches.

Hardware detection occurs next, where the boot loader uses EFI interfaces to determine the number and type of the following devices:

- Network adapters
- Video adapters
- Keyboards
- Disk controllers
- Storage devices

Just as Ntldr does on x86 and x64 systems, the boot loader then presents a menu of boot selections with an optional timeout. Once a boot selection is made, the loader navigates to the subdirectory on the EFI System partition corresponding to the selection and loads several other files required to continue the boot: Fpswa.efi and Ia64ldr.efi. The EFI specification requires that the system have a partition designated as the EFI System partition that is formatted with the FAT file system and that is between 100 MB and 1 GB in size or up to one percent of the size of the disk, and each Windows installation has a subdirectory on the EFI System partition under EFI\Microsoft. The first installation is assigned the folder Winnt50, the second Winnt50.1, and each subsequent installation has a unique index number following the period in the folder name. Ia64ldr.efi is responsible for loading Ntoskrnl.exe, Hal.dll, and the boot-start drivers, after which the boot proceeds through the same steps as for x86 and x64.

Table 5-3 IA64 Boot Process Components

Component	Location	Responsibilities
Fpswa.efi	EFI\Microsoft\Winnt50.x on the EFI System partition	A file that contains support for EFI floating-point operations
Ia64ldr.efi	EFI\Microsoft\Winnt50.x on the EFI System partition	Loads Ntoskrnl.exe, Hal.dll, and boot drivers
Ntoskrnl.exe	\Windows\System32	Initializes executive subsystems and boot and system-start device drivers, prepares the system for running native applications, and runs Smss.exe
Hal.dll	\Windows\System32	Kernel-mode DLL that interfaces Ntoksnrl and drivers to the hardware
Service control manager (SCM)	\Windows\System32	Loads and initializes auto-start device drivers and Windows services
Smss	\Windows\System32	Loads Windows subsystem, including Win32k.sys and Csrss.exe, and starts Winlogon process
Winlogon	\Windows\System32	Starts the service control manager (SCM), the Local Security Authority Subsystem (LSASS), and presents the interactive logon dialog box

Initializing the Kernel and Executive Subsystems

When Ntldr calls Ntoskrnl, it passes a data structure that contains a copy of the line in Boot.ini that represents the selected menu option for this boot, a pointer to the memory tables Ntldr generated to describe the physical memory on the system, a pointer to the in-memory copy of the HARDWARE and SYSTEM registry hives, and a pointer to the list of boot drivers Ntldr loaded.

Ntoskrnl then begins the first of its two-phase initialization process, called *phase 0* and *phase 1*. Most executive subsystems have an initialization function that takes a parameter that identifies which phase is executing.

During phase 0, interrupts are disabled. The purpose of this phase is to build the rudimentary structures required to allow the services needed in phase 1 to be invoked. Ntoskrnl's main function calls *KiSystemStartup*, which in turn calls *HalInitializeProcessor* and *KiInitializeKernel* for each CPU. *KiInitializeKernel*, if running on the boot CPU, performs systemwide kernel initialization, such as initializing internal listheads and other data structures that all CPUs share. Each instance of *KiInitializeKernel* then calls the function responsible for orchestrating phase 0, *ExpInitializeExecutive*.

ExpInitializeExecutive starts by calling the HAL function *HalInitSystem*, which gives the HAL a chance to gain system control before Windows performs significant further initialization. One responsibility of *HalInitSystem* is to prepare the system interrupt controller of each CPU for interrupts and to configure the interval clock timer interrupt, which is used for CPU time accounting. (See the section "Quantum Accounting" in Chapter 6 for more on CPU time accounting.)

Only on the boot processor does *ExpInitializeExecutive* perform initialization other than calling *HalInitSystem*. When *HalInitSystem* returns control, *ExpInitializeExecutive* on the boot CPU proceeds by processing the /BURNMEMORY Boot.ini switch (if the switch is present in the line from the Boot.ini file that corresponds to the menu selection the user made when choosing which installation to boot) and discarding the amount of memory the switch specifies.

Next, *ExpInitializeExecutive* calls the phase 0 initialization routines for the memory manager, object manager, security reference monitor, process manager, and Plug and Play manager. These components perform the following initialization steps:

1. The memory manager constructs page tables and internal data structures that are necessary to provide basic memory services. The memory manager also builds and reserves an area for the system file cache and creates memory areas for the paged and nonpaged pools. The other executive subsystems, the kernel, and the device drivers use these two memory pools for allocating their data structures.
2. During the object manager initialization, the objects that are necessary to construct the object manager namespace are defined so that other subsystems can insert objects into it. A handle table is created so that resource tracking can begin.

3. The security reference monitor initializes the token type object and then uses the object to create and prepare the first local system account token for assignment to the initial process. (See Chapter 8 for a description of the local system account.)
4. The process manager performs most of its initialization in phase 0, defining the process and thread object types and setting up lists to track active processes and threads. The process manager also creates a process object for the initial process and names it *Idle*. As its last step, the process manager creates the System process and a system thread to execute the routine *Phase1Initialization*. This thread doesn't start running right away because interrupts are still disabled.
5. The Plug and Play manager's phase 0 initialization then takes place, which involves simply initializing an executive resource used to synchronize bus resources.

When control returns to the *KiInitializeKernel* function on each processor, control proceeds to the *Idle* loop, which then causes the system thread created in step 4 of the previous process description to begin executing phase 1. (Secondary processors wait to begin their initialization until step 5 of phase 1, described in the following list.) Phase 1 consists of the following steps. The boot splash screen of Windows 2000 systems includes a progress bar, and the steps at which the progress bar on the screen is updated are included in this list:

1. *HalInitSystem* is called to prepare the system to accept interrupts from devices and to enable interrupts.
2. The boot video driver (\Windows\System32\Bootvid.dll) is called, which in turn displays the Windows startup screen. (On Windows XP and Windows Server 2003 systems, the driver presents the same graphic that Ntldr placed on the screen earlier in the boot.)
3. The power manager's initialization is called.
4. The system time is initialized (by calling *HalQueryRealTimeClock*) and then stored as the time the system booted.
5. On a multiprocessor system, the remaining processors are initialized and execution starts.
6. The progress bar is set to 5 percent.
7. The object manager creates the namespace root directory (\), \ObjectTypes directory, and the DOS device name mapping directory (\?? on Windows 2000, and \Global?? on Windows XP and Windows Server 2003). It then creates the \DosDevices symbolic link that points at the DOS device name mapping directory.
8. The executive is called to create the executive object types, including semaphore, mutex, event, and timer.
9. The kernel initializes scheduler (dispatcher) data structures and the system service dispatch table.

10. The security reference monitor creates the \Security directory in the object manager namespace and initializes auditing data structures if auditing is enabled.
11. The progress bar is set to 10 percent.
12. The memory manager is called to create the section object and the memory manager's system worker threads (which are explained in Chapter 7).
13. National language support (NLS) tables are mapped into system space.
14. Ntdll.dll is mapped into the system address space.
15. The cache manager initializes the file system cache data structures and creates its worker threads.
16. The configuration manager creates the \Registry key object in the object manager namespace and copies the initial registry data passed by Ntldr into the HARDWARE and SYSTEM hives.
17. Global file system driver data structures are initialized.
18. The Plug and Play manager calls the Plug and Play BIOS.
19. The progress bar is set to 20 percent.
20. The local procedure call (LPC) subsystem initializes the LPC port type object.
21. If the system was booted with boot logging (/BOOTLOG), the boot log file is initialized.
22. The progress bar is set to 25 percent.
23. The I/O manager initialization now takes place. This stage is a complex phase of system startup that accounts for 50 percent of the "progress" reported in the progress bar. The I/O manager considers each successful driver load to be another 2 percent of progress for the boot. (If there are more than 25 drivers to load, the progress bar stops at 75 percent.)

    The I/O manager first initializes various internal structures and creates the driver and device object types. It then calls the Plug and Play manager, power manager, and HAL to begin the various stages of dynamic device enumeration and initialization. (Because this process is complex and specific to the I/O system, we'll save the details for Chapter 9.) Then the Windows Management Instrumentation (WMI) subsystem is initialized, which provides WMI support for device drivers. (See the section "Windows Management Instrumentation" in Chapter 4 for more information.) Next, all the boot-start drivers are called to perform their driver-specific initialization, and the system-start device drivers are loaded and initialized. (Details on the processing of the driver load control information on the registry are also covered in Chapter 9.) Finally, the MS-DOS device names are created as symbolic links in the object manager's namespace.
24. The progress bar is set to 75 percent.

25. If the computer is booting in safe mode, this fact is recorded in the registry.
26. Unless explicitly disabled in the registry, paging of kernel-mode code (in Ntoskrnl and drivers) is enabled.
27. The progress bar is set to 80 percent.
28. The power manager is called to initialize various power management structures.
29. The progress bar is set to 85 percent.
30. The security reference monitor is called to create the Command Server Thread that communicates with Lsass. (See the section "Security System Components" in Chapter 8 for more on how security is enforced in Windows.)
31. The progress bar is set to 90 percent.
32. The last step is to create the Session Manager subsystem (Smss) process (introduced in Chapter 2). Smss is responsible for creating the user-mode environment that provides the visible interface to Windows—its initialization steps are covered in the next section.
33. The progress bar is (finally) set to 100%.

As a final step before considering the executive and kernel initialization complete, the phase 1 initialization thread waits for the handle to the Session Manager process with a timeout value of 5 seconds. If the Session Manager process exits before the 5 seconds elapse, the system crashes itself with a SESSION5_ INITIALIZATION_FAILED bug check code.

If the 5-second wait times out (that is, if 5 seconds elapse), the Session Manager is assumed to have started successfully, and the phase 1 initialization function calls the memory manager's zero page thread function (explained in Chapter 7). Thus, this system thread becomes the zero page thread for the remainder of the life of the system.

Smss, Csrss, and Winlogon

Smss is like any other user-mode process except for two differences: First, Windows considers Smss a trusted part of the operating system. Second, Smss is a *native* application. Because it's a trusted operating system component, Smss can perform actions few other processes can perform, such as creating security tokens. Because it's a native application, Smss doesn't use Windows APIs—it uses only core executive APIs known collectively as the Windows native API. Smss doesn't use the Windows APIs because the Windows subsystem isn't executing when Smss launches. In fact, one of Smss's first tasks is to start the Windows subsystem.

Smss then calls the configuration manager executive subsystem to finish initializing the registry, fleshing the registry out to include all its keys. The configuration manager is programmed to know where the core registry hives are stored on disk (excluding hives corresponding to user profiles), and it records the paths to the hives it loads in the HKLM\SYSTEM\CurrentControlSet\Control\hivelist key.

The main thread of Smss performs the following initialization steps:

1. Creates an LPC port object (*SmApiPort*) and two threads to wait for client requests (such as to load a new subsystem or create a session).
2. Defines the symbolic links for MS-DOS device names (such as COM1 and LPT1).
3. If Terminal Services is installed, creates the \Sessions directory in the object manager's namespace (for multiple sessions).
4. Runs any programs defined in HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\BootExecute. Typically, this value contains one command to run Autochk (the boot-time version of Chkdsk).
5. Performs delayed file rename and delete operations as directed by HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\PendingFileRenameOperations and HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\PendingFileRenameOperations2.
6. Opens known DLLs, and creates section objects for them in the \Knowndlls directory of the Object Manager namespace. The list of DLLs considered known is located in HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\KnownDLLs, and the path to the directory in which the DLLs are located is stored in the Dlldirectory value of the key. See Chapter 6 for information on how the Known DLLs sections are used during DLL loading.
7. Creates additional paging files. Paging file configuration is stored under HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management\PagingFiles.
8. Initializes the registry. The configuration manager fleshes out the registry by loading the registry hives for the HKLM\SAM, HKLM\SECURITY, and HKLM\SOFTWARE keys. Although HKLM\SYSTEM\CurrentControlSet\Control\hivelist locates the hive files on disk, the configuration manager is coded to look for them in \Windows\System32\Config.
9. Creates system environment variables that are defined in HKLM\System\CurrentControlSet\Session Manager\Environment.
10. Loads the kernel-mode part of the Windows subsystem (Win32k.sys). Smss determines the location of Win32k.sys and other components it loads by looking for their paths in HKLM\SYSTEM\CurrentControlSet\Control\Session Manager. The initialization code in Win32k.sys uses the video driver to switch the screen to the resolution defined by the default profile, so this is the point at which the screen changes from the VGA mode the boot video driver uses to the default resolution chosen for the system.
11. Starts the subsystem processes, including Csrss. (As noted in Chapter 2, on Windows 2000 the POSIX and OS/2 subsystems are defined to start on demand.)

12. Starts the logon process (Winlogon). The startup steps of Winlogon are described shortly.
13. Creates LPC ports for debug event messages (*DbgSsApiPort* and *DbgUiApiPort*) and threads to listen on those ports.

Pending File Rename Operations

The fact that executable images and DLLs are memory-mapped when they are used makes it impossible to update core system files after Windows has finished booting. The MoveFileEx Windows API has an option to specify that a file move be delayed until the next boot. Service Packs and hotfixes that must update in-use memory-mapped files install replacement files onto a system in temporary locations and use the MoveFileEx API to have them replace otherwise in-use files. When used with that option, MoveFileEx simply records commands in the PendingFileRenameOperations and PendingFileRenameOperations2 values under HKLM\SYSTEM\CurrentControlSet\Control\Session Manager. These registry values are of type MULTI_SZ, where each operation is specified in pairs of file names: the first file name is the source location, and the second is the target location. Delete operations use an empty string as their target path. You can use the Pendmoves utility from *www.sysinternals.com* to view registered delayed rename and delete commands.

After performing these initialization steps, the main thread in Smss waits forever for the process handles to Csrss and Winlogon. If either of these processes terminates unexpectedly, Smss crashes the system, because Windows relies on their existence. (In Windows XP and later, if Csrss exits for any reason, the kernel crashes the system, not the Smss.)

Winlogon then performs its startup steps, such as creating the initial window station and desktop objects. If a DLL is specified in HKLM\Software\Microsoft\Windows NT\Current Version\WinLogon\GinaDLL, Winlogon uses that DLL as the GINA; otherwise, it uses the Microsoft default GINA, Msgina (\Windows\System32\Msgina.dll), which displays the standard Windows logon dialog box. Winlogon then creates the service control manager (SCM) process (\Windows\System32\Services.exe), which loads all services and device drivers marked for auto-start, and the local security authentication subsystem (Lsass) process (\Windows\System32\Lsass.exe). (For more details on the startup sequence for Winlogon and Lsass, see the section “Winlogon Initialization” in Chapter 8.)

After the SCM initializes the auto-start services and drivers and a user has successfully logged on at the console, the SCM deems the boot successful. The registry last known good control set (as indicated by HKLM\SYSTEM\Select\LastKnownGood) is updated to match \CurrentControlSet.

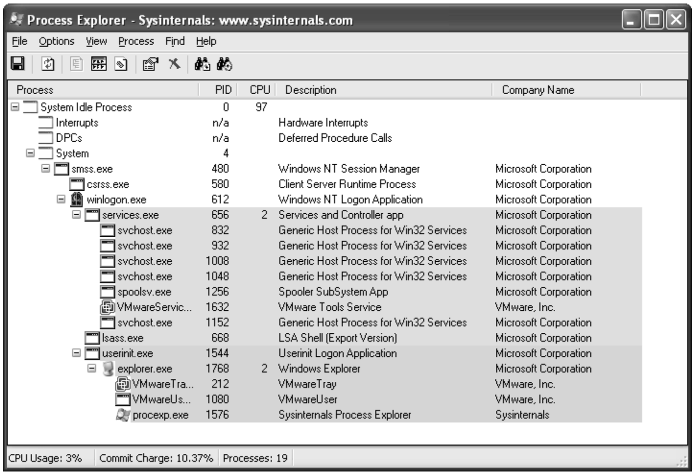
> **Note** Because noninteractive servers might never have an interactive logon, they might not get LastKnownGood updated to reflect the control set used for a successful boot.
>
> You can override the definition of a successful boot by setting HKLM\Software\Microsoft\Windows NT\CurrentVersion\Winlogon\ReportBootOk to 0, writing a custom boot verification program that calls the NotifyBootConfigStatus Windows API when a boot is successful, and entering the path to the verification program in HKLM\System\CurrentControlSet\Control\BootVerificationProgram.

After launching the SCM, Winlogon waits for an interactive logon notification from the GINA. When it receives a logon and validates the logon (a process for which you can find more information in the section "User Logon Steps" in Chapter 8), Winlogon loads the registry hive from the profile of the user logging in and maps it to HKCU. It then sets the user's environment variables that are stored in HKCU\Environment and notifies the Winlogon notification packages registered in HKLM\Software\Microsoft\Windows NT\CurrentVersion\Winlogon\Notify that a logon has occurred.

Winlogon next tells the GINA to start the shell. In response to this request, Msgina launches the executable or executables specified in HKLM\Software\Microsoft\Windows NT\CurrentVersion\WinLogon\Userinit (with multiple executables separated by commas) that by default points at \Windows\System32\Userinit.exe. Userinit.exe performs the following steps:

1. Processes the user scripts specified in HKCU\Software\Policies\Microsoft\Windows\System\Scripts and the machine logon scripts in HKLM\Software\Policies\Microsoft\Windows\System\Scripts. (Because machine scripts run after user scripts, they can override user settings.)
2. If group policy specifies a user profile quota, starts \Windows\System32\Proquota.exe to enforce the quota for the current user.
3. Launches the comma-separated shell or shells specified in HKCU\Software\Microsoft\Windows NT\CurrentVersion\Winlogon\Shell. If that value doesn't exist, Userinit.exe launches the shell or shell specified in HKLM\Software\Microsoft\Windows NT\CurrentVersion\Winlogon\Shell, which is by default Explorer.exe.

Winlogon then notifies registered network providers that a user has logged in. The Microsoft network provider, Multiple Provider Router (\Windows\System32\Mpr.dll), restores the user's persistent drive letter and printer mappings stored in HCU\Network and HKCU\Printers, respectively. Figure 5-3 shows the process tree as seen in Process Explorer during a login before Userinit has exited.


Process Explorer - Sysinternals: www.sysinternals.com
File Options View Process Find Help

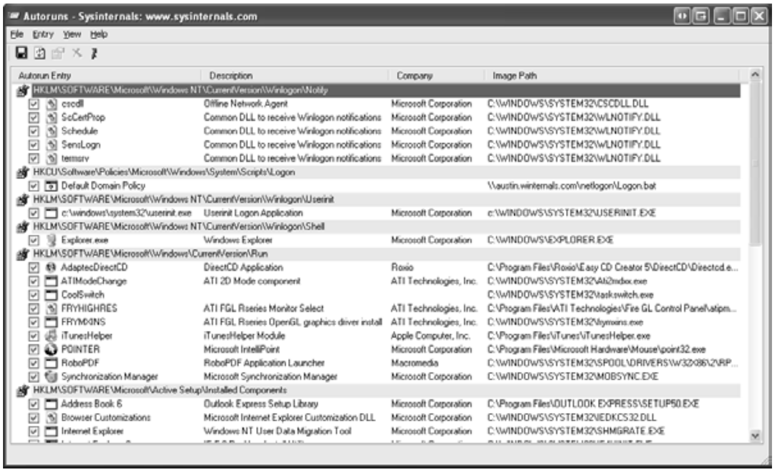
Process	PID	CPU	Description	Company Name
System Idle Process	0	97		
Interrupts	n/a		Hardware Interrupts	
DPCs	n/a		Deferred Procedure Calls	
System	4			
smss.exe	480		Windows NT Session Manager	Microsoft Corporation
csrss.exe	580		Client Server Runtime Process	Microsoft Corporation
winlogon.exe	612		Windows NT Logon Application	Microsoft Corporation
services.exe	656	2	Services and Controller app	Microsoft Corporation
svchost.exe	832		Generic Host Process for Win32 Services	Microsoft Corporation
svchost.exe	932		Generic Host Process for Win32 Services	Microsoft Corporation
svchost.exe	1008		Generic Host Process for Win32 Services	Microsoft Corporation
svchost.exe	1048		Generic Host Process for Win32 Services	Microsoft Corporation
spoolsv.exe	1256		Spooler SubSystem App	Microsoft Corporation
VMwareServic...	1632		VMware Tools Service	VMware, Inc.
svchost.exe	1152		Generic Host Process for Win32 Services	Microsoft Corporation
lsass.exe	668		LSA Shell (Export Version)	Microsoft Corporation
userinit.exe	1544		Userinit Logon Application	Microsoft Corporation
explorer.exe	1768	2	Windows Explorer	Microsoft Corporation
VMwareTra...	212		VMwareTray	VMware, Inc.
VMwareUs...	1080		VMwareUser	VMware, Inc.
procexp.exe	1576		Sysinternals Process Explorer	Sysinternals

CPU Usage: 3% Commit Charge: 10.37% Processes: 19

Figure 5-3 Process tree during logon

Images that Start Automatically

In addition to the Userinit and Shell registry values in Winlogon's key, there are many other registry locations and directories that default system components check and process for automatic process startup during the boot and logon process. The Msconfig utility (included in Windows XP and Windows Server 2003 in \Windows\System32\Msconfig.exe) displays the images configured by several of the locations. Sysinternals' Autoruns tool, which you can download from *www.sysinternals.com* and that is shown in Figure 5-4, examines more locations than Msconfig and displays more information about the images configured to automatically run. By default, Autoruns shows only the locations that are configured to automatically execute at least one image, but checking the Include Empty Locations entry in the View menu causes Autoruns to show all the locations it inspects. The View menu also has selections to direct Autoruns to display information about other types of autostarting images, such as Windows services and Explorer add-ons.



Autorun Entry	Description	Company	Image Path
HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon\Notify			
☑ cscdll	Offline Network Agent	Microsoft Corporation	C:\WINDOWS\SYSTEM32\CSCDLL.DLL
☑ ScCertProp	Common DLL to receive Winlogon notifications	Microsoft Corporation	C:\WINDOWS\SYSTEM32\WLNOTIFY.DLL
☑ Schedule	Common DLL to receive Winlogon notifications	Microsoft Corporation	C:\WINDOWS\SYSTEM32\WLNOTIFY.DLL
☑ SensLogn	Common DLL to receive Winlogon notifications	Microsoft Corporation	C:\WINDOWS\SYSTEM32\WLNOTIFY.DLL
☑ termsrv	Common DLL to receive Winlogon notifications	Microsoft Corporation	C:\WINDOWS\SYSTEM32\WLNOTIFY.DLL
HKCU\Software\Policies\Microsoft\Windows\System\Scripts\Logon			
☑ Default Domain Policy			\\austin.winternals.com\netlogon\Logon.bat
HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon\Userinit			
☑ c:\windows\system32\userinit.exe	Userinit Logon Application	Microsoft Corporation	c:\WINDOWS\SYSTEM32\USERINIT.EXE
HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon\Shell			
☑ Explorer.exe	Windows Explorer	Microsoft Corporation	C:\WINDOWS\EXPLORER.EXE
HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run			
☑ AdaptecDirectCD	DirectCD Application	Roxio	C:\Program Files\Roxio\Easy CD Creator 5\DirectCD\Directcd.e...
☑ ATIModeChange	ATI 2D Mode component	ATI Technologies, Inc.	C:\WINDOWS\SYSTEM32\Ati2mdxx.exe
☑ CoolSwitch			C:\WINDOWS\SYSTEM32\taskswitch.exe
☑ FRYHIGHRES	ATI FGL Rseries Monitor Select	ATI Technologies, Inc.	C:\Program Files\ATI Technologies\Fire GL Control Panel\atipm...
☑ FRYMXINS	ATI FGL Rseries OpenGL graphics driver install	ATI Technologies, Inc.	C:\WINDOWS\SYSTEM32\frymxins.exe
☑ iTunesHelper	iTunesHelper Module	Apple Computer, Inc.	C:\Program Files\iTunes\iTunesHelper.exe
☑ POINTER	Microsoft IntelliPoint	Microsoft Corporation	C:\Program Files\Microsoft Hardware\Mouse\point32.exe
☑ RoboPDF	RoboPDF Application Launcher	Macromedia	C:\WINDOWS\SYSTEM32\SPOOL\DRIVERS\W32X86\2\RP...
☑ Synchronization Manager	Microsoft Synchronization Manager	Microsoft Corporation	C:\WINDOWS\SYSTEM32\MOBSYNC.EXE
HKLM\SOFTWARE\Microsoft\Active Setup\Installed Components			
☑ Address Book 6	Outlook Express Setup Library	Microsoft Corporation	C:\Program Files\OUTLOOK EXPRESS\SETUP50.EXE
☑ Browser Customizations	Microsoft Internet Explorer Customization DLL	Microsoft Corporation	C:\WINDOWS\SYSTEM32\IEDKCS32.DLL
☑ Internet Explorer	Windows NT User Data Migration Tool	Microsoft Corporation	C:\WINDOWS\SYSTEM32\SHMGRATE.EXE

Figure 5-4 The Autoruns tool available from www.sysinternals.com

EXPERIMENT: Autoruns

Many users are unaware of how many programs execute as part of their logon. Original equipment manufacturers (OEMs) often configure their systems with add-on utilities that execute in the background using registry values or file system directories processed for automatic execution and so are not normally visible. See what programs are configured to start automatically on your computer by running the Autoruns utility from *www.sysinternals.com*. Compare the list shown in Autoruns with that shown in Msconfig (available on Windows XP and Windows Server 2003), and identify any differences. Then ensure that you understand the purpose of each program.

Troubleshooting Boot and Startup Problems

This section presents approaches to solving problems that can occur during the Windows startup process as a result of hard disk corruption, file corruption, missing files, and third-party driver bugs. First, we describe three Windows boot-problem recovery modes: last known good, safe mode, and the Recovery Console. Then we present common boot problems, their causes, and approaches to solving them. The solutions refer to last known good, safe mode, the Recovery Console, and other tools that ship with Windows.

Last Known Good

Last known good (LKG) is a useful mechanism for getting a system that crashes during the boot process back to a bootable state. Because the system's configuration settings are stored in HKLM\System\CurrentControlSet\Control and driver and service configuration is stored in HKLM\System\CurrentControlSet\Services, changes to these parts of the registry can render a system unbootable. For example, if you install a device driver that has a bug that crashes the system during the boot, you can press the F8 key during the boot and select last known good from the resulting menu. The system marks the control set that it was using to boot the system as failed by setting the Failed value of HKLM\System\Select and then changes HKLM\System\Select\Current to the value stored in HKLM\System\Select\LastKnownGood. It also updates the symbolic link HKLM\System\CurrentControlSet to point at the LastKnownGood control set. Because the new driver's key is not present in the Services subkey of the LastKnownGood control set, the system will boot successfully.

Safe Mode

Perhaps the most common reason Windows systems become unbootable is that a device driver crashes the machine during the boot sequence. Because software or hardware configurations can change over time, latent bugs can surface in drivers at any time. Windows offers a way for an administrator to attack the problem: booting in *safe mode*. Safe mode is a concept Windows borrows from Consumer Windows—a boot configuration that consists of the minimal set of device drivers and services. By relying on only the drivers and services that are necessary for booting, Windows avoids loading third-party and other nonessential drivers that might crash.

When Windows boots, you press the F8 key to enter a special boot menu that contains the safe-mode boot options. You typically choose from three safe-mode variations: Safe Mode, Safe Mode With Networking, and Safe Mode With Command Prompt. Standard safe mode comprises the minimum number of device drivers and services necessary to boot successfully. Networking-enabled safe mode adds network drivers and services to the drivers and services that standard safe mode includes. Finally, safe mode with command prompt is identical to standard safe mode except that Windows runs the command prompt application (Cmd.exe) instead of Windows Explorer as the shell when the system enables GUI mode.

Windows includes a fourth safe mode-Directory Services Restore mode—which is different from the standard and networking—enabled safe modes. You use Directory Services Restore mode to boot the system into a mode where the Active Directory directory service of a domain controller is offline and unopened. This allows you to perform repair operations on the database or restore it from backup media. All drivers and services, with the exception of the Active Directory service, load during a Directory Services Restore mode boot. In cases where you can't log in a system because of Active Directory database corruption, this mode enables you to repair the corruption.

Driver Loading in Safe Mode

How does Windows know which device drivers and services are part of standard and networking-enabled safe mode? The answer lies in the HKLM\SYSTEM\CurrentControlSet\Control\SafeBoot registry key. This key contains the Minimal and Network subkeys. Each subkey contains more subkeys that specify the names of device drivers or services or of groups of drivers. For example, the vga.sys subkey identifies the VGA display device driver that the startup configuration includes. The VGA display driver provides basic graphics services for any PC-compatible display adapter. The system uses this driver as the safe-mode display driver in lieu of a driver that might take advantage of an adapter's advanced hardware features but that might also prevent the system from booting. Each subkey under the SafeBoot key has a default value that describes what the subkey identifies; the vga.sys subkey's default value is "Driver".

The Boot file system subkey has as its default value "Driver Group". When developers design a device driver's installation script, they can specify that the device driver belongs to a driver group. The driver groups that a system defines are listed in the List value of the HKLM\SYSTEM\CurrentControlSet\Control\ServiceGroupOrder key. A developer specifies a driver as a member of a group to indicate to Windows at what point during the boot process the driver should start. The ServiceGroupOrder key's primary purpose is to define the order in which driver groups load; some driver types must load either before or after other driver types. The Group value beneath a driver's configuration registry key associates the driver with a group.

Driver and service configuration keys reside beneath HKLM\SYSTEM\CurrentControlSet\Services. If you look under this key, you'll find the VgaSave key for the VGA display device driver, which you can see in the registry is a member of the Video Save group. Any file system drivers that Windows requires for access to the Windows system drive are in the Boot file system group. If the system drive is NTFS, the NTFS driver is part of this group. (The value of Group under the Ntfs key is Boot file system.) Otherwise, the Fastfat file system driver (which supports FAT12, FAT16, and FAT32 drives in Windows) is part of this group. Other file system drivers are part of the File system group, which the standard and networking-enabled safe-mode configurations also include.

When you boot into a safe-mode configuration, the boot loader (Ntldr) passes an associated switch to the kernel (Ntoskrnl.exe) as a command-line parameter, along with any switches you've specified in the Boot.ini file for the installation you're booting. If you boot into any safe mode, Ntldr passes the /SAFEBOOT: switch. Ntldr appends one or more additional strings to /SAFEBOOT:, depending on which type of safe mode you select. For standard safe mode, Ntldr appends MINIMAL, and for networking-enabled safe mode, it adds NETWORK. Ntldr adds MINIMAL(ALTERNATESHELL) for safe mode with command prompt and DSREPAIR for Directory Services Restore mode.

The Windows kernel scans boot parameters in search of the safe-mode switches early during the boot and sets the internal variable *InitSafeBootMode* to a value that reflects the switches the kernel finds. The kernel writes the *InitSafeBootMode* value to the registry value HKLM\SYSTEM\CurrentControlSet\Control\SafeBoot\Option\OptionValue so that user-mode components, such as the SCM, can determine what boot mode the system is in. In addition, if the system is booting safe mode with command prompt, the kernel sets the HKLM\SYSTEM\CurrentControlSet\Control\SafeBoot\Option\UseAlternateShell value to 1. The kernel records the parameters that Ntldr passes to it in the value HKLM\SYSTEM\CurrentControlSet\Control\SystemStartOptions.

When the I/O manager kernel subsystem loads device drivers that HKLM\SYSTEM\CurrentControlSet\Services specifies, the I/O manager executes the function *IopLoadDriver*. When the Plug and Play manager detects a new device and wants to dynamically load the device driver for the detected device, the Plug and Play manager executes the function *IopCallDriverAddDevice*. Both these functions call the function *IopSafeBootDriverLoad* before they load the driver in question. *IopSafeBootDriverLoad* checks the value of *InitSafeBootMode* and determines whether the driver should load. For example, if the system boots in standard safe mode, *IopSafeBootDriverLoad* looks for the driver's group, if the driver has one, under the Minimal subkey. If *IopSafeBootDriverLoad* finds the driver's group listed, *IopSafeBootDriverLoad* indicates to its caller that the driver can load. Otherwise, *IopSafeBootDriverLoad* looks for the driver's name under the Minimal subkey. If the driver's name is listed as a subkey, the driver can load. If *IopSafeBootDriverLoad* can't find the driver group or driver name subkeys, the driver can't load. If the system boots in networking-enabled safe mode, *IopSafeBootDriverLoad* performs the searches on the Network subkey. If the system doesn't boot in safe mode, *IopSafeBootDriverLoad* lets all drivers load.

An exception loophole exists regarding the drivers that safe mode excludes from a boot: Ntldr, rather than the kernel, loads any drivers with a Start value of 0 in their registry key, which specifies loading the drivers at boot time. Ntldr doesn't check the SafeBoot registry key because it assumes that any driver with a Start value of 0 is required for the system to boot successfully. Because Ntldr doesn't check the SafeBoot registry key to identify which drivers to load, Ntldr therefore loads all boot-start drivers (and later Ntoskrnl starts them).

Safe-Mode-Aware User Programs

When the service control manager (SCM) user-mode component (which Services.exe implements) initializes during the boot process, the SCM checks the value of HKLM\SYSTEM\CurrentControlSet\Control\SafeBoot\Option\OptionValue to determine whether the system is performing a safe mode boot. If so, the SCM mirrors the actions of *IopSafeBootDriverLoad*. Although the SCM processes the services listed under HKLM\SYSTEM\CurrentControlSet\Services, it loads only services that the appropriate safe-mode subkey specifies by name. You can find more information on the SCM initialization process in the section "Services" in Chapter 4.

Userinit (\Windows\System32\Userinit.exe) is another user-mode component that needs to know whether the system is booting in safe mode. Userinit, the component that initializes a user's environment when the user logs on, checks HKLM\SYSTEM\CurrentControlSet\Control\SafeBoot\Option\UseAlternateShell. If this value is set, Userinit runs the program specified as the user's shell in the value HKLM\SYSTEM\CurrentControlSet\Control\SafeBoot\AlternateShell rather than executing Explorer.exe. Windows writes the program name Cmd.exe to the AlternateShell value during installation, making the Windows command prompt the default shell for safe mode with command prompt. Even though command prompt is the shell, you can type **Explorer.exe** at the command prompt to start Windows Explorer, and you can run any other GUI program from the command prompt as well.

How does an application determine whether the system is booting in safe mode? By calling the Windows *GetSystemMetrics(SM_CLEANBOOT)* function. Batch scripts that need to perform certain operations when the system boots in safe mode look for the SAFEBOOT_OPTION environment variable because the system defines this environment variable only when booting in safe mode.

Boot Logging in Safe Mode

When you direct the system to boot into safe mode, Ntldr hands the string specified by the /BOOTLOG option to the Windows kernel as a parameter, together with the parameter that requests safe mode. When the kernel initializes, it checks for the presence of the boot log parameter, whether or not any safe-mode parameter is present. If the kernel detects a boot log string, the kernel records the action the kernel takes on every device driver it considers for loading. For example, if *IopSafeBootDriverLoad* tells the I/O manager not to load a driver, the

I/O manager calls *IopBootLog* to record that the driver wasn't loaded. Likewise, after *IopLoadDriver* successfully loads a driver that is part of the safe-mode configuration, *IopLoadDriver* calls *IopBootLog* to record that the driver loaded. You can examine boot logs to see which device drivers are part of a boot configuration.

Because the kernel wants to avoid modifying the disk until Chkdsk executes, late in the boot process, *IopBootLog* can't simply dump messages into a log file. Instead, *IopBootLog* records messages in the HKLM\SYSTEM\CurrentControlSet\BootLog registry value. As the first user-mode component to load during a boot, the Session Manager (\Windows\System32\Smss.exe) executes Chkdsk to ensure the system drives' consistency and then completes registry initialization by executing the *NtInitializeRegistry* system call. The kernel takes this action as a cue that it can safely open a log file on the disk, which it does, invoking the function *IopCopyBootLogRegistryToFile*. This function creates the file Ntbtlog.txt in the Windows system directory (\Windows by default) and copies the contents of the BootLog registry value to the file. *IopCopyBootLogRegistryToFile* also sets a flag for *IopBootLog* that lets *IopBootLog* know that writing directly to the log file, rather than recording messages in the registry, is now OK. The following output shows the partial contents of a sample boot log:

```
Service Pack 1 3 30 2004 14:05:21.500
Loaded driver \WINDOWS\system32\ntoskrnl.exe
Loaded driver \WINDOWS\system32\hal.dll
Loaded driver \WINDOWS\system32\KDCOM.DLL
Loaded driver \WINDOWS\system32\BOOTVID.dll
Loaded driver ACPI.sys
Loaded driver \WINDOWS\System32\DRIVERS\WMILIB.SYS
Loaded driver pci.sys
Loaded driver isapnp.sys
Loaded driver intelide.sys
Loaded driver \WINDOWS\System32\DRIVERS\PCIIDEX.SYS
Loaded driver MountMgr.sys
Loaded driver ftdisk.sys
Loaded driver dmload.sys
Loaded driver dmio.sys Microsoft (R) Windows 2000 (R) Version 5.0 (Build 2195)
 2 11 2000 10:53:27.500
Loaded driver \WINNT\System32\ntoskrnl.exe
Loaded driver \WINNT\System32\hal.dll
Loaded driver \WINNT\System32\BOOTVID.DLL
Loaded driver ACPI.sys
Loaded driver \WINNT\System32\DRIVERS\WMILIB.SYS
Loaded driver pci.sys
Loaded driver isapnp.sys
Loaded driver compbatt.sys
Loaded driver \WINNT\System32\DRIVERS\BATTC.SYS
Loaded driver intelide.sys
Loaded driver \WINNT\System32\DRIVERS\PCIIDEX.SYS
Loaded driver pcmcia.sys
Loaded driver ftdisk.sys
Loaded driver Diskperf.sys
Loaded driver dmload.sys
Loaded driver dmio.sys
```

```
§
Did not load driver \SystemRoot\System32\Drivers\lbrtfdc.SYS
Did not load driver \SystemRoot\System32\Drivers\Sfloppy.SYS
Did not load driver \SystemRoot\System32\Drivers\i2omgmt.SYSDid not load driver Media Contro
l Devices
Did not load driver Communications Port
Did not load driver Audio Codecs
§
```

Recovery Console

Safe mode is a satisfactory fallback for systems that become unbootable because a device driver crashes during the boot sequence, but in some situations a safe-mode boot won't help the system boot. For example, if a driver that prevents the system from booting is a member of a Safe group, safe-mode boots will fail. Another example of a situation in which safe mode won't help the system boot is when a third-party driver, such as a virus scanner driver, that loads at the boot prevents the system from booting. (Boot-start drivers load whether or not the system is in safe mode.) Other situations in which safe-mode boots will fail are when a system module or critical device driver file that is part of a safe-mode configuration becomes corrupt or when the system drive's Master Boot Record (MBR) is damaged. You can get around these problems by using the Windows Recovery Console. The Recovery Console allows you to boot into a limited command-line shell from the Windows CD or boot disks to repair an installation without having to boot the installation.

When you boot a system from the Windows CD or boot disks, you eventually see a screen that gives you the choice of either installing Windows or repairing an existing installation. If you choose to repair an installation, the system prompts you to insert the Windows CD (if it isn't already loaded in the system's CD drive) and then to choose among two repair options: to start the Recovery Console or to initiate the emergency repair process. If you press the F10 key at the Setup Welcome screen, you bypass the menu options and take a shortcut directly to the Recovery Console.

When you start the Recovery Console, it gives you a list of Windows NT and Windows installations to choose from that it compiled when it scanned the computer's hard disks. After you make a selection, the system prompts you to enter the Administrator account password to log on to the installation as the administrator. If you successfully log on, the system puts you into a command shell that is similar to an MS-DOS environment. The command set is flexible and lets you perform simple file operations (such as copy, rename, and delete), enable and disable services and drivers, and even repair MBRs and boot records. However, the Recovery Console won't let you access directories other than root directories, the system directory of the installation you logged on to, or directories on removable drives such as CDs and 3.5-inch floppy disks unless local security policy settings stored in the SECURITY hive of the Registry of the installation into which you log in permit it. This prohibition provides a certain level of security for data that an administrator might not usually be able to access. You can override this restriction by using the Local Security Policy editor (secpol.msc) to configure the Recovery Console settings in the Security Options folder of Local Policies when the system is booted normally.

The Recovery Console uses the native Windows system call interface to perform file I/O to support commands such as *Cd*, *Rename*, and *Move*. The *Enable* and *Disable* commands, which let you change the startup modes of device drivers and services, work differently. For example, when you tell the Recovery Console that you want to disable a device driver, it reaches into the installation's Services key and manipulates the Start value of the specified driver's key, changing the value to SERVICE_DISABLED. The next time the installation boots, that device driver won't load. (The Recovery Console also loads the SYSTEM hive [\Windows\System32\Config\System] for the installation you log on to. This hive contains the information stored in the HKLM\SYSTEM\CurrentControlSet\Services registry key.)

When you boot from the Windows CD or the boot disks, by the time the system gives you the choice to install or repair Windows, the CD has booted a copy of the Windows kernel, including all necessary supporting device drivers (for example, NTFS or FAT drivers, SCSI drivers, a video driver). On x86 systems, the Txtsetup.sif file in the I386 directory of the Windows CD guides the boot from the CD; the file contains directives that identify which files need to load and where the files are located on the CD. Just as when you boot Windows from a hard disk, the first user-mode program the kernel executes is Session Manager (Smss.exe), located in the I386\System32 folder. The Session Manager that Windows Setup uses differs from the standard-installation Session Manager. The former component presents you with the menus that let you install or repair Windows and the menu that asks you what type of repair you want to perform. If you're installing Windows, Session Manager is the component that guides you through choosing a partition to install to and copies files to the hard disk.

When you run the Recovery Console, Session Manager loads and starts two device drivers that implement the Recovery Console: Spcmdcon.sys and Setupdd.sys. Spcmdcon.sys presents an interactive command prompt and performs high-level command processing. Setupdd.sys is a support driver that gives Spcmdcon.sys a set of functions that let Spcmdcon.sys manage disk partitions, load registry hives, and display and manage video output. Setupdd.sys also communicates with disk drivers to manage disk partitions and uses basic video support built into the Windows kernel to display messages on the screen.

When you choose an installation to log on to and the Recovery Console accepts your password, the Recovery Console must validate your logon attempt, even though the installation's Windows security subsystem isn't up and running. Thus, the Recovery Console alone must determine whether your password matches the system's Administrator account. The Recovery Console's first step in this process is to use Setupdd.sys to load the installation's Security Accounts Manager (SAM) registry hive, which stores password information, from the hard disk. The SAM hive resides in \Windows\System32\Config\Sam. After loading the hive, the Recovery Console locates the system key in the installation's registry and uses the system key to decrypt the in-memory copy of the SAM. SAM hive encryption is a feature introduced in Windows NT 4 Service Pack 3 that adds protection against MS-DOS-based password snoopers who try to read passwords directly out of a hive file.

Next, the Recovery Console (Spcmdcon.sys) locates the Administrator account password in the SAM, and in the final authentication step, the Recovery Console uses the MD5 hash algorithm—the same algorithm that the Windows logon process uses—to hash the password entered and compares the hash against the hashed password that the SAM stores. If the Recovery Console finds a match, the system considers you logged on. If the Recovery Console doesn't find a match, the system denies you access to the Recovery Console.

Solving Common Boot Problems

This section describes problems that can occur during the boot process, describing their symptoms, causes, and approaches to solving them. To help you locate a problem that you might encounter, they are organized according to the place in the boot at which they occur.

MBR Corruption

- **Symptoms** A system that has Master Boot Record (MBR) corruption will execute the BIOS power-on self test (POST), display BIOS version information or OEM branding, switch to a black screen, and then hang. Depending on the type of corruption the MBR has experienced, you might see one of the following messages: "Invalid Partition Table," "Error Loading Operating System," or "Missing Operating System."
- **Cause** The MBR can become corrupt because of hard-disk errors, disk corruption as a result of a driver bug while Windows is running, or intentional scrambling as a result of a virus.
- **Resolution** Boot into the Recovery Console and execute the *fixmbr* command. This command replaces the executable code in the MBR. Unfortunately, it does not repair the partition table. The only way to restore a damaged partition table is to restore it from a backup copy or to use a third-party disk-corruption repair tool.

Boot Sector Corruption

- **Symptoms** Boot sector corruption can look like MBR corruption where the system hangs after BIOS POST at a black screen, or you might see the messages "A disk read error occurred," "NTLDR is missing," or "NTLDR is compressed" displayed in a black screen.
- **Cause** The MBR can become corrupt because of hard disk errors, disk corruption as a result of a driver bug while Windows is running, or intentional scrambling as a result of a virus.
- **Resolution** Boot into the Recovery Console and execute the *fixboot* command. This command rewrites the boot sector of the volume that you specify. You should execute the command on both the system and boot volumes if they are different.

Boot.ini Misconfiguration

- **Symptom** After BIOS POST, you'll see a message that begins "Windows could not start because of a computer disk hardware configuration problem," "Could not read from selected boot disk," or "Check boot path and disk hardware."
- **Cause** The Boot.ini file has been deleted, is corrupted, or no longer references the boot volume because the addition of a partition has changed the Advanced RISC Computing (ARC) name of the volume.
- **Resolution** Boot into the Recovery Console, and execute the "bootcfg /rebuild". This command has the Recovery Console scan each volume looking for Windows installations. When it discovers an installation, it asks you whether it should add it to Boot.ini as a boot option and what name it should display for the installation in the boot menu.

System File Corruption

- **Symptoms** There are several ways the corruption of system files—which include executables, drivers, or DLLs—can manifest. One way is with a message on a black screen after BIOS POST that says, "Windows could not start because the following file is missing or corrupt," followed by the name of a file and a request to re-install the file. Another way is with a blue screen crash during the boot with the text, "STOP: 0xC0000135 {Unable to Locate Component}."
- **Causes** The volume on which a system file is located is corrupt or one or more system files have been deleted or become corrupt.
- **Resolution** Boot into the Recovery Console, and execute the *chkdsk* command. Chkdsk will attempt to repair volume corruption. If Chkdsk does not report any problems, obtain a backup copy of the system file in question. One place to check is in the \Windows\System32\DllCache directory, in which Windows places copies of many system files for access by Windows File Protection. (See the "Windows File Protection" sidebar.) If you cannot find a copy of the file there, see if you can locate a copy from another system in the network. Note that the backup file must be from the same Service Pack or hot fix as the file that you are replacing.

In some cases, multiple system files are deleted or become corrupt, so the repair process can involve multiple reboots and boot failures as you repair the files one by one. If you believe the system file corruption to be extensive, you should consider restoring the system from a backup image, such as one generated by Automated System Recovery (ASR). When you run Windows Backup (located in the System folder under Accessories in the Start menu), you can generate an ASR backup image, which includes all the files on the system and boot volumes, plus a floppy disk on which it stores information about the system's disks and volumes. To restore a system from an ASR, back up boot from the Windows setup media and press F2 when prompted.

If you do not have a backup from which to restore, a last resort is to execute a Windows repair install: boot from the Windows setup media, and follow the wizard as if you were going to perform a new installation. The wizard will ask you whether you want to perform a repair or fresh install. When you tell it that you want to repair, Setup reinstalls all system files, leaving your application data and registry settings intact.

Windows File Protection

In addition to its role as the interactive logon interface and Session Manager, Winlogon also implements Windows File Protection (WFP). WFP, which is implemented in the two DLLs \Windows\System32\Sfc.dll and \Windows\System32\Sfc_os.dll, monitors several directories for changes to key drivers, executables, and DLLs, including most subdirectories under \Windows, using the native API version of *ReadDirectoryChangesW*. When WFP sees that a change has occurred to a system file listed in \Windows\System32\Sfcfiles.Dll (and you can use the Strings utility from *www.sysinternals.com* to see the files listed in Sfcfiles.dll), it checks to see whether the file is digitally signed by Microsoft (a process for which you can find more information in the "Driver Installation" section of Chapter 9). If the file is digitally signed by Microsoft, WFP allows the change and copies the file to the WFP backup directory. By default, the backup directory is \Windows\System32\DllCache, although that can be overridden by defining the Registry value HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\Winlogon\SFCDllCacheDir. Hot fixes and service packs always install Microsoft-signed system files.

If the file modification doesn't result in a file that isn't Microsoft-signed, WFP replaces the modification with the backup version of the file from the DLLCache subdirectory. If Winlogon can't find a backup version in that directory, it checks in the network install path if the system was installed using a network install or in the setup media (prompting for insertion) if the install was from local media.

System Hive Corruption

- **Symptoms** If the System registry hive (which is discussed along with hive files in the "Registry" section of Chapter 4) is missing or corrupted, NTLDR will display the message, "Windows could not start because the following file is missing or corrupt: \WINDOWS\SYSTEM32\CONFIG\SYSTEM," on a black screen after the BIOS POST.
- **Causes** The System registry hive, which contains configuration information necessary for the system to boot, has become corrupt or has been deleted.
- **Resolution** Boot into the Recovery Console, and execute the *chkdsk* command on the boot volume to correct any volume corruption. If the problem is not corrected, obtain a backup of the System registry hive. If you have made ASR backups of the system or have used the Windows Backup utility to make backups of system state (an option in the backup UI), copies of the registry hives from the most recent backup are stored in \Windows\Repair, so copy the file named System to \Windows\System32\Config.

If you're running Windows XP and System Restore is enabled (System Restore is discussed in Chapter 12) , you can often obtain a more recent backup of the registry hives, including the System hive, from the most recent restore point. However, you may not be able to access the directory in which restores points are stored, \System Volume Information, from within the Recovery Console. Windows XP Service Pack 1 Versions of the Recovery Console allow access to that directory, but older versions do not unless the system's local security policy allows it. You can override the restriction if necessary by using the Local Security Policy Editor to change Recovery Console settings, as described earlier. You can also use third-party tools to gain access to other directories. If you can access the restore point directories, you can follow these steps to get at their registry hives:

1. Navigate to the directory whose name begins with "_restore" under the \System Volume Information directory of the boot volume.
2. Locate the RP subdirectory with the highest number as its suffix (for example, "RP173").
3. Copy the file named _REGISTRY_MACHINE_SYSTEM from the snapshot subdirectory to \Windows\System32\Config\System.
4. Reboot.

Another option is to try and repair the corruption using the Microsoft ChkReg tool. The tool attempts to automatically repair registry corruption, and it works by running off of the Windows XP setup floppy disks. You can find the tool and instructions on how to use it at *http://www.microsoft.com/downloads/details.aspx?displaylang=en&familyid=56d3c201-2c68-4de8-9229-ca494362419c.*

If you haven't made backups, don't have access to restore points, and the ChkReg tool doesn't fix the corruption (or the hives are missing), you can use the copy of the System hive stored in \Windows\Repair as a last resort. Windows Setup makes a copy of the System hive after it completes an installation, so you will lose system configuration changes and device driver installations made since then.

Post–Splash Screen Crash or Hang

- **Symptoms** Problems that occur after the Windows splash screen displays, the desktop appears, or you log in fall into this category and can appear as a blue screen crash or a hang, where the entire system is frozen or the mouse cursor tracks the mouse but the system is otherwise unresponsive.
- **Causes** These problems are almost always a result of a bug in a device driver, but they can sometimes be the result of corruption of a registry hive other than the System hive.
- **Resolution** You can take several steps to try and correct the problem. The first thing you should try is the last known good configuration. Last known good (LKG), which is described earlier in this chapter and in the "Services" section of Chapter 4, consists of the registry control set that was last used to boot the system successfully. Because a control set includes core system configuration and the device driver and services

registration database, using a version that does not reflect changes or newly installed drivers or services might avoid the source of the problem. You access last known good by pressing the F8 key early in the boot process to access the same menu from which you can boot into safe mode.

As stated earlier in the chapter, when you boot into LKG, the system saves the control set that you are avoiding and labels it as the *failed* control set. You can leverage the failed control set in cases where LKG makes a system bootable to determine what was causing the system to fail to boot by exporting the contents of the current control set of the successful boot and the failed control set to .reg files. You do this by using the Regedit's export functionality, which you access under the File menu (or under the Registry menu if you are running Windows 2000):

1. Run Regedit, and select HKLM\System\CurrentControlSet.
2. Select Export from the File menu, and save to a file named good.reg.
3. Open HKLM\System\Select, read the value of Failed, and select the subkey named HKLM\System\Control*XXX*, where *XXX* is the value of Failed.
4. Export the contents of the control set to bad.reg.
5. Use Wordpad (which is found under Accessories in the Start menu) to globally replace all instances of "CurrentControlSet" in good.reg with "ControlSet".
6. Use Wordpad to change all instances of "Control*XXX*" (replacing *XXX* with the value of the Failed control set) in bad.reg with "ControlSet".
7. Run Windiff from the Support Tools, and compare the two files.

The differences between a failed control set and a good one can be numerous, so you should focus your examination on changes beneath the Control subkey as well as under the Parameters subkeys of drivers and services registered in the Services subkey. Ignore changes made to Enum subkeys of driver registry keys in the Services branch of the control set.

If the problem you're experiencing is caused by a driver or service that was present on the system since before the last successful boot, LKG will not make the system bootable. Similarly, if a problematic configuration setting changed outside the control set or was made before the last successful boot, LKG will not help. In those cases, the next option to try is safe mode (described earlier in this section). If the system boots successfully in safe mode and you know that particular driver was causing the normal boot to fail, you can disable the driver by using the Device Manager (accessible from the Hardware tab of the System Control Panel applet). To do so, select the driver in question and choose Disable from the Action menu. If you're running Windows XP or Windows Server 2003, you recently updated the driver, and believe that the update introduced a bug, you can choose to roll back the driver to its previous version instead, also with the Device Manager. To restore a driver to its previous version, double-click on the driver to open its properties dialog box and press Roll Back Driver on the Drivers tab.

On Windows XP systems with System Restore enabled, an option when LKG fails is to roll back all system state (as defined by System Restore) to a previous point in time. Safe mode detects the existence of restore points, and when they are present it will ask you whether you want to log in to the installation to perform a manual diagnosis and repair or launch the System Restore Wizard. Using System Restore to make a system bootable again is attractive when you know the cause of a problem and want the repair to be automatic or when you don't know the cause but do not want to invest time to determine the cause.

If System Restore is not an option or you want to determine the cause of a crash during the normal boot and the system boots successfully in safe mode, attempt to obtain a boot log from the unsuccessful boot by pressing F8 to access the special boot menu and choosing the boot logging option. As described earlier in this chapter, Session Manager (\Windows\System32\Smss.exe) saves a log of the boot that includes a record of device drivers that the system loaded and chose not to load to \Windows\ntbtlog.txt, so you'll obtain a boot log if the crash or hang occurs after Session Manager initializes. When you reboot into safe mode, the system appends new entries to the existing boot log. Extract the portions of the log file that refer to the failed attempt, and safe mode boots into separate files. Strip out lines that contain the text "Did not load driver", and then compare them with a text comparison tool such as Windiff. One by one, disable the drivers that loaded during the normal boot but not in the safe-mode boot until the system boots successfully again. (Then re-enable the drivers that were not responsible for the problem.)

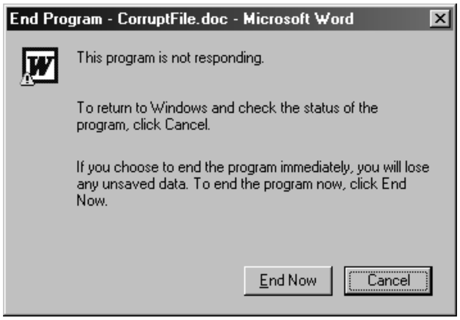
If you cannot obtain a boot log from the normal boot (for instance, because the system is crashing before Session Manager initializes), if the system also crashes during the safe-mode boot, or if a comparison of boot logs from the normal and safe-mode boots do not reveal any significant differences (for example, when the driver that's crashing the normal boot starts after Session Manager initializes), the next tool to try is the Driver Verifier combined with crash dump analysis. (See Chapter 14 for more information on both these topics.)

Shutdown

If someone is logged on and a process initiates a shutdown by calling the Windows *ExitWindowsEx* function, a message is sent to Csrss instructing it to perform the shutdown. Csrss in turn impersonates the caller and sends a Windows message to a hidden window owned by Winlogon, telling it to perform a system shutdown. Winlogon then impersonates the currently logged-on user (who might or might not have the same security context as the user who initiated the system shutdown) and calls *ExitWindowsEx* with some special internal flags. Again, this call causes a message to be sent to Csrss requesting a system shutdown.

This time, Csrss sees that the request is from Winlogon and loops through all the processes in the logon session of the interactive user (again, not the user who requested a shutdown) in reverse order of their *shutdown level*. A process can specify a shutdown level, which indicates to the system when they want to exit with respect to other processes, by calling *SetProcessShutdownParameters*. Valid shutdown levels are in the range 0 through 1023, and the default level is 640. Explorer, for example, sets its shutdown level to 2 and Task Manager specifies 1. For each process that owns a top-level window, Csrss sends the WM_QUERYENDSESSION message to each thread in the process that has a Windows message loop. If the thread returns TRUE, the system shutdown can proceed. Csrss then sends the WM_ENDSESSION Windows message to the thread to request it to exit. Csrss waits the number of seconds defined in HKCU\Control Panel\Desktop\HungAppTimeout for the thread to exit. (The default is 5000 milliseconds.)

If the thread doesn't exit before the timeout, Csrss displays the hung-program dialog box shown in Figure 5-5. (You can disable this dialog box by changing the registry value HKCU\Control Panel\Desktop\AutoEndTasks to 1.) This dialog box indicates that a program isn't shutting down in a timely manner and gives the user a choice of either killing the process or aborting the shutdown. (There is no timeout on this dialog box, which means that a shutdown request could wait forever at this point.)



End Program - CorruptFile.doc - Microsoft Word

This program is not responding.

To return to Windows and check the status of the program, click Cancel.

If you choose to end the program immediately, you will lose any unsaved data. To end the program now, click End Now.

Figure 5-5 Hung program dialog box

If the thread does exit before the timeout, Csrss continues sending the WM_QUERYENDSESSION/WM_ENDSESSION message pairs to the other threads in the process that own windows. Once all the threads that own windows in the process have exited, Csrss terminates the process and goes on to the next process in the interactive session.

EXPERIMENT: Witnessing the HungAppTimeout

You can see the use of the HungAppTimeout registry value by running Notepad, entering text into its editor, and then logging off. After the amount of time specified by the HungAppTimeout registry value has expired, Csrss.exe presents a dialog box that asks you whether or not you want to end the Notepad process, which has not exited because it's waiting for you to tell it whether or not to save the entered text to a file. If you press the Cancel button on the dialog box, Csrss.exe aborts the shutdown.

If Csrss finds a console application, it invokes the console control handler by sending the CTRL_LOGOFF_EVENT event. (Only service processes receive the CTRL_SHUTDOWN_EVENT event on shutdown.) If the handler returns FALSE, Csrss kills the process. If the handler returns TRUE or doesn't respond by the number of seconds defined by HKCU\Control Panel\Desktop\WaitToKillAppTimeout (the default is 20,000 milliseconds), Csrss displays the hung-program dialog box shown in Figure 5-5.

Next, Winlogon calls *ExitWindowsEx* to have Csrss terminate any COM processes that are part of the interactive user's session.

At this point, all the processes in the interactive user's session have been terminated. Winlogon calls *ExitWindowsEx* again, but this time in the system process context, which again sends a message to Csrss, which looks at all the processes belonging to the system context and performs and sends the WM_ QUERYENDSESSION/WM_ENDSESSION messages to GUI threads (as before). Instead of sending CTRL_LOGOFF_EVENT, however, it sends CTRL_ SHUTDOWN_EVENT to console applications that have registered control handlers. Note that the SCM is a console program that does register a control handler. When it receives the shutdown request, it in turn sends the service shutdown control message to all services that registered for shutdown notification. For more details on service shutdown (such as the shutdown timeout Csrss uses for the SCM), see the "Services" section in Chapter 4.

Although Csrss performs the same timeouts as when it was terminating the user processes, it doesn't display any dialog boxes and doesn't kill any processes. (The registry values for the system process timeouts are taken from the default user profile.) These timeouts simply allow system processes a chance to clean up and exit before the system shuts down. Therefore, many system processes are in fact still running when the system shuts down, such as Smss, Winlogon, the SCM, and Lsass.

Once Csrss has finished its pass notifying system processes that the system is shutting down, Winlogon finishes the shutdown process by calling the executive subsystem function *NtShutdownSystem*. This function calls the function *NtSetSystemPowerState* to orchestrate the shutdown of drivers and the rest of the executive subsystems (Plug and Play manager, power manager, executive, I/O manager, configuration manager, and memory manager).

For example, *NtSetSystemPowerState* calls the I/O manager to send shutdown I/O packets to all device drivers that have requested shutdown notification. This action gives device drivers a chance to perform any special processing their device might require before Windows exits. The configuration manager flushes any modified registry data to disk, and the memory manager writes all modified pages containing file data back to their respective files. If the option to clear the paging file at shutdown is enabled, the memory manager clears the paging file at this time. The I/O manager is called a second time to inform the file system drivers that the system is shutting down. System shutdown ends in the power manager. The action the power manager takes depends on whether the user specified a shutdown, a reboot, or a power down.

Conclusion

In this chapter, we've examined the detailed steps involved in starting and shutting down Windows (both normally and in error cases). So far, we've examined the overall structure of Windows and the core system mechanisms that get the system going, keep it running, and eventually shut it down. With this foundation laid, we're ready to explore the individual executive components in more detail, starting with processes and threads.

Chapter 6

Processes, Threads, and Jobs

In this chapter, we'll explain the data structures and algorithms that deal with processes, threads, and jobs in Microsoft Windows. The first section focuses on the internal structures that make up a process. The second section outlines the steps involved in creating a process (and its initial thread). The internals of threads and thread scheduling are then described. The chapter concludes with a description of the job object.

Where relevant performance counters or kernel variables exist, they are mentioned. Although this book isn't a Windows programming book, the pertinent process, thread, and job Windows functions are listed so that you can pursue additional information on their use.

Because processes and threads touch so many components in Windows, a number of terms and data structures (such as working sets, objects and handles, system memory heaps, and so on) are referred to in this chapter but are explained in detail elsewhere in the book. To fully understand this chapter, you need to be familiar with the terms and concepts explained in chapters 1 and 2, such as the difference between a process and a thread, the Windows virtual address space layout, and the difference between user mode and kernel mode.

Process Internals

This section describes the key Windows process data structures. Also listed are key kernel variables, performance counters, and functions and tools that relate to processes.

Data Structures

Each Windows process is represented by an executive process (EPROCESS) block. Besides containing many attributes relating to a process, an EPROCESS block contains and points to a number of other related data structures. For example, each process has one or more threads represented by executive thread (ETHREAD) blocks. (Thread data structures are explained in the section "Thread Internals" later in this chapter.) The EPROCESS block and its related data structures exist in system space, with the exception of the process environment block (PEB), which exists in the process address space (because it contains information that is modified by user-mode code).

In addition to the EPROCESS block, the Windows subsystem process (Csrss) maintains a parallel structure for each Windows process that executes a Windows program. Also, the kernel-mode part of the Windows subsystem (Win32k.sys) has a per-process data structure that is created the first time a thread calls a Windows USER or GDI function that is implemented in kernel mode.

Figure 6-1 is a simplified diagram of the process and thread data structures. Each data structure shown in the figure is described in detail in this chapter.

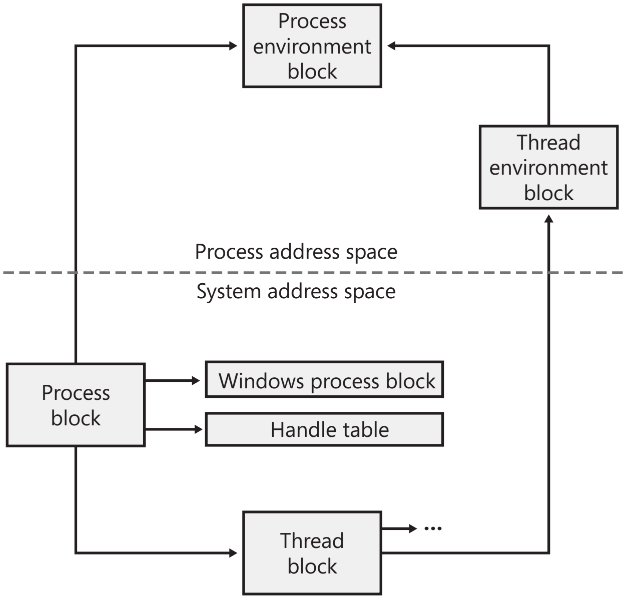


Figure 6-1 Data structures associated with processes and threads

First let's focus on the process block. (We'll get to the thread block in the section "Thread Internals" later in the chapter.) Figure 6-2 shows the key fields in an EPROCESS block.

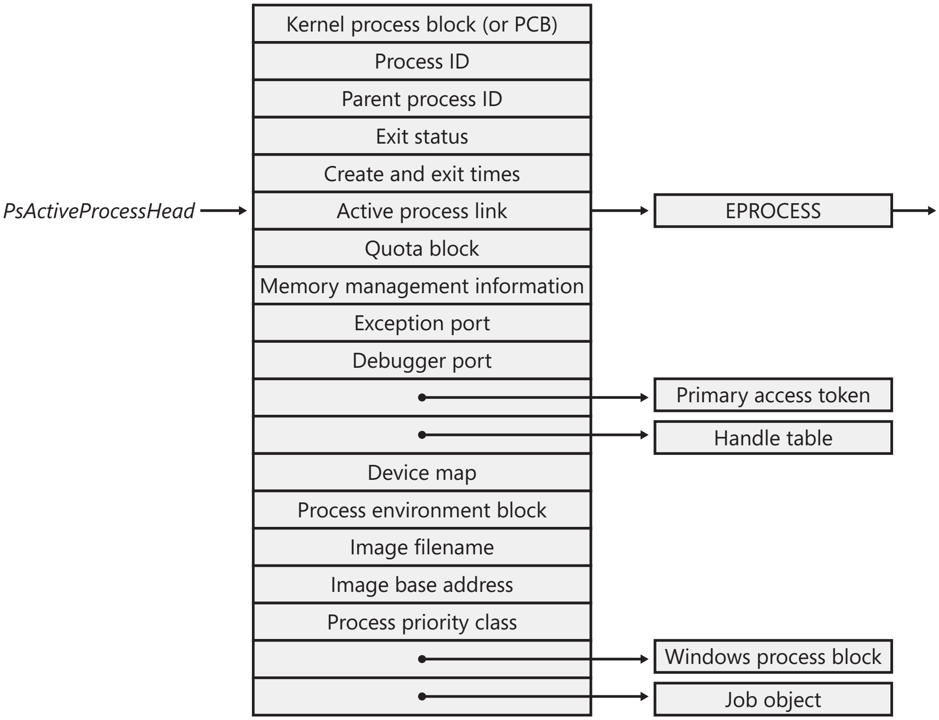


Figure 6-2 Structure of an executive process block

EXPERIMENT: Displaying the Format of an EPROCESS Block

For a list of the fields that make up an EPROCESS block and their offsets in hexadecimal, type **dt _eprocess** in the kernel debugger. (See Chapter 1 for more information on the kernel debugger and how to perform kernel debugging on the local system.) The output (truncated for the sake of space) looks like this:

```
lkd> dt _eprocess
nt!_EPROCESS
   +0x000 Pcb              : _KPROCESS
   +0x06c ProcessLock      : _EX_PUSH_LOCK
   +0x070 CreateTime       : _LARGE_INTEGER
   +0x078 ExitTime         : _LARGE_INTEGER
   +0x080 RundownProtect   : _EX_RUNDOWN_REF
   +0x084 UniqueProcessId  : Ptr32 Void
   +0x088 ActiveProcessLinks : _LIST_ENTRY
   +0x090 QuotaUsage       : [3] Uint4B
   +0x09c QuotaPeak        : [3] Uint4B
   +0x0a8 CommitCharge     : Uint4B
   +0x0ac PeakVirtualSize  : Uint4B
   +0x0b0 VirtualSize      : Uint4B
   +0x0b4 SessionProcessLinks : _LIST_ENTRY
   +0x0bc DebugPort        : Ptr32 Void
   +0x0c0 ExceptionPort    : Ptr32 Void
   +0x0c4 ObjectTable      : Ptr32 _HANDLE_TABLE
   +0x0c8 Token            : _EX_FAST_REF
   +0x0cc WorkingSetLock   : _FAST_MUTEX
   +0x0ec WorkingSetPage   : Uint4B
   +0x0f0 AddressCreationLock : _FAST_MUTEX
   +0x110 HyperSpaceLock   : Uint4B
   +0x114 ForkInProgress   : Ptr32 _ETHREAD
   +0x118 HardwareTrigger  : Uint4B
```

Note that the first field (Pcb) is actually a substructure, the kernel process block (KPROCESS), which is where scheduling-related information is stored. To display the format of the kernel process block, type **dt_kprocess**:

```
lkd> dt _kprocess
nt!_KPROCESS
   +0x000 Header           : _DISPATCHER_HEADER
   +0x010 ProfileListHead  : _LIST_ENTRY
   +0x018 DirectoryTableBase : [2] Uint4B
   +0x020 LdtDescriptor    : _KGDTENTRY
   +0x028 Int21Descriptor  : _KIDTENTRY
   +0x030 IopmOffset       : Uint2B
   +0x032 Iopl             : UChar
   +0x033 Unused           : UChar
   +0x034 ActiveProcessors : Uint4B
   +0x038 KernelTime       : Uint4B
   +0x03c UserTime         : Uint4B
   +0x040 ReadyListHead    : _LIST_ENTRY
   +0x048 SwapListEntry    : _SINGLE_LIST_ENTRY
   +0x04c VdmTrapcHandler  : Ptr32 Void
   +0x050 ThreadListHead   : _LIST_ENTRY
   +0x058 ProcessLock      : Uint4B
   +0x05c Affinity         : Uint4B
   +0x060 StackCount       : Uint2B
   +0x062 BasePriority     : Char
   +0x063 ThreadQuantum    : Char
   +0x064 AutoAlignment    : UChar
   +0x065 State            : UChar
   +0x066 ThreadSeed       : UChar
   +0x067 DisableBoost     : UChar
   +0x068 PowerState       : UChar
   +0x069 DisableQuantum   : UChar
   +0x06a IdealNode        : UChar
   +0x06b Spare            : UChar
```

An alternate way to see the KPROCESS (and other substructures in the EPROCESS) is to use the recursion (*-r*) switch of the *dt* command. For example, typing **dt _eprocess –r1** will recurse and display all substructures one level deep.

The *dt* command shows the format of a process block, not its contents. To show an instance of an actual process, you can specify the address of an EPROCESS structure as an argument to the *dt* command. You can get the address of all the EPROCESS blocks in the system by using the *!process 0 0* command. An annotated example of the output from this command is included later in this chapter.

Table 6-1 explains some of the fields in the preceding experiment in more detail and includes references to other places in the book where you can find more information about them. As we've said before and will no doubt say again, processes and threads are such an integral part of Windows that it's impossible to talk about them without referring to many other parts of the system. To keep the length of this chapter manageable, however, we've covered those related subjects (such as memory management, security, objects, and handles) elsewhere.

Table 6-1 Contents of the EPROCESS Block

Element	Purpose	Additional Reference
Kernel process (KPROCESS) block	Common dispatcher object header, pointer to the process page directory, list of kernel thread (KTHREAD) blocks belonging to the process, default base priority, quantum, affinity mask, and total kernel and user time for the threads in the process.	Thread Scheduling (Chapter 6)
Process identification	Unique process ID, creating process ID, name of image being run, window station process is running on.	
Quota block	Limits on nonpaged pool, paged pool, and page file usage plus current and peak process nonpaged and paged pool usage. (*Note*: Several processes can share this structure: all the system processes point to the single systemwide default quota block; all the processes in the interactive session share a single quota block that Winlogon sets up.)	
Virtual address descriptors (VADs)	Series of data structures that describes the status of the portions of the address space that exist in the process.	Virtual Address Descriptors (Chapter 7)
Working set information	Pointer to working set list (MMWSL structure); current, peak, minimum, and maximum working set size; last trim time; page fault count; memory priority; outswap flags; page fault history.	Working Sets (Chapter 7)
Virtual memory information	Current and peak virtual size, page file usage, hardware page table entry for process page directory.	Chapter 7
Exception local procedure call (LPC) port	Interprocess communication channel to which the process manager sends a message when one of the process's threads causes an exception.	Exception Dispatching (Chapter 3)

Table 6-1 Contents of the EPROCESS Block

Element	Purpose	Additional Reference
Debugging LPC port	Interprocess communication channel to which the process manager sends a message when one of the process's threads causes a debug event.	Local Procedure Calls (LPCs) (Chapter 3)
Access token (ACCESS_TOKEN)	Executive object describing the security profile of this process.	Chapter 8
Handle table	Address of per-process handle table.	Object Handles and the Process Handle Table (Chapter 3)
Device map	Address of object directory to resolve device name references in (supports multiple users).	Object Names (Chapter 3)
Process environment block (PEB)	Image information (base address, version numbers, module list), process heap information, and thread-local storage utilization. (*Note*: The pointers to the process heaps start at the first byte after the PEB.)	Chapter 6
Windows subsystem process block (W32PROCESS)	Process details needed by the kernel-mode component of the Windows subsystem.	

The kernel process (KPROCESS) block, which is part of the EPROCESS block, and the process environment block (PEB), which is pointed to by the EPROCESS block, contain additional details about the process object. The KPROCESS block (which is sometimes called the PCB, or process control block) is illustrated in Figure 6-3. It contains the basic information that the Windows kernel needs to schedule threads. (Page directories are covered in Chapter 7, and kernel thread blocks are described in more detail later in this chapter.)

The PEB, which lives in the user process address space, contains information needed by the image loader, the heap manager, and other Windows system DLLs that need to modify it from user mode. (The EPROCESS and KPROCESS blocks are accessible only from kernel mode.) The basic structure of the PEB is illustrated in Figure 6-4 and is explained in more detail later in this chapter.

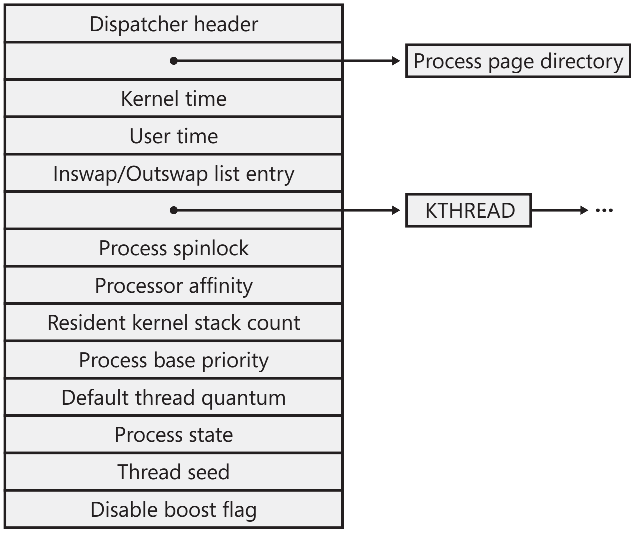


Figure 6-3 Structure of the executive process block

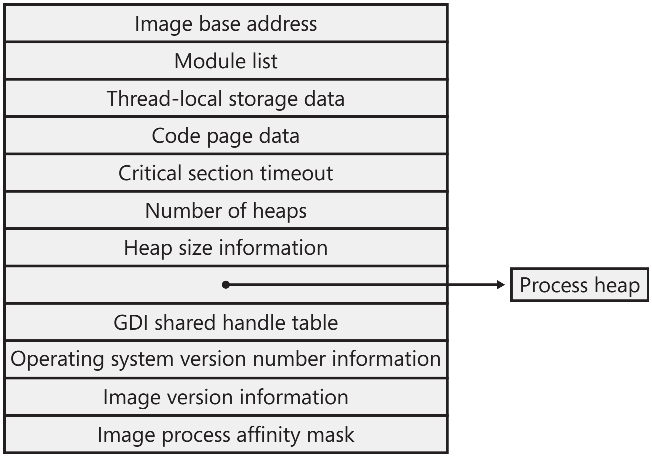


Figure 6-4 Fields of the process environment block

EXPERIMENT: Examining the PEB

You can dump the PEB structure with the *!peb* command in the kernel debugger. To get the address of the PEB, use the *!process* command as follows:

```
lkd> !process
PROCESS 8575f030  SessionId: 0  Cid: 08d0    Peb: 7ffdf000  ParentCid: 0360
    DirBase: 1a81b000  ObjectTable: e12bd418  HandleCount:  66.
    Image: windbg.exe
```

Then specify that address to the *!peb* command as follows:

```
lkd> !peb 7ffdf000
PEB at 7ffdf000
    InheritedAddressSpace:    No
    ReadImageFileExecOptions: No
    BeingDebugged:            No
    ImageBaseAddress:         01000000
    Ldr                       00181e90
    Ldr.Initialized:          Yes
    Ldr.InInitializationOrderModuleList: 00181f28 . 00183188
    Ldr.InLoadOrderModuleList:           00181ec0 . 00183178
    Ldr.InMemoryOrderModuleList:         00181ec8 . 00183180
            Base TimeStamp                     Module
         1000000 40478dbd Mar 04 15:12:45 2004 C:\Program Files\Debugging Tools for
             Windows\windbg.exe
        77f50000 3eb1b41a May 01 19:56:10 2003 C:\WINDOWS\System32\ntdll.dll
        77e60000 3d6dfa28 Aug 29 06:40:40 2002 C:\WINDOWS\system32\kernel32.dll
         2000000 40476db2 Mar 04 12:56:02 2004 C:\Program Files\Debugging Tools for
             Windows\dbgeng.dll
        .
    SubSystemData:     00000000
    ProcessHeap:       00080000
    ProcessParameters: 00020000
    WindowTitle:  'C:\Documents and Settings\All Users\Start Menu\Programs\Debugging
        Tools for Windows\WinDbg.lnk'
    ImageFile:    'C:\Program Files\Debugging Tools for Windows\windbg.exe'
    CommandLine:  '"C:\Program Files\Debugging Tools for Windows\windbg.exe" '
    DllPath:      'C:\Program Files\Debugging Tools for Windows;C:\WINDOWS\System32;C:
\WINDOWS\system;C:\WINDOWS;.;C:\Program Files\Windows Resource Kits\Tools\;C:\WINDOWS\
system32;C:\WINDOWS;C:\WINDOWS\System32\Wbem;C:\Program Files\Support Tools\;c:\sysint
;C:\Program Files\ATI Technologies\ATI Control Panel;C:\Program Files\Resource Kit\;C:
\PROGRA~1\CA\Common\SCANEN~1;C:\PROGRA~1\CA\eTrust\ANTIVI~1;C:\Program Files\Common
    Files\Roxio Shared\DLLShared;C:\SFU\common\'
    Environment:  00010000
        =::=::\
        ALLUSERSPROFILE=C:\Documents and Settings\All Users
        APPDATA=C:\Documents and Settings\dsolomon\Application Data
    .
    .
    .
```

Kernel Variables

A few key kernel global variables that relate to processes are listed in Table 6-2. These variables are referred to later in the chapter, when the steps in creating a process are described.

Table 6-2 Process-Related Kernel Variables

Variable	Type	Description
PsActiveProcessHead	Queue header	List head of process blocks
PsIdleProcess	EPROCESS	Idle process block
PsInitialSystemProcess	Pointer to EPROCESS	Pointer to the process block of the initial system process that contains the system threads
PspCreateProcessNotifyRoutine	Array of pointers	Array of pointers to routines to be called on process creation and deletion (maximum of eight)
PspCreateProcessNotifyRoutineCount	DWORD	Count of registered process notification routines
PspLoadImageNotifyRoutine	Array of pointers	Array of pointers to routines to be called on image load
PspLoadImageNotifyRoutineCount	DWORD	Count of registered image-load notification routines
PspCidTable	Pointer to HANDLE_TABLE	Handle table for process and thread client IDs

Performance Counters

Windows maintains a number of counters with which you can track the processes running on your system; you can retrieve these counters programmatically or view them with the Performance tool. Table 6-3 lists the performance counters relevant to processes (except for memory management and I/O-related counters, which are described in Chapters 7 and 9, respectively).

Table 6-3 Process-Related Performance Counters

Object: Counter	Function
Process: % Privileged Time	Describes the percentage of time that the threads in the process have run in kernel mode during a specified interval.
Process: % Processor Time	Describes the percentage of CPU time that the threads in the process have used during a specified interval. This count is the sum of % Privileged Time and % User Time.
Process: % User Time	Describes the percentage of time that the threads in the process have run in user mode during a specified interval.

Table 6-3 Process-Related Performance Counters

Object: Counter	Function
Process: Elapsed Time	Describes the total elapsed time in seconds since this process was created.
Process: ID Process	Returns the process ID. This ID applies only while the process exists because process IDs are reused.
Process: Creating Process ID	Returns the process ID of the creating process. This value isn't updated if the creating process exits.
Process: Thread Count	Returns the number of threads in the process.
Process: Handle Count	Returns the number of handles open in the process.

Relevant Functions

For reference purposes, some of the Windows functions that apply to processes are described in Table 6-4. For further information, consult the Windows API documentation in the MSDN Library.

Table 6-4 Process-Related Functions

Function	Description
CreateProcess	Creates a new process and thread using the caller's security identification
CreateProcessAsUser	Creates a new process and thread with the specified alternate security token
CreateProcessWithLogonW	Creates a new process and thread to run under the credentials of the specified username and password
CreateProcessWithTokenW	Creates a new process and thread with the specified alternate security token, with additional options such as allowing the user profile to be loaded
OpenProcess	Returns a handle to the specified process object
ExitProcess	Ends a process, and notifies all attached DLLs
TerminateProcess	Ends a process without notifying the DLLs
FlushInstructionCache	Empties the specified process's instruction cache
GetProcessTimes	Obtains a process's timing information, describing how much time the process has spent in user and kernel mode
GetExitCodeProcess	Returns the exit code for a process, indicating how and why the process shut down
GetCommandLine	Returns a pointer to the command-line string passed to the current process
GetCurrentProcess	Returns a pseudo handle for the current process
GetCurrentProcessId	Returns the ID of the current process

Table 6-4 Process-Related Functions

Function	Description
GetProcessVersion	Returns the major and minor versions of the Windows version on which the specified process expects to run
GetStartupInfo	Returns the contents of the STARTUPINFO structure specified during *CreateProcess*
GetEnvironmentStrings	Returns the address of the environment block
GetEnvironmentVariable	Returns a specific environment variable
Get/SetProcessShutdownParameters	Defines the shutdown priority and number of retries for the current process
GetGuiResources	Returns a count of User and GDI handles

EXPERIMENT: Using the Kernel Debugger *!process* Command

The kernel debugger *!process* command displays a subset of the information in an EPROCESS block. This output is arranged in two parts for each process. First you see the information about the process, as shown here (when you don't specify a process address or ID, !process lists information for the active process on the current CPU):

```
lkd> !process
PROCESS 8575f030  SessionId: 0  Cid: 08d0    Peb: 7ffdf000  ParentCid: 0360
    DirBase: 1a81b000  ObjectTable: e12bd418  HandleCount:  65.
    Image: windbg.exe
    VadRoot 857f05e0 Vads 71 Clone 0 Private 1152. Modified 98. Locked 1.
    DeviceMap e1e96c88
    Token                             e1f5b8a8
    ElapsedTime                       1:23:06.0219
    UserTime                          0:00:11.0897
    KernelTime                        0:00:07.0450
    QuotaPoolUsage[PagedPool]         38068
    QuotaPoolUsage[NonPagedPool]      2840
    Working Set Sizes (now,min,max)  (2552, 50, 345) (10208KB, 200KB, 1380KB)
    PeakWorkingSetSize                2715
    VirtualSize                       41 Mb
    PeakVirtualSize                   41 Mb
    PageFaultCount                    3658
    MemoryPriority                    BACKGROUND
    BasePriority                      8
    CommitCharge                      1566
```

After the basic process output comes a list of the threads in the process. That output is explained in the "Experiment: Using the Kernel Debugger *!thread* Command" section later in the chapter. Other commands that display process information include *!handle*, which dumps the process handle table (which is described in more detail in the section "Object Handles and the Process Handle Table" in Chapter 3). Process and thread security structures are described in Chapter 8.

Flow of CreateProcess

So far in this chapter, you've seen the structures that are part of a process and the API functions with which you (and the operating system) can manipulate processes. You've also found out how you can use tools to view how processes interact with your system. But how did those processes come into being, and how do they exit once they've fulfilled their purpose? In the following sections, you'll discover how a Windows process comes to life.

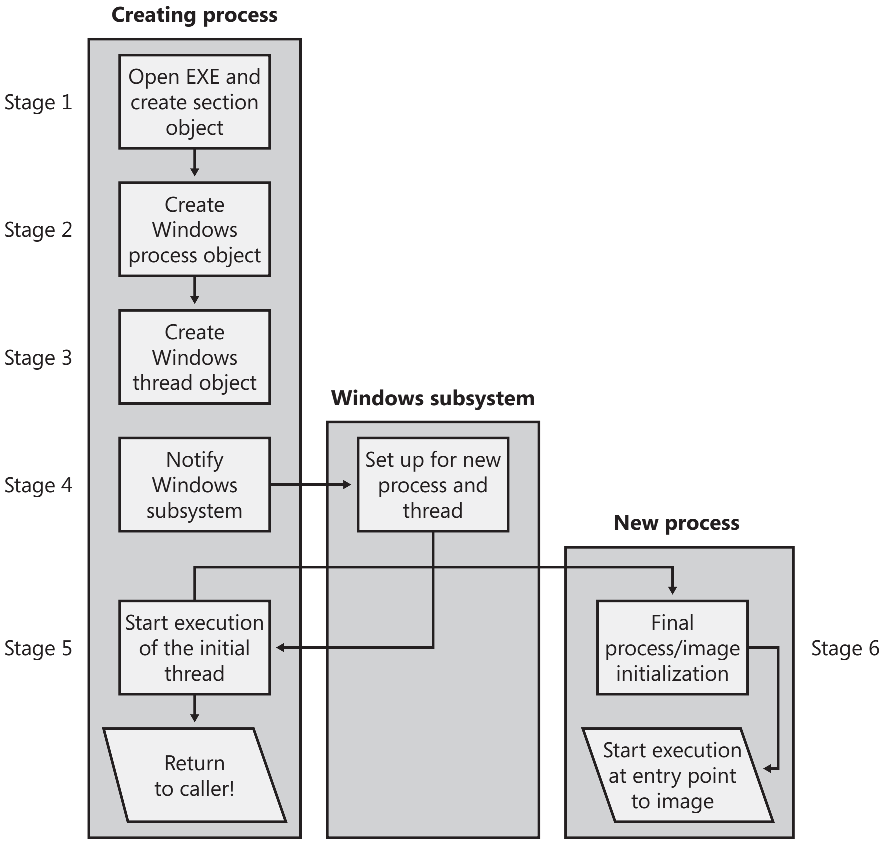
A Windows process is created when an application calls one of the process creation functions, such as *CreateProcess*, *CreateProcessAsUser*, *CreateProcessWithTokenW*, or *CreateProcessWithLogonW*. Creating a Windows process consists of several stages carried out in three parts of the operating system: the Windows client-side library Kernel32.dll, the Windows executive, and the Windows subsystem process (Csrss). Because of the multiple environment subsystem architecture of Windows, creating a Windows executive process object (which other subsystems can use) is separated from the work involved in creating a Windows process. So, although the following description of the flow of the Windows *CreateProcess* function is complicated, keep in mind that part of the work is specific to the semantics added by the Windows subsystem as opposed to the core work needed to create a Windows executive process object.

The following list summarizes the main stages of creating a process with the Windows *CreateProcess* function. The operations performed in each stage are described in detail in the subsequent sections.

> **Note** Many steps of *CreateProcess* are related to the setup of the process virtual address space and therefore refer to many memory management terms and structures that are defined in Chapter 7.

1. Open the image file (.exe) to be executed inside the process.
2. Create the Windows executive process object.
3. Create the initial thread (stack, context, and Windows executive thread object).
4. Notify the Windows subsystem of the new process so that it can set up for the new process and thread.
5. Start execution of the initial thread (unless the CREATE_ SUSPENDED flag was specified).
6. In the context of the new process and thread, complete the initialization of the address space (such as load required DLLs) and begin execution of the program.

Figure 6-5 shows an overview of the stages Windows follows to create a process.



Creating process
Stage 1
Open EXE and create section object
Stage 2
Create Windows process object
Stage 3
Create Windows thread object
Windows subsystem
Stage 4
Notify Windows subsystem
Set up for new process and thread
New process
Stage 5
Start execution of the initial thread
Final process/image initialization
Stage 6
Return to caller!
Start execution at entry point to image

Figure 6-5 The main stages of process creation

Before opening the executable image to run, *CreateProcess* performs the following steps:

- In *CreateProcess*, the priority class for the new process is specified as independent bits in the *CreationFlags* parameter. Thus, you can specify more than one priority class for a single *CreateProcess* call. Windows resolves the question of which priority class to assign to the process by choosing the lowest-priority class set.
- If no priority class is specified for the new process, the priority class defaults to Normal unless the priority class of the process that created it is Idle or Below Normal, in which case the priority class of the new process will have the same priority as the creating class.
- If a Real-time priority class is specified for the new process and the process's caller doesn't have the Increase Scheduling Priority privilege, the High priority class is used instead. In other words, *CreateProcess* doesn't fail just because the caller has insufficient privileges to create the process in the Real-time priority class; the new process just won't have as high a priority as Real-time.
- All windows are associated with desktops, the graphical representation of a workspace. If no desktop is specified in *CreateProcess*, the process is associated with the caller's current desktop.

Stage 1: Opening the Image to Be Executed

As illustrated in Figure 6-6, the first stage in *CreateProcess* is to find the appropriate Windows image that will run the executable file specified by the caller and to create a section object to later map it into the address space of the new process. If no image name is specified, the first token of the command line (defined to be the first part of the command-line string ending with a space or tab that is a valid file specification) is used as the image filename.

On Windows XP and Windows Server 2003, *CreateProcess* checks whether software restriction policies on the machine prevent the image from being run. (See Chapter 8 for a complete description of software restriction policies.)

If the executable file specified is a Windows .exe, it is used directly. If it's not a Windows .exe (for example, if it's an MS-DOS, Win16, or a POSIX application), *CreateProcess* goes through a series of steps to find a Windows *support image* to run it. This process is necessary because non-Windows applications aren't run directly—Windows instead uses one of a few special support images that in turn are responsible for actually running the non-Windows program. For example, if you attempt to run a POSIX application, *CreateProcess* identifies it as such and changes the image to be run on the Windows executable file Posix.exe. If you attempt to run an MS-DOS or a Win16 executable, the image to be run becomes the Windows executable Ntvdm.exe. In short, you can't directly create a process that is *not* a Windows process. If Windows can't find a way to resolve the activated image as a Windows process (as shown in Table 6-5), *CreateProcess* fails.

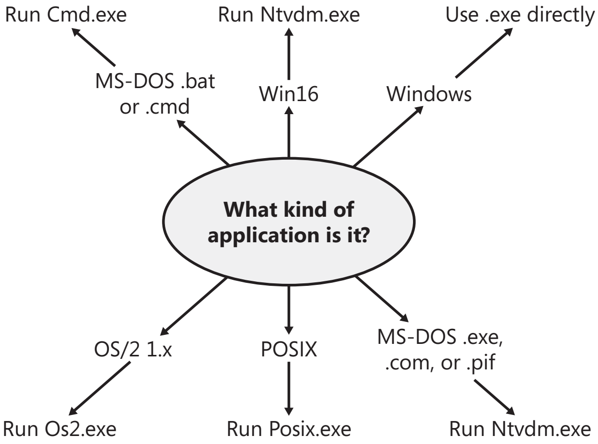


Figure 6-6 Choosing a Windows image to activate

Table 6-5 **Decision Tree for Stage 1 of *CreateProcess***

If the image is a/an	And this will happen	This image will run
POSIX executable file	Posix.exe	*CreateProcess* restarts Stage 1.
MS-DOS application with an .exe, a .com, or a .pif extension	Ntvdm.exe	*CreateProcess* restarts Stage 1.
Win16 application	Ntvdm.exe	*CreateProcess* restarts Stage 1.
Command procedure (application with a .bat or a .cmd extension)	Cmd.exe	*CreateProcess* restarts Stage 1.

Specifically, the decision tree that *CreateProcess* goes through to run an image is as follows:

- If the image is an MS-DOS application with an .exe, a .com, or a .pif extension, a message is sent to the Windows subsystem to check whether an MS-DOS support process (Ntvdm.exe, specified in the registry value HKLM\SYSTEM\CurrentControlSet\Control\WOW\ cmdline) has already been created for this session. If a support process has been created, it is used to run the MS-DOS application. (The Windows subsystem sends the message to the VDM [Virtual DOS Machine] process to run the new image.) Then *CreateProcess* returns. If a support process hasn't been created, the image to be run changes to Ntvdm.exe and *CreateProcess* restarts at Stage 1.
- If the file to run has a .bat or a .cmd extension, the image to be run becomes Cmd.exe, the Windows command prompt, and *CreateProcess* restarts at Stage 1. (The name of the batch file is passed as the first parameter to Cmd.exe.)
- If the image is a Win16 (Windows 3.1) executable, *CreateProcess* must decide whether a new VDM process must be created to run it or whether it should use the default session-wide shared VDM process (which might not yet have been created). The *CreateProcess* flags CREATE_SEPARATE_WOW_VDM and CREATE_SHARED_WOW_VDM control this decision. If these flags aren't specified, the registry value HKLM\SYSTEM\CurrentControlSet\Control\WOW\ DefaultSeparateVDM dictates the default behavior. If the application is to be run in a separate VDM, the image to be run changes to the value of HKLM\SYSTEM\CurrentControlSet\Control\WOW\wowcmdline and *CreateProcess* restarts at Stage 1. Otherwise, the Windows subsystem sends a message to see whether the shared VDM process exists and can be used. (If the VDM process is running on a different desktop or isn't running under the same security as the caller, it can't be used and a new VDM process must be created.) If a shared VDM process can be used, the Windows subsystem sends a message to it to run the new image and *CreateProcess* returns. If the VDM process hasn't yet been created (or if it exists but can't be used), the image to be run changes to the VDM support image and *CreateProcess* restarts at Stage 1.

At this point, *CreateProcess* has successfully opened a valid Windows executable file and created a section object for it. The object isn't mapped into memory yet, but it is open. Just because a section object has been successfully created doesn't mean that the file is a valid Windows image, however; it could be a DLL or a POSIX executable. If the file is a POSIX executable, the image to be run changes to Posix.exe and *CreateProcess* restarts from the beginning of Stage 1. If the file is a DLL, *CreateProcess* fails.

Now that *CreateProcess* has found a valid Windows executable image, it looks in the registry under HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options to see whether a subkey with the filename and extension of the executable image (but without the directory and path information—for example, Image.exe) exists there. If it does, *CreateProcess* looks for a value named Debugger for that key. If this is present, the image to be run becomes the string in that value and *CreateProcess* restarts at Stage 1.

Tip You can take advantage of this *CreateProcess* behavior and debug the startup code of Windows service processes before they start rather than attach the debugger after starting the service, which doesn't allow you to debug the startup code.

Stage 2: Creating the Windows Executive Process Object

At this point, *CreateProcess* has opened a valid Windows executable file and created a section object to map it into the new process address space. Next it creates a Windows executive process object to run the image by calling the internal system function *NtCreateProcess*. Creating the executive process object (which is done by the creating thread) involves the following substages:

- Setting up the EPROCESS block
- Creating the initial process address space
- Initializing the kernel process block (KPROCESS)
- Concluding the setup of the process address space (which includes initializing the working set list and virtual address space descriptors and mapping the image into address space)
- Setting up the PEB
- Completing the setup of the executive process object

Note The only time there won't be a parent process is during system initialization. After that point, a parent process is always required to provide a security context for the new process.

Stage 2A: Setting Up the EPROCESS Block

This substage involves nine steps:

1. Allocate and initialize the Windows EPROCESS block.
2. Inherit the process affinity mask from the parent process.
3. The process minimum and maximum working set size are set to the values of *PsMinimumWorkingSet* and *PsMaximumWorkingSet*, respectively.
4. Set the new process's quota block to the address of its parent process's quota block, and increment the reference count for the parent's quota block.
5. Inherit the Windows device name space (including the definition of drive letters, COM ports, and so on).
6. Store the parent process's process ID in the *InheritedFromUniqueProcessId* field in the new process object.
7. Create the process's primary access token (a duplicate of its parent's primary token). New processes inherit the security profile of their parents. If the *CreateProcessAsUser* function is being used to specify a different access token for the new process, the token is then changed appropriately.
8. The process handle table is initialized. If the inherit handles flag is set for the parent process, any inheritable handles are copied from the parent's object handle table into the new process. (For more information about object handle tables, see Chapter 3.)
9. Set the new process's exit status to STATUS_PENDING.

Stage 2B: Creating the Initial Process Address Space

The initial process address space consists of the following pages:

- Page directory (and it's possible there'll be more than one for systems with page tables more than two levels, such as x86 systems in PAE mode or 64-bit systems)
- Hyperspace page
- Working set list

To create these three pages, the following steps are taken:

1. Page table entries are created in the appropriate page tables to map the initial pages.

   The number of pages is deducted from the kernel variable *MmTotalCommittedPages* and added to *MmProcessCommit*.
2. The systemwide default process minimum working set size (*PsMinimumWorkingSet*) is deducted from *MmResidentAvailablePages*.
3. The page table pages for the nonpaged portion of system space and the system cache are mapped into the process.

Stage 2C: Creating the Kernel Process Block

The next stage of *CreateProcess* is the initialization of the KPROCESS block, which contains a pointer to a list of kernel threads. (The kernel has no knowledge of handles, so it bypasses the object table.) The kernel process block also points to the process's page table directory (which is used to keep track of the process's virtual address space), the total time the process's threads have executed, the process's default base-scheduling priority (which starts as Normal, or 8, unless the parent process was set to Idle or Below Normal, in which case the setting is inherited), the default processor affinity for the threads in the process, and the initial value of the process default quantum (which is described in more detail in the "Thread Scheduling" section later in the chapter), which is taken from the value of *PspForegroundQuantum[0]*, the first entry in the systemwide quantum array.

Note The default initial quantum differs between Windows client and server systems. For more information on thread quantums, turn to their discussion in the section "Thread Scheduling."

Stage 2D: Concluding the Setup of the Process Address Space

Setting up the address space for a new process is somewhat complicated, so let's look at what's involved one step at a time. To get the most out of this section, you should have some familiarity with the internals of the Windows memory manager, which are described in Chapter 7.

- The virtual memory manager sets the value of the process's last trim time to the current time. The working set manager (which runs in the context of the balance set manager system thread) uses this value to determine when to initiate working set trimming.
- The memory manager initializes the process's working set list—page faults can now be taken.
- The section (created when the image file was opened) is now mapped into the new process's address space, and the process section base address is set to the base address of the image.
- Ntdll.dll is mapped into the process.
- The systemwide national language support (NLS) tables are mapped into the process's address space.

Note POSIX processes clone the address space of their parents, so they don't have to go through these steps to create a new address space. In the case of POSIX applications, the new process's section base address is set to that of its parent process and the parent's PEB is cloned for the new process.

Stage 2E: Setting Up the PEB

CreateProcess allocates a page for the PEB and initializes a number of fields, which are described in Table 6-6.

Table 6-6 Initial Values of the Fields of the PEB

Field	Initial Value
ImageBaseAddress	Base address of section
NumberOfProcessors	*KeNumberProcessors* kernel variable
NtGlobalFlag	*NtGlobalFlag* kernel variable
CriticalSectionTimeout	*MmCriticalSectionTimeout* kernel variable
HeapSegmentReserve	*MmHeapSegmentReserve* kernel variable
HeapSegmentCommit	*MmHeapSegmentCommit* kernel variable
HeapDeCommitTotalFreeThreshold	*MmHeapDeCommitTotalFreeThreshold* kernel variable
HeapDeCommitFreeBlockThreshold	*MmHeapDeCommitFreeBlockThreshold* kernel variable
NumberOfHeaps	0
MaximumNumberOfHeaps	(Size of a page - size of a PEB) / 4
ProcessHeaps	First byte after PEB
OSMajorVersion	*NtMajorVersion* kernel variable
OSMinorVersion	*NtMinorVersion* kernel variable
OSBuildNumber	*NtBuildNumber* kernel variable & 0x3FFF
OSPlatformId	2

If the image file specifies explicit Windows version values, this information replaces the initial values shown in Table 6-6. The mapping from image version information fields to PEB fields is described in Table 6-7.

Table 6-7 Windows Replacements for Initial PEB Values

Field Name	Value Taken from Image Header
OSMajorVersion	OptionalHeader.Win32VersionValue & 0xFF
OSMinorVersion	(OptionalHeader.Win32VersionValue >> 8) & 0xFF
OSBuildNumber	(OptionalHeader.Win32VersionValue >> 16) & 0x3FFF
OSPlatformId	(OptionalHeader.Win32VersionValue >> 30) ^ 0x2

Stage 2F: Completing the Setup of the Executive Process Object

Before the handle to the new process can be returned, a few final setup steps must be completed:

1. If systemwide auditing of processes is enabled (either as a result of local policy settings or group policy settings from a domain controller), the process's creation is written to the Security event log.

2. If the parent process was contained in a job, the new process is added to the job. (Jobs are described at the end of this chapter.)
3. If the image header characteristic's IMAGE_FILE_UP_SYSTEM_ ONLY flag is set (indicating that the image can run only on a uniprocessor system), a single CPU is chosen for all the threads in this new process to run on. This choosing process is done by simply cycling through the available processors—each time this type of image is run, the next processor is used. In this way, these types of images are spread out across the processors evenly.
4. If the image specifies an explicit processor affinity mask (for example, a field in the configuration header), this value is copied to the PEB and later set as the default process affinity mask.
5. *CreateProcess* inserts the new process block at the end of the Windows list of active processes (*PsActiveProcessHead*).
6. The process's creation time is set, the handle to the new process is returned to the caller (*CreateProcess* in Kernel32.dll).

Stage 3: Creating the Initial Thread and Its Stack and Context

At this point, the Windows executive process object is completely set up. It still has no thread, however, so it can't do anything yet. Before the thread can be created, it needs a stack and a context in which to run, so these are set up now. The stack size for the initial thread is taken from the image—there's no way to specify another size.

Now the initial thread can be created, which is done by calling *NtCreateThread*. The thread parameter (which can't be specified in *CreateProcess* but can be specified in *CreateThread*) is the address of the PEB. This parameter will be used by the initialization code that runs in the context of this new thread (as described in Stage 6). However, the thread won't do anything yet—it is created in a suspended state and isn't resumed until the process is completely initialized (as described in Stage 5). *NtCreateThread* calls *PspCreateThread* (a function also used to create system threads) and performs the following steps:

1. The thread count in the process object is incremented.
2. An executive thread block (ETHREAD) is created and initialized.
3. A thread ID is generated for the new thread.
4. The TEB is set up in the user-mode address space of the process.
5. The user-mode thread start address is stored in the ETHREAD. For Windows threads, this is the system-supplied thread startup function in Kernel32.dll (*BaseProcessStart* for the first thread in a process and *BaseThreadStart* for additional threads). The user's specified Windows start address is stored in the ETHREAD block in a different location so that the system-supplied thread startup function can call the user-specified startup function.

6. *KeInitThread* is called to set up the KTHREAD block. The thread's initial and current base priorities are set to the process's base priority, and its affinity and quantum are set to that of the process. This function also sets the initial thread ideal processor. (See the section "Ideal and Last Processor" for a description of how this is chosen.) *KeInitThread* next allocates a kernel stack for the thread and initializes the machine-dependent hardware context for the thread, including the context, trap, and exception frames. The thread's context is set up so that the thread will start in kernel mode in *KiThreadStartup*. Finally, KeInitThread sets the thread's state to Initialized and returns to *PspCreateThread*.
7. Any registered systemwide thread creation notification routines are called.
8. The thread's access token is set to point to the process access token, and an access check is made to determine whether the caller has the right to create the thread. This check will always succeed if you're creating a thread in the local process, but it might fail if you're using *CreateRemoteThread* to create a thread in another process and the process creating the thread doesn't have the debug privilege enabled.
9. Finally, the thread is readied for execution.

Stage 4: Notifying the Windows Subsystem about the New Process

If software restriction policies dictate, a restricted token is created for the new process. At this point, all the necessary executive process and thread objects have been created. Kernel32.dll next sends a message to the Windows subsystem so that it can set up for the new process and thread. The message includes the following information:

- Process and thread handles
- Entries in the creation flags
- ID of the process's creator
- Flag indicating whether the process belongs to a Windows application (so that Csrss can determine whether or not to show the startup cursor)

The Windows subsystem performs the following steps when it receives this message:

1. *CreateProcess* duplicates a handle for the process and thread. In this step, the usage count of the process and the thread is incremented from 1 (which was set at creation time) to 2.
2. If a process priority class isn't specified, *CreateProcess* sets it according to the algorithm described earlier in this section.
3. The Csrss process block is allocated.
4. The new process's exception port is set to be the general function port for the Windows subsystem so that the Windows subsystem will receive a message when an exception occurs in the process. (For further information on exception handling, see Chapter 3.)

5. If the process is being debugged (that is, if it is attached to a debugger process), the process debug port is set to the Windows subsystem's general function port. This setting ensures that Windows will send debug events that occur in the new process (such as thread creation and deletion, exceptions, and so on) as messages to the Windows subsystem so that it can then dispatch the events to the process that is acting as the new process's debugger.
6. The Csrss thread block is allocated and initialized.
7. *CreateProcess* inserts the thread in the list of threads for the process.
8. The count of processes in this session is incremented.
9. The process shutdown level is set to 0x280 (the default process shutdown level—see *SetProcessShutdownParameters* in the MSDN Library documentation for more information).
10. The new process block is inserted into the list of Windows subsystemwide processes.
11. The per-process data structure used by the kernel-mode part of the Windows subsystem (W32PROCESS structure) is allocated and initialized.
12. The application start cursor is displayed. This cursor is the familiar arrow with an hourglass attached—the way that Windows says to the user, "I'm starting something, but you can use the cursor in the meantime." If the process doesn't make a GUI call after 2 seconds, the cursor reverts to the standard pointer. If the process does make a GUI call in the allotted time, *CreateProcess* waits 5 seconds for the application to show a window. After that time, *CreateProcess* will reset the cursor again.

Stage 5: Starting Execution of the Initial Thread

At this point, the process environment has been determined, resources for its threads to use have been allocated, the process has a thread, and the Windows subsystem knows about the new process. Unless the caller specified the CREATE_ SUSPENDED flag, the initial thread is now resumed so that it can start running and perform the remainder of the process initialization work that occurs in the context of the new process (Stage 6).

Stage 6: Performing Process Initialization in the Context of the New Process

The new thread begins life running the kernel-mode thread startup routine *KiThreadStartup*. *KiThreadStartup* lowers the thread's IRQL level from DPC/dispatch level to APC level and then calls the system initial thread routine, *PspUserThreadStartup*. The user-specified thread start address is passed as a parameter to this routine.

On Windows 2000, *PspUserThreadStartup* first enables working set expansion. If the process being created is a debuggee, all threads in the process are suspended. (Threads might have been created during Stage 3.) A create process message is then sent to the process's debug port (which is the Windows subsystem function port, because this is a Windows process) so that the subsystem can deliver the process startup debug event (CREATE_PROCESS_DEBUG_INFO) to the appropriate debugger process. *PspUserThreadStartup* then waits for the Windows subsystem to get the reply from the debugger (via the *ContinueDebugEvent* function). When the Windows subsystem replies, all the threads are resumed.

On Windows XP and Windows Server 2003, *PspUserThreadStartup* checks whether application prefetching is enabled on the system and, if so, calls the logical prefetcher to process the prefetch instruction file (if it exists) and prefetch pages referenced during the first 10 seconds the process started last time. (For details on the prefetcher, see Chapter 3.) Finally, *PspUserThreadStartup* queues a user-mode APC to run the image loader initialization routine (*LdrInitializeThunk* in Ntdll.dll). The APC will be delivered when the thread attempts to return to user mode.

When *PspUserThreadStartup* returns to *KiThreadStartup*, it returns from kernel mode, the APC is delivered, and *LdrInitializeThunk* is called. The *LdrInitializeThunk* routine initializes the loader, heap manager, NLS tables, thread-local storage (TLS) array, and critical section structures. It then loads any required DLLs and calls the DLL entry points with the DLL_PROCESS_ATTACH function code. (See the sidebar "Side-by-Side Assemblies" for a description of a mechanism introduced in Windows XP to address DLL versioning problems.)

Finally, the image begins execution in user mode when the loader initialization returns to the user mode APC dispatcher, which then calls the thread's start function that was pushed on the user stack when the user APC was delivered.

Side-by-Side Assemblies

A problem that has long plagued Windows users is "DLL hell." You enter DLL hell when you install an application that replaces one or more core system DLLs, such as those for common controls, the Microsoft Visual Basic runtime, or MFC. Application installation programs make these replacements to ensure that the application runs properly, but at the same time, updated DLLs might have incompatibilities with other already-installed applications.

Windows 2000 partly addressed DLL hell by preventing the modification of core system DLLs with the Windows File Protection feature, and by allowing applications to use private copies of these core DLLs. To use a private copy of a DLL instead of the one in the system directory, an application's installation must include a file named *Application*.exe.local (where *Application* is the name of the application's executable), which directs the loader to first look for DLLs in that directory. This type of DLL redirection avoids application/DLL incompatibility problems, but it does so at the expense of sharing DLLs, which is one of the points of DLLs in the first place. In addition, any DLLs that are loaded from the list of KnownDLLs (DLLs that are permanently mapped into memory) or that are loaded by those DLLs cannot be redirected using this mechanism.

To further address application and DLL compatibility while allowing sharing, Windows XP introduces shared assemblies. An assembly consists of a group of resources, including DLLs, and an XML manifest file that describes the assembly and its contents. An application references an assembly through the existence of its own XML manifest. The manifest can be a file in the application's installation directory that has the same name as the application with ".manifest" appended (for example, application.exe.manifest), or it can be linked into the application as a resource. The manifest describes the application and its dependence on assemblies.

There are two types of assemblies: private and shared. The difference between the two is that shared assemblies are digitally signed so that corruption or modification of their contents can be detected. In addition, shared assemblies are stored under the \Windows\Winsxs directory, whereas private assemblies are stored in an application's installation directory. Thus, shared assemblies also have an associated catalog file (.cat) that contains its digital signature information. Shared assemblies can be "side-by-side" assemblies because multiple versions of a DLL can reside on a system simultaneously, with applications dependent on a particular version of a DLL always using that particular version.

An assembly's manifest file typically has a name that includes the name of the assembly, version information, some text that represents a unique signature, and the extension ".manifest". The manifests are stored in \Windows\Winsxs\Manifests, and the rest of the assembly's resources are stored in subdirectories of \Windows\Winsxs that have the same name as the corresponding manifest files, with the exception of the trailing .manifest extension.

An example of a shared assembly is version 6 of the Windows common controls DLL, comctl32.dll, which is new to Windows XP. Its manifest file is named \Windows\Winsxs\Manifest\x86_Microsoft.Windows.Common-Controls_6595b64144ccf1df_6.0.0.0_x-ww_1382d70a.manifest. It has an associated catalog file (which is the same name with the .cat extension) and a subdirectory of Winsxs that includes comctl32.dll.

Version 6 of Comctl32.dll includes integration with Windows XP themes, and because applications not written with themes-support in mind might not appear correctly with the new DLL, it's available only to applications that explicitly reference the shared assembly containing it—the version of Comctl32.dll installed in \Windows\System32 is an instance of version 5.x, which is not theme aware. When an application loads, the loader looks for the application's manifest, and if one exists, loads the DLLs from the assemblies specified. DLLs not included in assemblies referenced in the manifest are loaded in the traditional way. Legacy applications, therefore, link against the version in \Windows\System32, whereas theme-aware applications can specify the new version in their manifest.

You can see the effect of a manifest that directs the system to use the new common control library on Windows XP by running the User State Migration Wizard (\Windows\System32\Usmt\Migwiz.exe) with and without its manifest file:

1. Run it, and notice the Windows XP themes on the buttons in the wizard.
2. Open the Migwiz.exe.manifest file in Notepad, and locate the inclusion of the version 6 common control library.
3. Rename the Migwiz.exe.manifest to **Migwiz.exe.manifest.bak**.
4. Rerun the wizard, and notice the unthemed buttons.
5. Restore the manifest file to its original name.

A final advantage that shared assemblies have is that a publisher can issue a publisher configuration, which can redirect all applications that use a particular assembly to use an updated version. Publishers would do this if they were preserving backward compatibility while addressing bugs. Ultimately, however, because of the flexibility inherent in the assembly model, an application could decide to override the new setting and continue to use an older version.

Thread Internals

Now that we've dissected processes, let's turn our attention to the structure of a thread. Unless explicitly stated otherwise, you can assume that anything in this section applies to both user-mode threads and kernel-mode system threads (which are described in Chapter 2).

Data Structures

At the operating-system level, a Windows thread is represented by an executive thread (ETHREAD) block, which is illustrated in Figure 6-7. The ETHREAD block and the structures it points to exist in the system address space, with the exception of the thread environment block (TEB), which exists in the process address space. In addition, the Windows subsystem process (Csrss) maintains a parallel structure for each thread created in a Windows process. Also, for threads that have called a Windows subsystem USER or GDI function, the kernel-mode portion of the Windows subsystem (Win32k.sys) maintains a per-thread data structure (called the W32THREAD structure) that the ETHREAD block points to.

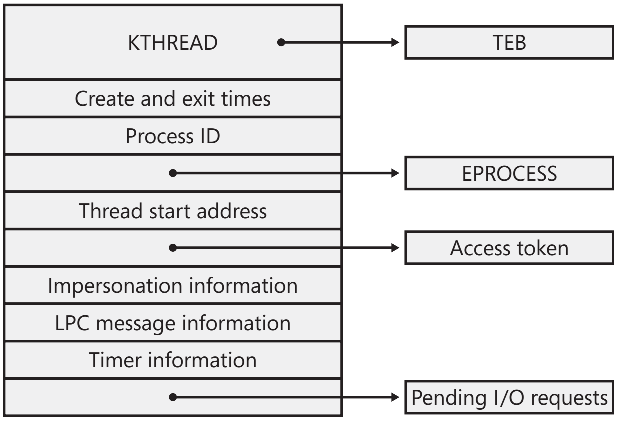


Figure 6-7 Structure of the executive thread block

Most of the fields illustrated in Figure 6-7 are self-explanatory. The first field is the kernel thread (KTHREAD) block. Following that are the thread identification information, the process identification information (including a pointer to the owning process so that its environment information can be accessed), security information in the form of a pointer to the access token and impersonation information, and finally, fields relating to LPC messages and pending I/O requests. As you can see in Table 6-8, some of these key fields are covered in more detail elsewhere in this book. For more details on the internal structure of an ETHREAD block, you can use the kernel debugger *dt* command to display the format of the structure.

Table 6-8 Key Contents of the Executive Thread Block

Element	Description	Additional Reference
KTHREAD	See Table 6-9.	
Thread time	Thread create and exit time information.	
Process identification	Process ID and pointer to EPROCESS block of the process that the thread belongs to.	
Start address	Address of thread start routine.	
Impersonation information	Access token and impersonation level (if the thread is impersonating a client).	Chapter 8
LPC information	Message ID that the thread is waiting for and address of message.	Local procedure calls (Chapter 3)
I/O information	List of pending I/O request packets (IRPs).	I/O system (Chapter 9)

Let’s take a closer look at two of the key thread data structures referred to in the preceding text: the KTHREAD block and the TEB. The KTHREAD block contains the information that the Windows kernel needs to access to perform thread scheduling and synchronization on behalf of running threads. Its layout is illustrated in Figure 6-8.

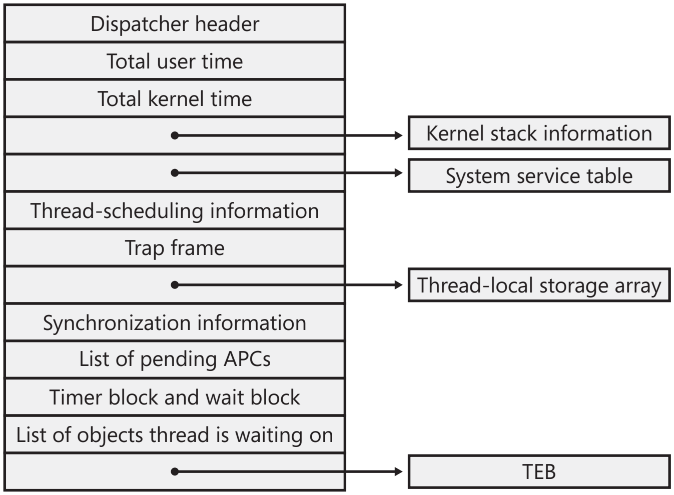

Dispatcher header
Total user time
Total kernel time
Kernel stack information
System service table
Thread-scheduling information
Trap frame
Thread-local storage array
Synchronization information
List of pending APCs
Timer block and wait block
List of objects thread is waiting on
TEB

Figure 6-8 Structure of the kernel thread block

The key fields of the KTHREAD block are described briefly in Table 6-9.

Table 6-9 Key Contents of the KTHREAD Block

Element	Description	Additional Reference
Dispatcher header	Because the thread is an object that can be waited on, it starts with a standard kernel dispatcher object header.	Kernel Dispatcher objects (Chapter 3)
Execution time	Total user and kernel CPU time.	
Pointer to kernel stack information	Base and upper address of the kernel stack.	Memory management (Chapter 7)
Pointer to system service table	Each thread starts out with this field service table pointing to the main system service table (*KeServiceDescriptorTable*). When a thread first calls a Windows GUI service, its system service table is changed to one that includes the GDI and USER services in Win32k.sys.	System Service Dispatching (Chapter 3)
Scheduling information	Base and current priority, quantum, affinity mask, ideal processor, scheduling state, freeze count, and suspend count.	Thread Scheduling
Wait blocks	The thread block contains four built-in wait blocks so that wait blocks don't have to be allocated and initialized each time the thread waits for something. (One wait block is dedicated to timers.)	Synchronization (Chapter 3)

Table 6-9 Key Contents of the KTHREAD Block

Element	Description	Additional Reference
Wait information	List of objects the thread is waiting for, wait reason, and time at which the thread entered the wait state.	Synchronization (Chapter 3)
Mutant list	List of mutant objects the thread owns.	Synchronization (Chapter 3)
APC queues	List of pending user-mode and kernel-mode APCs, and alertable flag.	Aynchronous Procedure Call (APC) Interrrupts (Chapter 3)
Timer block	Built-in timer block (also a corresponding wait block).	
Queue list	Pointer to queue object that the thread is associated with.	Synchronization (Chapter 3)
Pointer to TEB	Thread ID, TLS information, PEB pointer, and GDI and OpenGL information.	

EXPERIMENT: Displaying ETHREAD and KTHREAD Structures

The ETHREAD and KTHREAD structures can be displayed with the *dt* command in the kernel debugger. The following output shows the format of an ETHREAD:

```
lkd> dt nt!_ethread
nt!_ETHREAD
   +0x000 Tcb              : _KTHREAD
   +0x1c0 CreateTime       : _LARGE_INTEGER
   +0x1c0 NestedFaultCount : Pos 0, 2 Bits
   +0x1c0 ApcNeeded        : Pos 2, 1 Bit
   +0x1c8 ExitTime         : _LARGE_INTEGER
   +0x1c8 LpcReplyChain    : _LIST_ENTRY
   +0x1c8 KeyedWaitChain   : _LIST_ENTRY
   +0x1d0 ExitStatus       : Int4B
   +0x1d0 OfsChain         : Ptr32 Void
   +0x1d4 PostBlockList    : _LIST_ENTRY
   +0x1dc TerminationPort  : Ptr32 _TERMINATION_PORT
   +0x1dc ReaperLink       : Ptr32 _ETHREAD
   +0x1dc KeyedWaitValue   : Ptr32 Void
   +0x1e0 ActiveTimerListLock : Uint4B
   +0x1e4 ActiveTimerListHead : _LIST_ENTRY
   +0x1ec Cid              : _CLIENT_ID
   +0x1f4 LpcReplySemaphore : _KSEMAPHORE
   +0x1f4 KeyedWaitSemaphore : _KSEMAPHORE
   +0x208 LpcReplyMessage  : Ptr32 Void
   +0x208 LpcWaitingOnPort : Ptr32 Void
   +0x20c ImpersonationInfo : Ptr32 _PS_IMPERSONATION_INFORMATION
   +0x210 IrpList          : _LIST_ENTRY
   +0x218 TopLevelIrp      : Uint4B
   +0x21c DeviceToVerify   : Ptr32 _DEVICE_OBJECT
   +0x220 ThreadsProcess   : Ptr32 _EPROCESS
   +0x224 StartAddress     : Ptr32 Void
   +0x228 Win32StartAddress : Ptr32 Void
   +0x228 LpcReceivedMessageId : Uint4B
   +0x22c ThreadListEntry  : _LIST_ENTRY
   +0x234 RundownProtect   : _EX_RUNDOWN_REF
   +0x238 ThreadLock       : _EX_PUSH_LOCK
   +0x23c LpcReplyMessageId : Uint4B
   +0x240 ReadClusterSize  : Uint4B
   +0x244 GrantedAccess    : Uint4B
   +0x248 CrossThreadFlags : Uint4B
   +0x248 Terminated       : Pos 0, 1 Bit
   +0x248 DeadThread       : Pos 1, 1 Bit
   +0x248 HideFromDebugger : Pos 2, 1 Bit
   +0x248 ActiveImpersonationInfo : Pos 3, 1 Bit
   +0x248 SystemThread     : Pos 4, 1 Bit
   +0x248 HardErrorsAreDisabled : Pos 5, 1 Bit
   +0x248 BreakOnTermination : Pos 6, 1 Bit
   +0x248 SkipCreationMsg  : Pos 7, 1 Bit
   +0x248 SkipTerminationMsg : Pos 8, 1 Bit
   +0x24c SameThreadPassiveFlags : Uint4B
   +0x24c ActiveExWorker   : Pos 0, 1 Bit
   +0x24c ExWorkerCanWaitUser : Pos 1, 1 Bit
   +0x24c MemoryMaker      : Pos 2, 1 Bit
   +0x250 SameThreadApcFlags : Uint4B
   +0x250 LpcReceivedMsgIdValid : Pos 0, 1 Bit
   +0x250 LpcExitThreadCalled : Pos 1, 1 Bit
   +0x250 AddressSpaceOwner : Pos 2, 1 Bit
   +0x254 ForwardClusterOnly : UChar
   +0x255 DisablePageFaultClustering : UChar
```

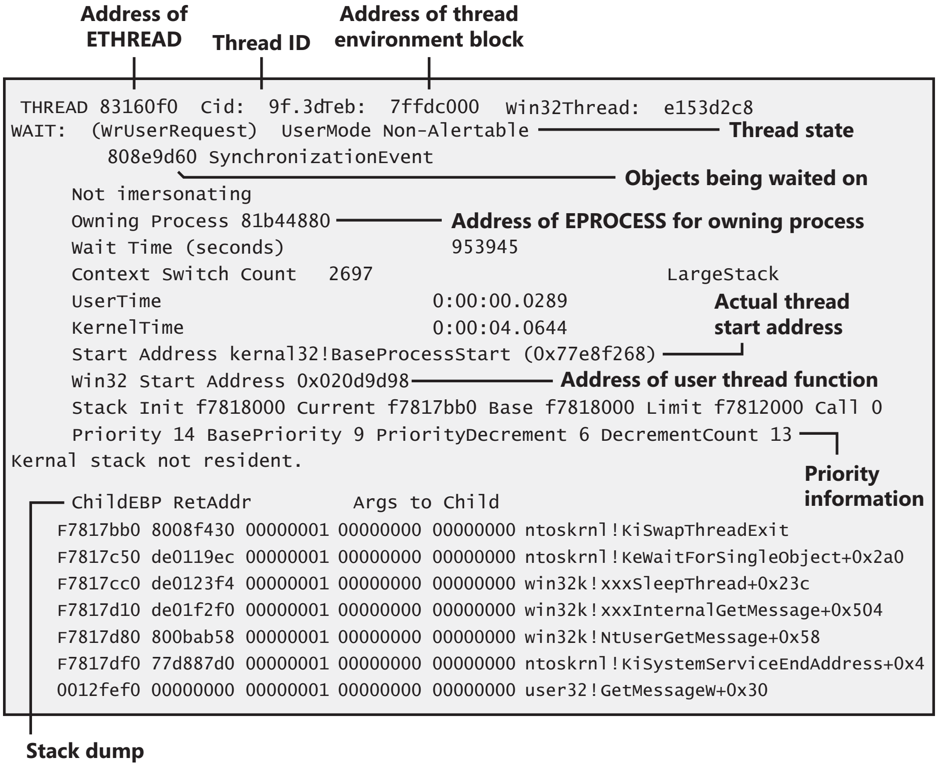
The KTHREAD can be displayed with a similar command:

```
lkd> dt nt!_kthread
nt!_KTHREAD
   +0x000 Header           : _DISPATCHER_HEADER
   +0x010 MutantListHead   : _LIST_ENTRY
   +0x018 InitialStack     : Ptr32 Void
   +0x01c StackLimit       : Ptr32 Void
   +0x020 Teb              : Ptr32 Void
   +0x024 TlsArray         : Ptr32 Void
   +0x028 KernelStack      : Ptr32 Void
   +0x02c DebugActive      : UChar
   +0x02d State            : UChar
   +0x02e Alerted          : [2] UChar
   +0x030 Iopl             : UChar
   +0x031 NpxState         : UChar
   +0x032 Saturation       : Char
   +0x033 Priority         : Char
   +0x034 ApcState         : _KAPC_STATE
   +0x04c ContextSwitches  : Uint4B
   +0x050 IdleSwapBlock    : UChar
   +0x051 Spare0           : [3] UChar
   +0x054 WaitStatus       : Int4B
```

EXPERIMENT: Using the Kernel Debugger *!thread* Command

The kernel debugger *!thread* command dumps a subset of the information in the thread data structures. Some key elements of the information the kernel debugger displays can't be displayed by any utility: internal structure addresses; priority details; stack information; the pending I/O request list; and, for threads in a wait state, the list of objects the thread is waiting for.

To display thread information, use either the *!process* command (which displays all the thread blocks after displaying the process block) or the *!thread* command to dump a specific thread. The output of the thread information, along with some annotations of key fields, is shown here:



```
 THREAD 83160f0  Cid:  9f.3dTeb:  7ffdc000  Win32Thread:  e153d2c8
WAIT:  (WrUserRequest)  UserMode Non-Alertable
        808e9d60 SynchronizationEvent
     Not imersonating
     Owning Process 81b44880
     Wait Time (seconds)                953945
     Context Switch Count   2697                             LargeStack
     UserTime                         0:00:00.0289
     KernelTime                       0:00:04.0644
     Start Address kernal32!BaseProcessStart (0x77e8f268)
     Win32 Start Address 0x020d9d98
     Stack Init f7818000 Current f7817bb0 Base f7818000 Limit f7812000 Call 0
     Priority 14 BasePriority 9 PriorityDecrement 6 DecrementCount 13
Kernal stack not resident.

     ChildEBP RetAddr         Args to Child
    F7817bb0 8008f430 00000001 00000000 00000000 ntoskrnl!KiSwapThreadExit
    F7817c50 de0119ec 00000001 00000000 00000000 ntoskrnl!KeWaitForSingleObject+0x2a0
    F7817cc0 de0123f4 00000001 00000000 00000000 win32k!xxxSleepThread+0x23c
    F7817d10 de01f2f0 00000001 00000000 00000000 win32k!xxxInternalGetMessage+0x504
    F7817d80 800bab58 00000001 00000000 00000000 win32k!NtUserGetMessage+0x58
    F7817df0 77d887d0 00000001 00000000 00000000 ntoskrnl!KiSystemServiceEndAddress+0x4
    0012fef0 00000000 00000001 00000000 00000000 user32!GetMessageW+0x30
```

Address of ETHREAD
Thread ID
Address of thread environment block
Thread state
Objects being waited on
Address of EPROCESS for owning process
Actual thread start address
Address of user thread function
Priority information
Stack dump

EXPERIMENT: Viewing Thread Information

The following output is the detailed display of a process produced by using the Tlist utility in the Windows Debugging Tools. Notice that the thread list shows the "Win32StartAddress." This is the address passed to the *CreateThread* function by the application. All the other utilities, except Process Explorer, that show the thread start address show the actual start address (a function in Kernel32.dll), not the application-specified start address.

```
C:\> tlist winword
 155 WINWORD.EXE        Document1 - Microsoft Word
   CWD:     C:\book\
   CmdLine: "C:\Program Files\Microsoft Office\Office\WINWORD.EXE"
   VirtualSize:    64448 KB   PeakVirtualSize:   106748 KB
   WorkingSetSize:  1104 KB   PeakWorkingSetSize:  6776 KB
   NumberOfThreads: 2
    156 Win32StartAddr:0x5032cfdb LastErr:0x00000000 State:Waiting
    167 Win32StartAddr:0x00022982 LastErr:0x00000000 State:Waiting
                     0x50000000  WINWORD.EXE
     5.0.2163.1 shp  0x77f60000  ntdll.dll
     5.0.2191.1 shp  0x77f00000  KERNEL32.dll
         §          list of DLLs loaded in process
```

The TEB, illustrated in Figure 6-9, is the only data structure explained in this section that exists in the process address space (as opposed to the system space).

The TEB stores context information for the image loader and various Windows DLLs. Because these components run in user mode, they need a data structure writable from user mode. That's why this structure exists in the process address space instead of in the system space, where it would be writable only from kernel mode. You can find the address of the TEB with the kernel debugger *!thread* command.

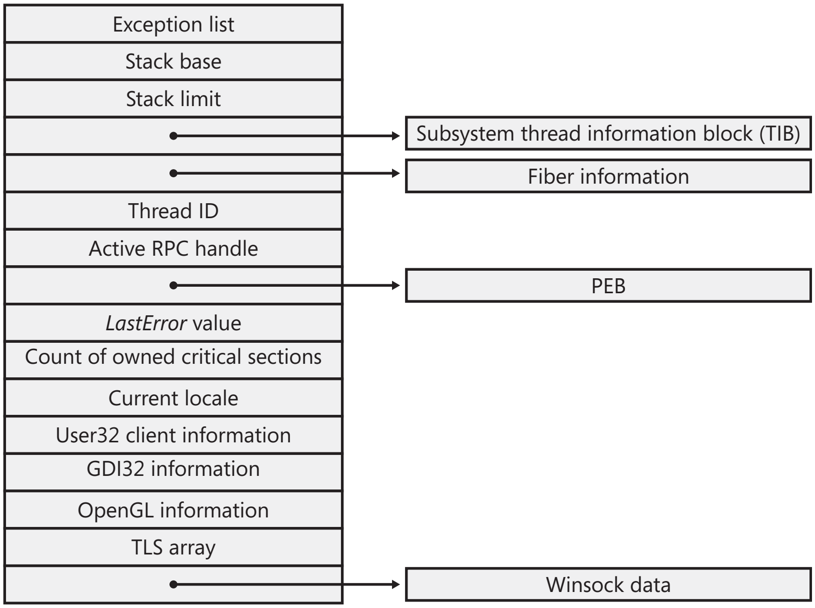


Figure 6-9 Fields of the thread environment block

EXPERIMENT: Examining the TEB

You can dump the TEB structure with the *!teb* command in the kernel debugger. The output looks like this:

```
kd> !teb
TEB at 7ffde000
    ExceptionList:        0006b540
    StackBase:            00070000
    StackLimit:           00065000
    SubSystemTib:         00000000
    FiberData:            00001e00
    ArbitraryUserPointer: 00000000
    Self:                 7ffde000
    EnvironmentPointer:   00000000
    ClientId:             00000254 . 000007ac
    RpcHandle:            00000000
    Tls Storage:          00000000
    PEB Address:          7ffdf000
    LastErrorValue:       2
    LastStatusValue:      c0000034
    Count Owned Locks:    0     HardErrorMode:        0
```

Kernel Variables

As with processes, a number of Windows kernel variables control how threads run. Table 6-10 shows the kernel-mode kernel variables that relate to threads.

Table 6-10 Thread-Related Kernel Variables

Variable	Type	Description
PspCreateThreadNotifyRoutine	Array of pointers	Array of pointers to routines to be called on during thread creation and deletion (maximum of eight).
PspCreateThreadNotifyRoutineCount	DWORD	Count of registered thread-notification routines.
PspCreateProcessNotifyRoutine	Array of pointers	Array of pointers to routines to be called on during process creation and deletion (maximum of eight).

Performance Counters

Most of the key information in the thread data structures is exported as performance counters, which are listed in Table 6-11. You can extract much information about the internals of a thread just by using the Performance tool in Windows.

Table 6-11 Thread-Related Performance Counters

Object: Counter	Function
Process: Priority Base	Returns the current base priority of the process. This is the starting priority for threads created within this process.
Thread: % Privileged Time	Describes the percentage of time that the thread has run in kernel mode during a specified interval.
Thread: % Processor Time	Describes the percentage of CPU time that the thread has used during a specified interval. This count is the sum of % Privileged Time and % User Time.
Thread: % User Time	Describes the percentage of time that the thread has run in user mode during a specified interval.
Thread: Context Switches/Sec	Returns the number of context switches per second that the system is executing.
Thread: Elapsed Time	Returns the amount of CPU time (in seconds) that the thread has consumed.
Thread: ID Process	Returns the process ID of the thread's process. This ID is valid only during the process's lifetime because process IDs are re-used.
Thread: ID Thread	Returns the thread's thread ID. This ID is valid only during the thread's lifetime because thread IDs are reused.
Thread: Priority Base	Returns the thread's current base priority. This number might be different from the thread's starting base priority.
Thread: Priority Current	Returns the thread's current dynamic priority.
Thread: Start Address	Returns the thread's starting virtual address (*Note*: This address will be the same for most threads.)
Thread: Thread State	Returns a value from 0 through 7 relating to the current state of the thread.
Thread: Thread Wait Reason	Returns a value from 0 through 19 relating to the reason why the thread is in a wait state.

Relevant Functions

Table 6-12 shows the Windows functions for creating and manipulating threads. This table doesn't include functions that have to do with thread scheduling and priorities—those are included in the section "Thread Scheduling" later in this chapter.

Table 6-12 **Windows Thread Functions**

Function	Description
CreateThread	Creates a new thread
CreateRemoteThread	Creates a thread in another process
OpenThread	Opens an existing thread
ExitThread	Ends execution of a thread normally
TerminateThread	Terminates a thread
GetExitCodeThread	Gets another thread's exit code
GetThreadTimes	Returns timing information for a thread
GetCurrentProcess	Returns a pseudo handle for the current thread
GetCurrentProcessId	Returns the thread ID of the current thread
GetThreadId	Returns the thread ID of the specified thread
Get/SetThreadContext	Returns or changes a thread's CPU registers
GetThreadSelectorEntry	Returns another thread's descriptor table entry (applies only to x86 systems)

Birth of a Thread

A thread's life cycle starts when a program creates a new thread. The request filters down to the Windows executive, where the process manager allocates space for a thread object and calls the kernel to initialize the kernel thread block. The steps in the following list are taken inside the Windows *CreateThread* function in Kernel32.dll to create a Windows thread.

1. *CreateThread* creates a user-mode stack for the thread in the process's address space.
2. *CreateThread* initializes the thread's hardware context (CPU architecture–specific). (For further information on the thread context block, see the Windows API reference documentation on the CONTEXT structure.)
3. *NtCreateThread* is called to create the executive thread object in the suspended state. For a description of the steps performed by this function, see the description of Stage 3 and Stage 6 in the section "Flow of *CreateProcess*."
4. *CreateThread* notifies the Windows subsystem about the new thread, and the subsystem does some setup work for the new thread.
5. The thread handle and the thread ID (generated during step 3) are returned to the caller.

6. Unless the caller created the thread with the CREATE_SUSPENDED flag set, the thread is now resumed so that it can be scheduled for execution. When the thread starts running, it executes the steps described in the earlier section "Stage 6: Performing Process Initialization in the Context of the New Process" before calling the actual user's specified start address.

Examining Thread Activity

Besides the Performance tool, several other tools expose various elements of the state of Windows threads. (The tools that show thread-scheduling information are listed in the section "Thread Scheduling.") These tools are itemized in Figure 6-10.

Note To display thread details with Tlist, you must type **tlist xxx**, where *xxx* is a process image name or window title. (Wildcards are supported.)

Object	Perfmon	Pviewer	Pstat	Qslice	Tlist	KD *!thread*	Process Explorer	Pslist
Thread ID	✓	✓	✓		✓	✓	✓	✓
Actual start address	✓	✓	✓			✓	✓	✓
Win32 start address					✓	✓	✓	
Current address	✓	✓				✓	✓	
Number of context switches	✓	✓	✓				✓	✓
Total user time		✓	✓			✓	✓	✓
Total privileged time		✓	✓			✓	✓	✓
Elapsed time	✓	✓				✓	✓	✓
Thread state	✓		✓		✓	✓	✓	✓
Reason for wait state	✓		✓		✓	✓	✓	✓
Last error					✓		✓	
Percentage of CPU time	✓			✓			✓	
Percentage of user time	✓			✓			✓	
Percentage of privileged time	✓			✓			✓	
Address of TEB						✓		
Address of ETHREAD						✓		
Objects waiting on						✓		

Figure 6-10 Thread-related tools and their functions

Process Explorer provides easy access to thread activity within a process. This is especially important if you are trying to determine why a process is running that is hosting multiple services (such as Svchost.exe, Dllhost.exe, Inetinfo.exe, or the System process) or why a process is hung.

To view the threads in a process, select a process and open the process properties (double-click on the process or click on the Process, Properties menu item). Then click on the Threads tab. This tab shows a list of the threads in the process. For each thread it shows the percentage of CPU consumed (based on the refresh interval configured), the number of context switches to the thread, and the thread start address. You can sort by any of these three columns.

New threads that are created are highlighted in green, and threads that exit are highlighted in red. (The highlight duration can be configured with the Options, Configure Highlighting menu item.) This might be helpful to discover unnecessary thread creation occurring in a process. (In general, threads should be created at process startup, not every time a request is processed inside a process.)

As you select each thread in the list, Process Explorer displays the thread ID, start time, state, CPU time counters, number of context switches, and the base and current priority. There is a Kill button, which will terminate an individual thread, but this should be used with extreme care.

The context switch delta represents the number of times that thread began running in between the refreshes configured for Process Explorer. It provides a different way to determine thread activity than using the percentage of CPU consumed. In some ways it is better because many threads run for such a short amount of time that they are seldom (if ever) the currently running thread when the interval clock timer interrupt occurs, and therefore, are not charged for their CPU time. For example, if you add the context switch delta column to the process display and sort by that column, you will see processes that have threads running but that also have a CPU time percentage of zero (or very small).

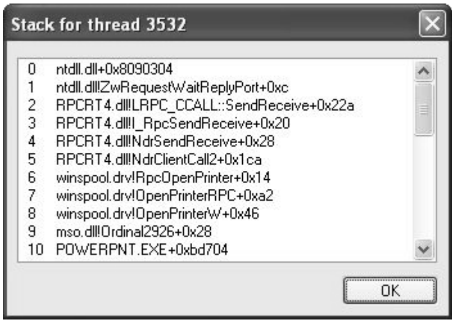
The thread start address is displayed in the form "*module!function*", where *module* is the name of the .exe or .dll. The function name relies on access to symbol files for the module. (See "Experiment: Viewing Process Details with Process Explorer" in Chapter 1.) If you are unsure what the module is, press the Module button. This opens an Explorer file properties window for the module containing the thread's start address (for example, the .exe or .dll).

> **Note** For threads created by the Windows CreateThread function, Process Explorer displays the function passed to *CreateThread*, not the actual thread start function. That is because all Windows threads start at a common process or thread startup wrapper function (*BaseProcessStart* or *BaseThreadStart* in Kernel32.dll). If Process Explorer showed the actual start address, most threads in processes would appear to have started at the same address, which would not be helpful in trying to understand what code the thread was executing.

However, the thread start address displayed might not be enough information to pinpoint what the thread is doing and which component within the process is responsible for the CPU consumed by the thread. This is especially true if the thread start address is a generic startup function (for example, if the function name does not indicate what the thread is actually doing). In this case, examining the thread stack might answer the question. To view the stack for a thread, double-click on the thread of interest (or select it and click the Stack button). Process Explorer displays the thread's stack (both user and kernel, if the thread was in kernel mode).

> **Note** While the user-mode debuggers (Windbg, Ntsd, and Cdb) permit you to attach to a process and display the user stack for a thread, Process Explorer shows both the user and kernel stack in one easy click of a button. You can also examine user and kernel thread stacks using Livekd from *www.sysinternals.com*. However, it is more difficult to use. Note that running Windbg in local kernel debugging mode, which is supported only on Windows XP or Windows Server 2003, does not show thread stacks.

Viewing the thread stack can also help you determine why a process is hung. As an example, on one system, Microsoft PowerPoint was hanging for one minute on startup. To determine why it was hung, after starting PowerPoint, Process Explorer was used to examine the thread stack of the one thread in the process. The result is shown in Figure 6-11.



```
0    ntdll.dll+0x8090304
1    ntdll.dll!ZwRequestWaitReplyPort+0xc
2    RPCRT4.dll!LRPC_CCALL::SendReceive+0x22a
3    RPCRT4.dll!I_RpcSendReceive+0x20
4    RPCRT4.dll!NdrSendReceive+0x28
5    RPCRT4.dll!NdrClientCall2+0x1ca
6    winspool.drv!RpcOpenPrinter+0x14
7    winspool.drv!OpenPrinterRPC+0xa2
8    winspool.drv!OpenPrinterW+0x46
9    mso.dll!Ordinal2926+0x28
10   POWERPNT.EXE+0xbd704
```

Figure 6-11 Hung Thread Stack in PowerPoint

This thread stack shows that PowerPoint (line 10) called a function in Mso.dll (the central Microsoft Office Dll), which called the *OpenPrinterW* function in Winspool.drv (a Dll used to connect to printers). Winspool.drv then dispatched to a function *OpenPrinterRPC*, which then called a function in the RPC runtime Dll, indicating it was sending the request to a remote printer. So, without having to understand the internals of PowerPoint, the module and function names displayed on the thread stack indicate that the thread was waiting to connect to a network printer. On this particular system, there was a network printer that was not responding, which explained the delay starting PowerPoint. (Microsoft Office applications connect to all configured printers at process startup.) The connection to that printer was deleted from the user's system, and the problem went away.

Thread Scheduling

This section describes the Windows scheduling policies and algorithms. The first subsection provides a condensed description of how scheduling works on Windows and a definition of key terms. Then Windows priority levels are described from both the Windows API and the Windows kernel points of view. After a review of the relevant Windows functions and

_Windows utilities and tools that relate to scheduling, the detailed data structures and algorithms that make up the Windows scheduling system are presented, with uniprocessor systems examined first and then multiprocessor systems.

Overview of Windows Scheduling

Windows implements a priority-driven, preemptive scheduling system—the highest-priority runnable (*ready*) thread always runs, with the caveat that the thread chosen to run might be limited by the processors on which the thread is allowed to run, a phenomenon called *processor affinity*. By default, threads can run on any available processor, but you can alter processor affinity by using one of the Windows scheduling functions listed in Table 6-13 (shown later in the chapter) or by setting an affinity mask in the image header.

EXPERIMENT: Viewing Ready Threads

You can view the list of ready threads with the kernel debugger *!ready* command. This command displays the thread or list of threads that are ready to run at each priority level. In the following example, two threads are ready to run at priority 10 and six at priority 8. Because this output was generated using LiveKd on a uniprocessor system, the current thread will always be the kernel debugger (Kd or WinDbg).

```
kd> !ready 1
Ready Threads at priority 10
 THREAD 810de030  Cid 490.4a8 Teb: 7ffd9000  Win32Thread: e297e008 READY
 THREAD 81110030  Cid 490.48c Teb: 7ffde000  Win32Thread: e29425a8 READY
Ready Threads at priority 8
 THREAD 811fe790  Cid 23c.274 Teb: 7ffdb000  Win32Thread: e258cda8 READY
 THREAD 810bec70  Cid 23c.50c Teb: 7ffd9000  Win32Thread: e2ccf748 READY
 THREAD 8003a950  Cid 23c.550 Teb: 7ffda000  Win32Thread: e29a7ae8 READY
 THREAD 85ac2db0  Cid 23c.5e4 Teb: 7ffd8000  Win32Thread: e297a9e8 READY
 THREAD 827318d0  Cid 514.560 Teb: 7ffd9000  Win32Thread: 00000000 READY
 THREAD 8117adb0  Cid 2d4.338 Teb: 7ffaf000  Win32Thread: 00000000 READY
```

When a thread is selected to run, it runs for an amount of time called a *quantum*. A quantum is the length of time a thread is allowed to run before another thread at the same priority level (or higher, which can occur on a multiprocessor system) is given a turn to run. Quantum values can vary from system to system and process to process for any of three reasons: system configuration settings (long or short quantums), foreground/background status of the process, or use of the job object to alter the quantum. (Quantums are described in more detail in the "Quantum" section later in the chapter.) A thread might not get to complete its quantum, however. Because Windows implements a preemptive scheduler, if another thread with a higher priority becomes ready to run, the currently running thread might be preempted before finishing its time slice. In fact, a thread can be selected to run next and be preempted before even beginning its quantum!

The Windows scheduling code is implemented in the kernel. There's no single "scheduler" module or routine, however—the code is spread throughout the kernel in which scheduling-related events occur. The routines that perform these duties are collectively called the kernel's *dispatcher*. The following events might require thread dispatching:

- A thread becomes ready to execute—for example, a thread has been newly created or has just been released from the wait state.
- A thread leaves the running state because its time quantum ends, it terminates, it yields execution, or it enters a wait state.
- A thread's priority changes, either because of a system service call or because Windows itself changes the priority value.
- A thread's processor affinity changes so that it will no longer run on the processor on which it was running.

At each of these junctions, Windows must determine which thread should run next. When Windows selects a new thread to run, it performs a *context switch* to it. A context switch is the procedure of saving the volatile machine state associated with a running thread, loading another thread's volatile state, and starting the new thread's execution.

As already noted, Windows schedules at the thread granularity. This approach makes sense when you consider that processes don't run but only provide resources and a context in which their threads run. Because scheduling decisions are made strictly on a thread basis, no consideration is given to what process the thread belongs to. For example, if process *A* has 10 runnable threads, process *B* has 2 runnable threads, and all 12 threads are at the same priority, each thread would theoretically receive one-twelfth of the CPU time—Windows wouldn't give 50 percent of the CPU to process *A* and 50 percent to process *B*.

Priority Levels

To understand the thread-scheduling algorithms, you must first understand the priority levels that Windows uses. As illustrated in Figure 6-12, internally, Windows uses 32 priority levels, ranging from 0 through 31. These values divide up as follows:

- Sixteen real-time levels (16 through 31)
- Fifteen variable levels (1 through 15)
- One system level (0), reserved for the zero page thread

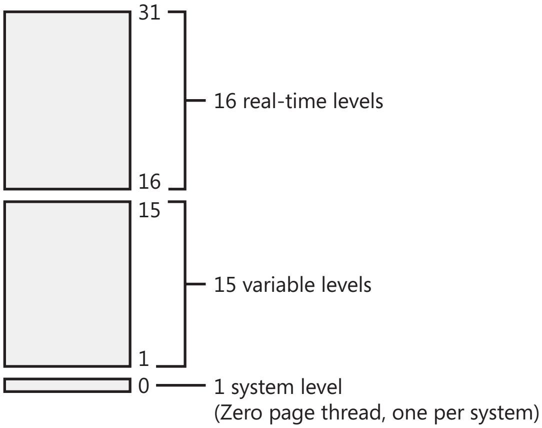


Figure 6-12 Thread priority levels

Thread priority levels are assigned from two different perspectives: those of the Windows API and those of the Windows kernel. The Windows API first organizes processes by the priority class to which they are assigned at creation (Real-time, High, Above Normal, Normal, Below Normal, and Idle) and then by the relative priority of the individual threads within those processes (Time-critical, Highest, Above-normal, Normal, Below-normal, Lowest, and Idle).

In the Windows API, each thread has a base priority that is a function of its process priority class and its relative thread priority. The mapping from Windows priority to internal Windows numeric priority is shown in Figure 6-13.

Whereas a process has only a single base priority value, each thread has two priority values: current and base. Scheduling decisions are made based on the current priority. As explained in the following section on priority boosting, the system under certain circumstances increases the priority of threads in the dynamic range (1 through 15) for brief periods. Windows never adjusts the priority of threads in the real-time range (16 through 31), so they always have the same base and current priority.

A thread's initial base priority is inherited from the process base priority. A process, by default, inherits its base priority from the process that created it. This behavior can be overridden on the *CreateProcess* function or by using the command-line START command. A process priority can also be changed after being created by using the *SetPriorityClass* function or various tools that expose that function such as Task Manager and Process Explorer (by right-clicking on the process and choosing a new priority class). For example, you can lower the priority of a CPU-intensive process so that it does not interfere with normal system activities. Changing the priority of a process changes the thread priorities up or down, but their relative settings remain the same. It usually doesn't make sense, however, to change individual thread priorities within a process, because unless you wrote the program or have the source code, you don't really know what the individual threads are doing, and changing their relative importance might cause the program not to behave in the intended fashion.

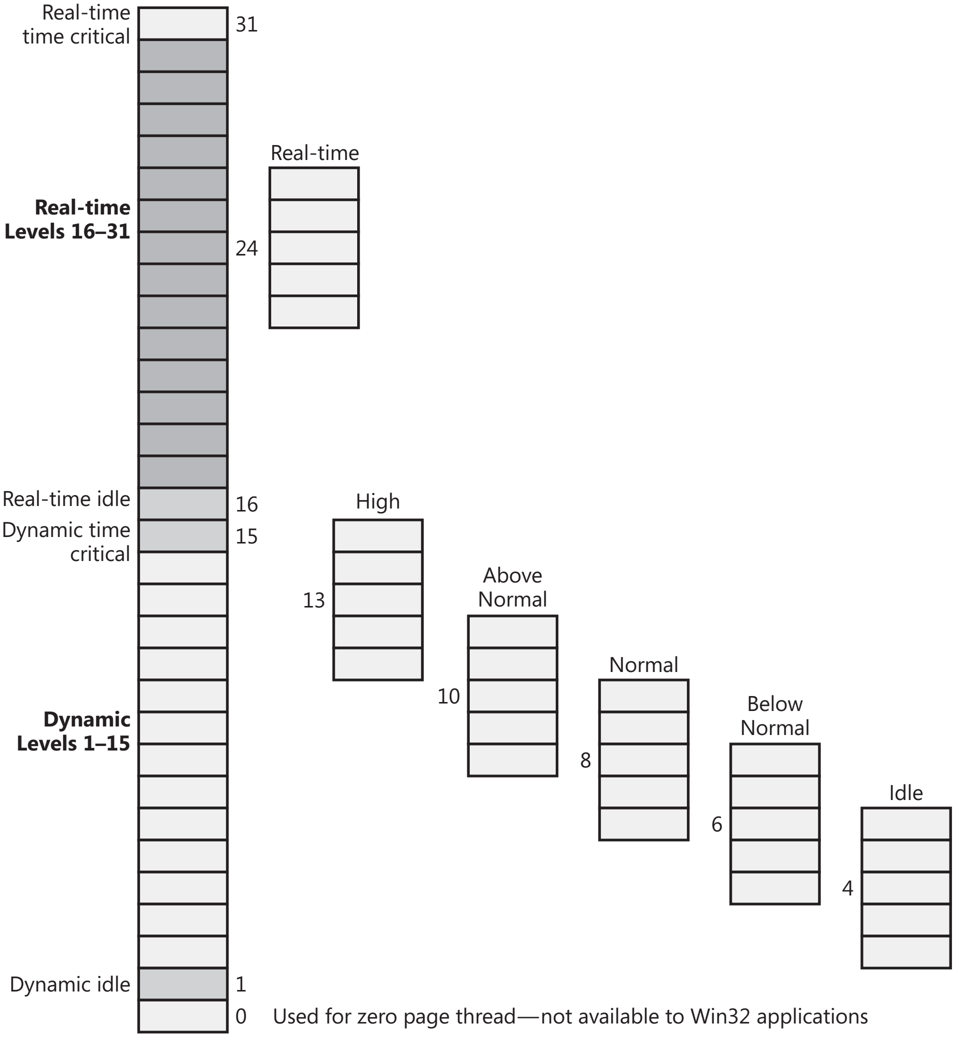


Figure 6-13 Mapping of Windows kernel priorities to the Windows API

Normally, the process base priority (and therefore the starting thread base priority) will default to the value at the middle of each process priority range (24, 13, 10, 8, 6, or 4). However, some Windows system processes (such as the Session Manager, service controller, and local security authentication server) have a base process priority slightly higher than the default for the Normal class (8). This higher default value ensures that the threads in these processes will all start at a higher priority than the default value of 8. These system processes use an internal system call (*NtSetInformationProcess*) to set its process base priority to a numeric value other than the normal default starting base priority.

Windows Scheduling APIs

The Windows API functions that relate to thread scheduling are listed in Table 6-13. (For more information, see the Windows API reference documentation.)

Table 6-13 Scheduling-Related APIs and Their Functions

API	Function
Suspend/ResumeThread	Suspends or resumes a paused thread from execution.
Get/SetPriorityClass	Returns or sets a process's priority class (base priority).
Get/SetThreadPriority	Returns or sets a thread's priority (relative to its process base priority).
Get/SetProcessAffinityMask	Returns or sets a process's affinity mask.
SetThreadAffinityMask	Sets a thread's affinity mask (must be a subset of the process's affinity mask) for a particular set of processors, restricting it to running on those processors.
SetInformationJobObject	Sets attributes for a job; some of the attributes affect scheduling, such as affinity and priority. (See the "Job Objects" section later in the chapter for a description of the job object.)
GetLogicalProcessorInformation	Returns details about processor hardware configuration (for hyperthreaded and NUMA systems).
Get/SetThreadPriorityBoost	Returns or sets the ability for Windows to boost the priority of a thread temporarily. (This ability applies only to threads in the dynamic range.)
SetThreadIdealProcessor	Establishes a preferred processor for a particular thread, but doesn't restrict the thread to that processor.
Get/SetProcessPriorityBoost	Returns or sets the default priority boost control state of the current process. (This function is used to set the thread priority boost control state when a thread is created.)
SwitchToThread	Yields execution to another thread (at priority 1 or higher) that is ready to run on the current processor.
Sleep	Puts the current thread into a wait state for a specified time interval (figured in milliseconds [msec]). A zero value relinquishes the rest of the thread's quantum.
SleepEx	Causes the current thread to go into a wait state until either an I/O completion callback is completed, an APC is queued to the thread, or the specified time interval ends.

Relevant Tools

The following table lists the tools related to thread scheduling. You can change (and view) the base process priority with a number of different tools, such as Task Manager, Process Explorer, Pview, or Pviewer. Note that you can kill individual threads in a process with Process Explorer. This should be done, of course, with extreme care.

You can view individual thread priorities with the Performance tool, Process Explorer, Pslist, Pview, Pviewer, and Pstat. While it might be useful to increase or lower the priority of a process, it typically does not make sense to adjust individual thread priorities within a process because only a person who thoroughly understands the program would understand the relative importance of the threads within the process.

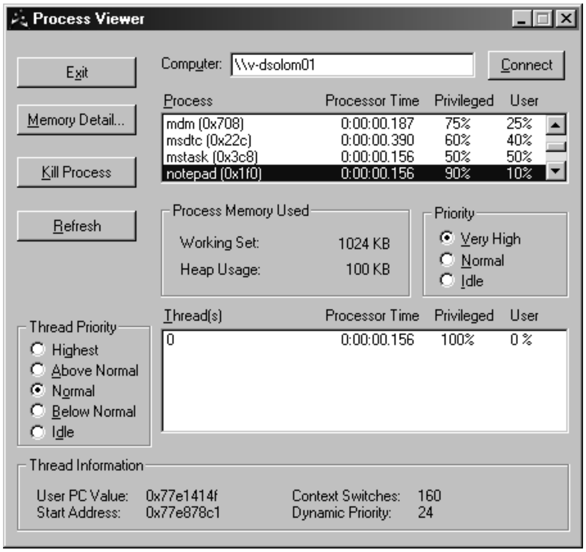
						KD	Process
Object	**Taskman**	**Perfmon**	**Pviewer**	**Pview**	**Pstat**	***!thread***	**Explorer**
Process priority class	✓		✓	✓			✓
Process base priority		✓			✓		✓
Thread base priority		✓					✓
Thread current priority		✓	✓	✓	✓	✓	✓

The only way to specify a starting priority class for a process is with the *start* command in the Windows command prompt. If you want to have a program start every time with a specific priority, you can define a shortcut to use the start command by beginning the command with **cmd /c**. This runs the command prompt, executes the command on the command line, and terminates the command prompt. For example, to run Notepad in the low-process priority, the shortcut would be **cmd /c start /low notepad.exe**.

EXPERIMENT: Examining and Specifying Process and Thread Priorities

Try the following experiment:

1. From the command prompt, type **start /realtime notepad**. Notepad should open.
2. Run either Process Explorer or the Process Viewer utility in the Support Tools (Pviewer.exe), and select Notepad.exe from the list of processes, as shown here. Notice that the dynamic priority of the thread in Notepad is 24. This matches the real-time value shown in Figure 6-13.



Computer: \\v-dsolom01

Process	Processor Time	Privileged	User
mdm (0x708)	0:00:00.187	75%	25%
msdtc (0x22c)	0:00:00.390	60%	40%
mstask (0x3c8)	0:00:00.156	50%	50%
notepad (0x1f0)	0:00:00.156	90%	10%

Process Memory Used

Working Set: 1024 KB

Heap Usage: 100 KB

Priority

● Very High
○ Normal
○ Idle

Thread(s)	Processor Time	Privileged	User
0	0:00:00.156	100%	0 %

Thread Priority

○ Highest
○ Above Normal
● Normal
○ Below Normal
○ Idle

Thread Information

User PC Value: 0x77e1414f   Context Switches: 160
Start Address: 0x77e878c1   Dynamic Priority: 24

3. Task Manager can show you similar information. Press Ctrl+Shift+Esc to start Task Manager, and go to the Processes tab. Right-click on the Notepad.exe process, and select the Set Priority option. You can see that Notepad's process priority class is Realtime, as shown in the following dialog box.

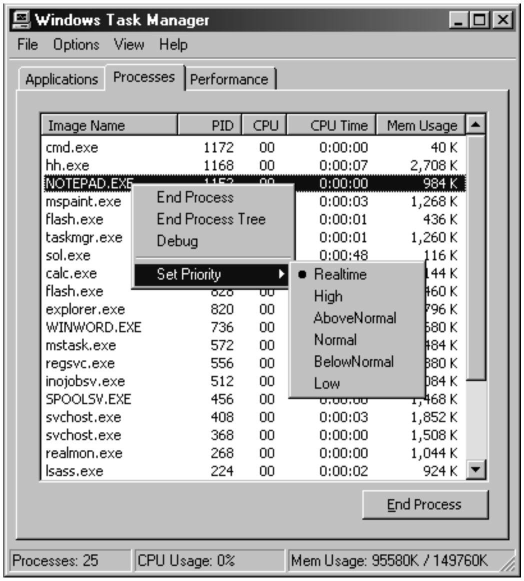


Image Name	PID	CPU	CPU Time	Mem Usage
cmd.exe	1172	00	0:00:00	40 K
hh.exe	1168	00	0:00:07	2,708 K
NOTEPAD.EXE	1152	00	0:00:00	984 K
mspaint.exe			0:00:03	1,268 K
flash.exe			0:00:01	436 K
taskmgr.exe			0:00:01	1,260 K
sol.exe			0:00:48	116 K
calc.exe				144 K
flash.exe	828	00		460 K
explorer.exe	820	00		796 K
WINWORD.EXE	736	00		580 K
mstask.exe	572	00		484 K
regsvc.exe	556	00		380 K
inojobsv.exe	512	00		084 K
SPOOLSV.EXE	456	00	0:00:00	1,468 K
svchost.exe	408	00	0:00:03	1,852 K
svchost.exe	368	00	0:00:00	1,508 K
realmon.exe	268	00	0:00:00	1,044 K
lsass.exe	224	00	0:00:02	924 K

Processes: 25   CPU Usage: 0%   Mem Usage: 95580K / 149760K

Windows System Resource Manager

Windows Server 2003, Enterprise Edition and Windows Server 2003, Datacenter Edition include an optionally installable component called Windows System Resource Manager (WSRM). It permits the administrator to configure policies that specify CPU utilization, affinity settings, and memory limits (both physical and virtual) for processes. In addition, WSRM can generate resource utilization reports that can be used for accounting and verification of service-level agreements with users.

Policies can be applied for specific applications (by matching the name of the image with or without specific command-line arguments), users, or groups. The policies can be scheduled to take effect at certain periods or can be enabled all the time.

After you have set a resource-allocation policy to manage specific processes, the WSRM service monitors CPU consumption of managed processes and adjusts process base priorities when those processes do not meet their target CPU allocations.

The physical memory limitation uses the function *SetProcessWorkingSetSizeEx* to set a hard-working set maximum. The virtual memory limit is implemented by the service checking the private virtual memory consumed by the processes. (See Chapter 7 for an explanation of these memory limits.) If this limit is exceeded, WSRM can be configured to either kill the processes or write an entry to the event log. This behavior could be used to detect a process with a memory leak before it consumes all the available committed virtual memory on the system. Note that WSRM memory limits do not apply to Address Windowing Extensions (AWE) memory, large page memory, or kernel memory (nonpaged or paged pool).

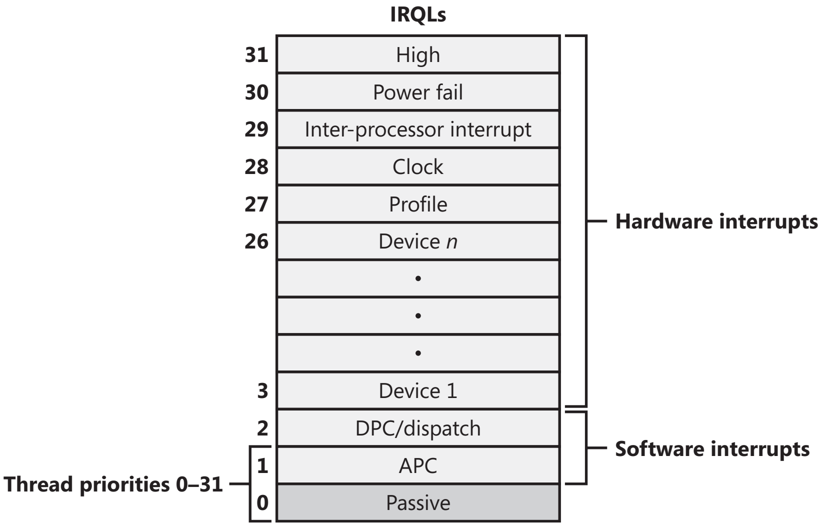
Real-Time Priorities

You can raise or lower thread priorities within the dynamic range in any application; however, you must have the *increase scheduling priority* privilege to enter the real-time range. Be aware that many important Windows kernel-mode system threads run in the real-time priority range, so if threads spend excessive time running in this range, they might block critical system functions (such as in the memory manager, cache manager, or other device drivers).

Note As illustrated in the following figure showing the x86 Interrupt Request Levels (IRQLs), although Windows has a set of priorities called *real-time*, they are not real-time in the common definition of the term. This is because Windows doesn't provide true real-time operating system facilities, such as guaranteed interrupt latency or a way for threads to obtain a guaranteed execution time. For more information, see the sidebar "Windows and Real-Time Processing" in Chapter 3 as well as the MSDN Library article "Real-Time Systems and Microsoft Windows NT."

Interrupt Levels vs. Priority Levels

As illustrated in the following figure, threads normally run at IRQL 0 or 1. (For a description of how Windows uses interrupt levels, see Chapter 3.) User-mode threads always run at IRQL 0. Because of this, no user-mode thread, regardless of its priority, blocks hardware interrupts (although high-priority real-time threads can block the execution of important system threads). Only kernel-mode APCs execute at IRQL 1 because they interrupt the execution of a thread. (For more information on APCs, see Chapter 3.) Threads running in kernel mode can raise IRQL to higher levels, though—for example, while executing a system call that involves thread dispatching.



Thread States

Before you can comprehend the thread-scheduling algorithms, you need to understand the various execution states that a thread can be in. Figure 6-14 illustrates the state transitions for threads on Windows 2000 and Windows XP. (The numeric values shown represent the value of the thread state performance counter.) More details on what happens at each transition are included later in this section.

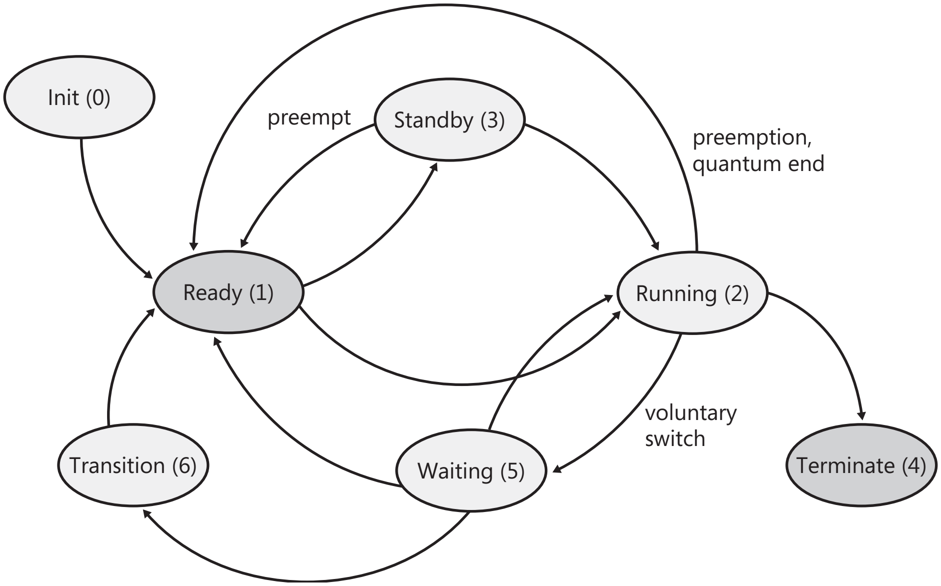


Figure 6-14 Thread states on Windows 2000 and Windows XP

The thread states are as follows:

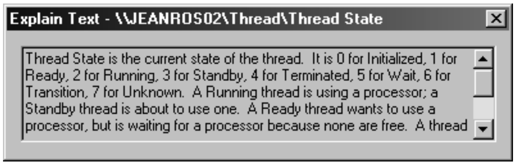
- **Ready** A thread in the ready state is waiting to execute. When looking for a thread to execute, the dispatcher considers only the pool of threads in the ready state.
- **Standby** A thread in the standby state has been selected to run next on a particular processor. When the correct conditions exist, the dispatcher performs a context switch to this thread. Only one thread can be in the standby state for each processor on the system. Note that a thread can be preempted out of the standby state before it ever executes (if, for example, a higher priority thread becomes runnable before the standby thread begins execution).
- **Running** Once the dispatcher performs a context switch to a thread, the thread enters the running state and executes. The thread's execution continues until its quantum ends (and another thread at the same priority is ready to run), it is preempted by a higher priority thread, it terminates, it yields execution, or it voluntarily enters the wait state.
- **Waiting** A thread can enter the wait state in several ways: a thread can voluntarily wait for an object to synchronize its execution, the operating system can wait on the thread's behalf (such as to resolve a paging I/O), or an environment subsystem can direct the thread to suspend itself. When the thread's wait ends, depending on the priority, the thread either begins running immediately or is moved back to the ready state.

- **Transition** A thread enters the transition state if it is ready for execution but its kernel stack is paged out of memory. Once its kernel stack is brought back into memory, the thread enters the ready state.
- **Terminated** When a thread finishes executing, it enters the terminated state. Once the thread is terminated, the executive thread block (the data structure in nonpaged pool that describes the thread) might or might not be deallocated. (The object manager sets policy regarding when to delete the object.)
- **Initialized** This state is used internally while a thread is being created.

EXPERIMENT: Thread-Scheduling State Changes

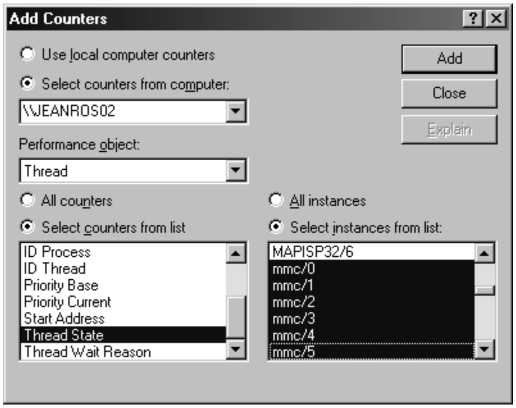
You can watch thread-scheduling state changes with the Performance tool in Windows. This utility can be useful when you're debugging a multithreaded application if you're unsure about the state of the threads running in the process. To watch thread-scheduling state changes by using the Performance tool, follow these steps:

1. Run the Microsoft Notepad utility (Notepad.exe).
2. Start the Performance tool by selecting Programs from the Start menu and then selecting Performance from the Adminstrative Tools menu.
3. Select chart view if you're in some other view.
4. Right-click on the graph, and choose Properties.
5. Click the Graph tab, and change the chart vertical scale maximum to 7. (As you'll see from the explanation text for the performance counter, thread states are numbered from 0 through 7.) Click OK.
6. Click the Add button on the toolbar to bring up the Add Counters dialog box.
7. Select the Thread performance object, and then select the Thread State counter. Click the Explain button to see the definition of the values:



Thread State is the current state of the thread. It is 0 for Initialized, 1 for Ready, 2 for Running, 3 for Standby, 4 for Terminated, 5 for Wait, 6 for Transition, 7 for Unknown. A Running thread is using a processor; a Standby thread is about to use one. A Ready thread wants to use a processor, but is waiting for a processor because none are free. A thread

8. In the Instances box, scroll down until you see the Notepad process (notepad/0); select it, and click the Add button.
9. Scroll back up in the Instances box to the Mmc process (the Microsoft Management Console process running the System Monitor), select all the threads (mmc/0, mmc/1, and so on), and add them to the chart by clicking the Add button. Before you click Add, you should see something like the following dialog box.


Add Counters

○ Use local computer counters

● Select counters from computer:

\\JEANROS02

Performance object:

Thread

○ All counters

● Select counters from list

- ID Process
- ID Thread
- Priority Base
- Priority Current
- Start Address
- Thread State
- Thread Wait Reason

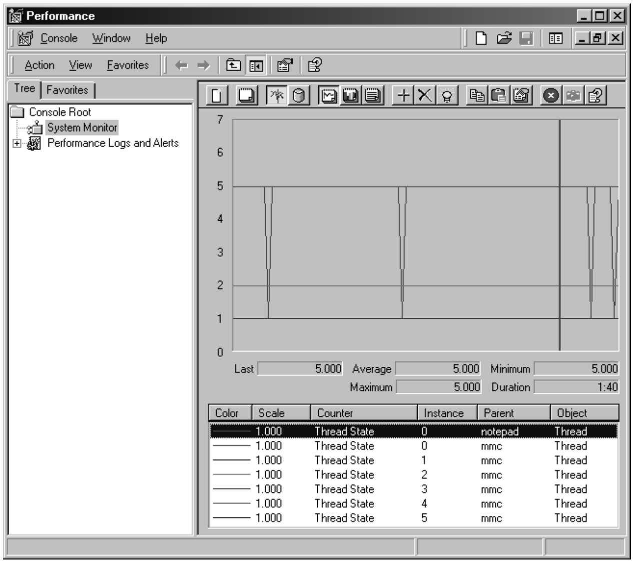
○ All instances

● Select instances from list:

- MAPISP32/6
- mmc/0
- mmc/1
- mmc/2
- mmc/3
- mmc/4
- mmc/5

10. Now close the Add Counters dialog box by clicking Close.

11. You should see the state of the Notepad thread (the very top line in the following figure) as a 5, which, as shown in the explanation text you saw under step 5, represents the waiting state (because the thread is waiting for GUI input):



Last 5.000 Average 5.000 Minimum 5.000
Maximum 5.000 Duration 1:40

Color	Scale	Counter	Instance	Parent	Object
	1.000	Thread State	0	notepad	Thread
	1.000	Thread State	0	mmc	Thread
	1.000	Thread State	1	mmc	Thread
	1.000	Thread State	2	mmc	Thread
	1.000	Thread State	3	mmc	Thread
	1.000	Thread State	4	mmc	Thread
	1.000	Thread State	5	mmc	Thread

12. Notice that one thread in the Mmc process (running the Performance tool snap-in) is in the running state (number 2). This is the thread that's querying the thread states, so it's always displayed in the running state.

13. You'll never see Notepad in the running state (unless you're on a multiprocessor system) because Mmc is always in the running state when it gathers the state of the threads you're monitoring.

The state diagram for threads on Windows Server 2003 is shown in Figure 6-15. Notice the new state called *deferred ready*. This state is used for threads that have been selected to run on a specific processor but have not yet been scheduled. This new state exists so that the kernel can minimize the amount of time the systemwide lock on the scheduling database is held. (This process is explained further in the section "Multiprocessor Dispatcher Database.")

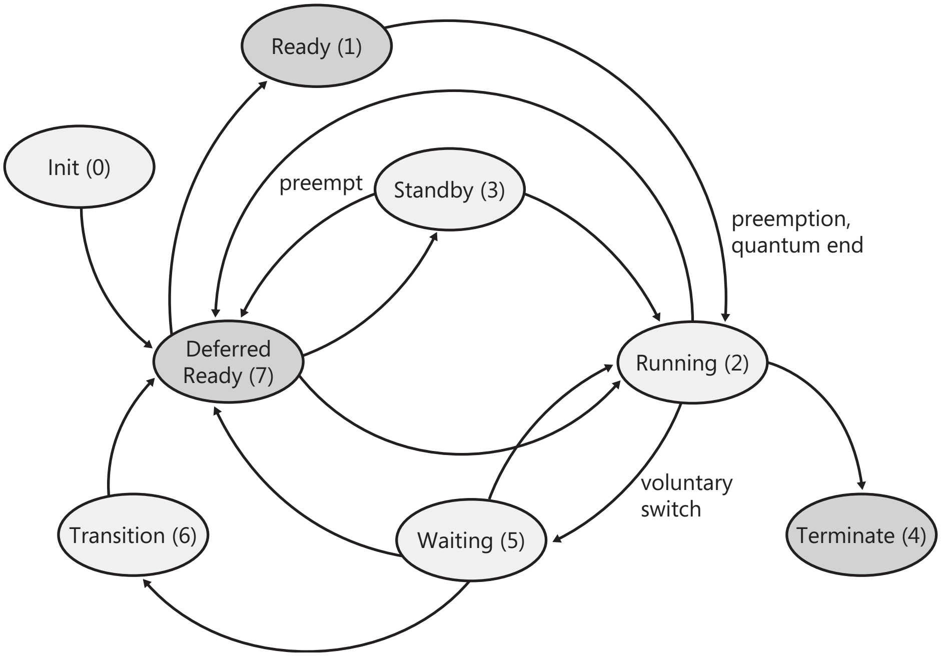


Figure 6-15 Thread states on Windows Server 2003

Dispatcher Database

To make thread-scheduling decisions, the kernel maintains a set of data structures known collectively as the *dispatcher database*, illustrated in Figure 6-16. The dispatcher database keeps track of which threads are waiting to execute and which processors are executing which threads.

> **Note** The dispatcher database on a uniprocessor system has the same structure as on multiprocessor Windows 2000 and Windows XP systems, but is different on Windows Server 2003 systems. These differences, as well as the differences in the way Windows selects threads to run on multiprocessor systems, are explained in the section "Multiprocessor Systems."

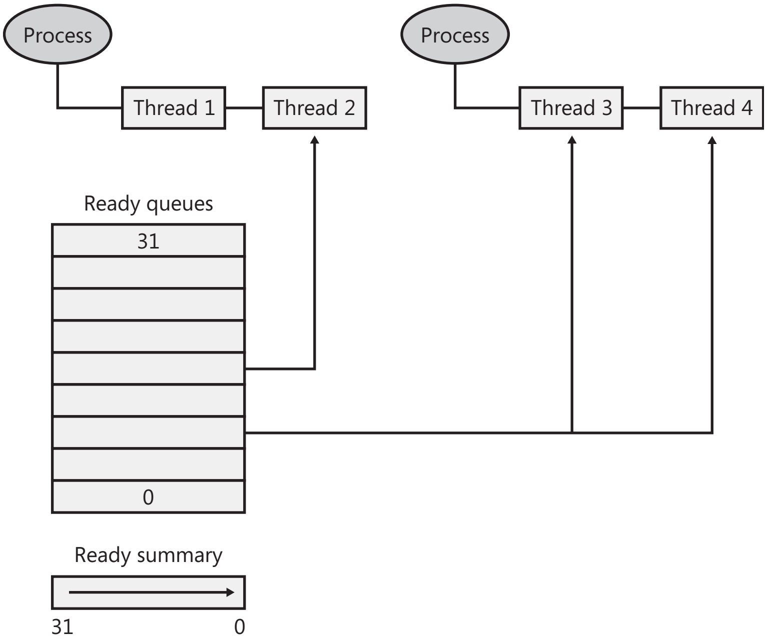


Figure 6-16 Dispatcher database (uniprocessor and Windows 2000/XP multiprocessor)

The dispatcher *ready queues (KiDispatcherReadyListHead)* contain the threads that are in the ready state, waiting to be scheduled for execution. There is one queue for each of the 32 priority levels. To speed up the selection of which thread to run or preempt, Windows maintains a 32-bit bit mask called the *ready summary* (*KiReadySummary*). Each bit set indicates one or more threads in the ready queue for that priority level. (Bit 0 represents priority 0, and so on.)

Table 6-15 lists the kernel-mode kernel variables that are related to thread scheduling on uniprocessor systems.

Table 6-14 Thread-Scheduling Kernel Variables

Variable	Type	Description
KiReadySummary	Bitmask (32 bits)	Bitmask of priority levels that have one or more ready threads
KiDispatcherReadyListHead	Array of 32 list entries	List heads for the 32 ready queues

On uniprocessor systems, the dispatcher database is synchronized by raising IRQL to DPC/dispatch level and SYNCH_LEVEL (both of which are defined as level 2). (For an explanation of interrupt priority levels, see the “Trap Dispatching” section in Chapter 3.) Raising IRQL in this way prevents other threads from interrupting thread dispatching because threads normally run at IRQL 0 or 1. On multiprocessor systems, more is required than raising IRQL because each processor can, at the same time, raise to the same IRQL and attempt to operate on the dispatcher database. How Windows synchronizes access to the dispatcher database on multiprocessor systems is explained in the “Multiprocessor Systems” section later in the chapter.

Quantum

As mentioned earlier in the chapter, a quantum is the amount of time a thread gets to run before Windows checks to see whether another thread at the same priority is waiting to run. If a thread completes its quantum and there are no other threads at its priority, Windows permits the thread to run for another quantum.

On Windows 2000 Professional and Windows XP, threads run by default for 2 clock intervals; on Windows Server systems, by default, a thread runs for 12 clock intervals. (We’ll explain how you can change these values later.) The rationale for the longer default value on server systems is to minimize context switching. By having a longer quantum, server applications that wake up as the result of a client request have a better chance of completing the request and going back into a wait state before their quantum ends.

The length of the clock interval varies according to the hardware platform. The frequency of the clock interrupts is up to the HAL, not the kernel. For example, the clock interval for most x86 uniprocessors is about 10 milliseconds and for most x86 multiprocessors it is about 15 milliseconds. (The actual clock rate is not exactly a round number of milliseconds—see the following experiment for a way to check the actual clock interval.)

EXPERIMENT: Determining the Clock Interval Frequency

The Windows *GetSystemTimeAdjustment* function returns the clock interval. To determine the clock interval, download and run the Clockres program from *www.sysinternals.com*. Here’s the output from a uniprocessor x86 system:

```
C:\>clockres

ClockRes - View the system clock resolution
By Mark Russinovich
SysInternals - www.sysinternals.com

The system clock interval is 10.014400 ms
```

Quantum Accounting

Each process has a quantum value in the kernel process block. This value is used when giving a thread a new quantum. As a thread runs, its quantum is reduced at each clock interval. If there is no remaining thread quantum, the quantum end processing is triggered. If there is another thread at the same priority waiting to run, a context switch occurs to the next thread in the ready queue. Note that when the clock interrupt interrupts a DPC or another interrupt that was in progress, the thread that was in the running state has its quantum deducted, even if it hadn't been running for a full clock interval. If this was not done and device interrupts or DPCs occurred right before the clock interval timer interrupts, threads might not ever get their quantum reduced.

Internally, the quantum value is stored as a multiple of three times the number of clock ticks. This means that on Windows 2000 and Windows XP systems, threads, by default, have a quantum value of 6 (2 * 3), and Windows Server systems have a quantum value of 36 (12 * 3). Each time the clock interrupts, the clock-interrupt routine deducts a fixed value (3) from the thread quantum.

The reason quantum is stored internally in terms of a multiple of 3 quantum units per clock tick rather than as single units is to allow for partial quantum decay on wait completion. When a thread that has a current priority less than 16 and a base priority less than 14 executes a wait function (such as *WaitForSingleObject* or *WaitForMultipleObjects*) that is satisfied immediately (for example, without having to wait), its quantum is reduced by 1 quantum unit. In this way, threads that wait will eventually expire their quantum.

In the case where a wait is not satisfied immediately, threads below priority 16 also have their quantum reduced by 1 unit (except if the thread is waking up to execute a kernel APC because the wait code will charge the quantum after the wait is actually satisfied). However, before doing the reduction, if the thread is at priority 14 or above, its quantum is reset to a full turn. This is also done for threads running at less than 14 if they are not running with a special priority boost (such as is done for foreground processes or in the case of CPU starvation) and if they are receiving a priority boost as a result of the unwait operation. (Priority boosting is explained in the next section.)

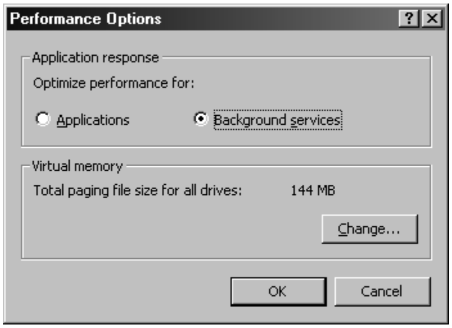
This partial decay addresses the case in which a thread enters a wait state before the clock interval timer fires. If this adjustment were not made, it would be possible for threads never to have their quantums reduced. For example, if a thread ran, entered a wait state, ran again, and entered another wait state but was never the currently running thread when the clock interval timer fired, it would never have its quantum charged for the time it was running. Note that this does not apply to zero timeout waits, but only wait operations that do not require waiting because all the wait conditions are fulfilled at the time of the wait.

Controlling the Quantum

You can change the thread quantum for all processes, but you can choose only one of two settings: short (2 clock ticks, the default for client machines) or long (12 clock ticks, the default for server systems).

> **Note** By using the job object on a system running with long quantums, you can select other quantum values for the processes in the job. For more information on the job object, see the "Job Objects" section later in the chapter.

To change this on Windows 2000, right click My Computer, Properties (or go to Control Panel and open the System settings applet), click the Advanced tab, and then click the Performance Options button. The dialog box you will see is shown in Figure 6-17.



Performance Options

Application response

Optimize performance for:

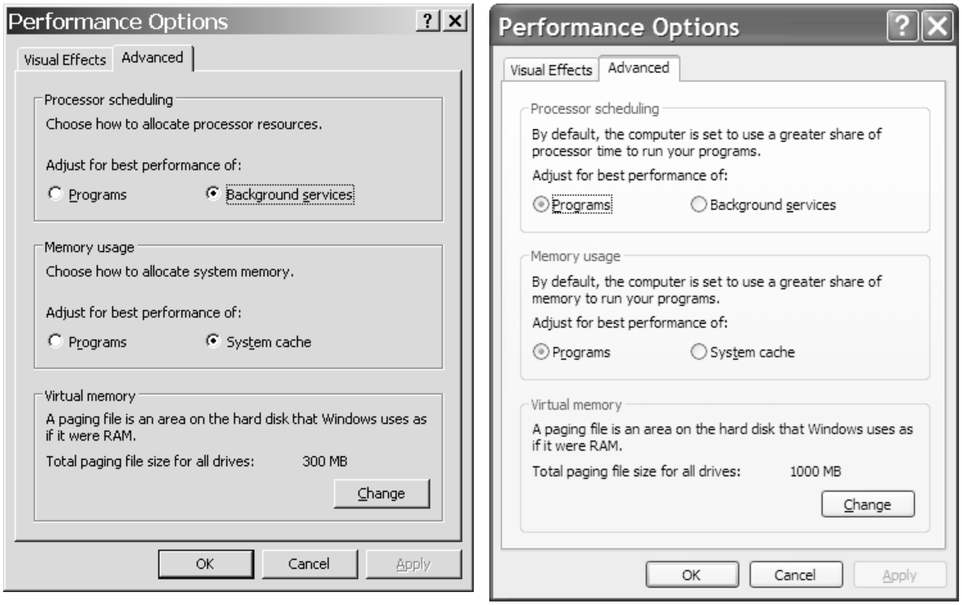
○ Applications ● Background services

Virtual memory

Total paging file size for all drives: 144 MB

Figure 6-17 Quantum Configuration on Windows 2000

To change this on Windows XP and Windows Server 2003, right click My Computer, Properties, click the Advanced tab, click the Settings button in the Performance section, and finally click the Advanced tab. The dialog box displayed is slightly different for Windows XP and Windows Server 2003. These are shown in Figure 6-18.



Performance Options

Visual Effects | Advanced

Processor scheduling

Choose how to allocate processor resources.

Adjust for best performance of:

○ Programs ● Background services

Memory usage

Choose how to allocate system memory.

Adjust for best performance of:

○ Programs ● System cache

Virtual memory

A paging file is an area on the hard disk that Windows uses as if it were RAM.

Total paging file size for all drives: 300 MB

Performance Options

Visual Effects | Advanced

Processor scheduling

By default, the computer is set to use a greater share of processor time to run your programs.

Adjust for best performance of:

● Programs ○ Background services

Memory usage

By default, the computer is set to use a greater share of memory to run your programs.

Adjust for best performance of:

● Programs ○ System cache

Virtual memory

A paging file is an area on the hard disk that Windows uses as if it were RAM.

Total paging file size for all drives: 1000 MB

Figure 6-18 Quantum Configuration on Windows XP and Windows Server 2003

The Programs setting (called "Applications" on Windows 2000) designates the use of short, variable quantums—the default for Windows 2000 Professional and Windows XP. If you install Terminal Services on Windows Server systems and configure the server as an application server, this setting is selected so that the users on the terminal server will have the same quantum settings that would normally be set on a desktop or client system. You might also select this manually if you were running Windows Server as your desktop operating system.

The Background Services option designates the use of long, fixed quantums—the default for Windows Server systems. The only reason why you might select this on a workstation system is if you were using the workstation as a server system.

One additional difference between the Programs and Background Services settings is the effect they have on the quantum of the threads in the foreground process. This is explained in the next section.

Quantum Boosting

Prior to Windows NT 4.0, when a window was brought into the foreground on a workstation or client system, all the threads in the foreground process (the process that owns the thread that owns the window that's in focus) received a priority boost of 2. This priority boost remained in effect while any thread in the process owned the foreground window. The problem with this approach was that if you started a long-running, CPU-intensive process (such as a spreadsheet recalculation) and then switched to another CPU-intensive process (such as a computer-aided design tool, graphics editor, or a game), the process now running in the background would get little or no CPU time because the foreground process would have its threads boosted by 2 (assuming the base priority of the threads in both processes are the same) while it remained in the foreground.

This default behavior was changed as of Windows NT 4.0 Workstation to instead triple the quantum of the threads in the foreground process. Thus, threads in the foreground process run with a quantum of 6 clock ticks, whereas threads in other processes have the default workstation quantum of 2 clock ticks. In this way, when you switch away from a CPU-intensive process, the new foreground process will get proportionally more of the CPU, because when its threads run they will have a longer turn that background threads (again, assuming the thread priorities are the same in both the foreground and background processes).

Note that this adjustment of quantums applies only to processes with a priority higher than Idle on systems configured to Programs (or Applications, in Windows 2000) in the Performance Options settings described in the previous section. Thread quantums are not changed for the foreground process on systems configured to Background Services (the default on Windows Server systems).

Quantum Settings Registry Value

The user interface to control quantum settings described earlier modify the registry value HKLM\SYSTEM\CurrentControlSet\Control\PriorityControl\Win32PrioritySeparation. In addition to specifying the relative length of thread quantums (short or long), this registry value also defines whether or not threads in the foreground process should have their quantums boosted (and if so, the amount of the boost). This value consists of 6 bits divided into the three 2-bit fields shown in Figure 6-19.

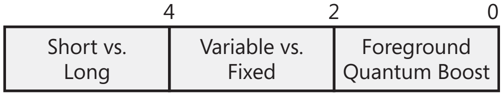


Figure 6-19 Fields of the Win32PrioritySeparation registry value

The fields shown in Figure 6-19 can be defined as follows:

- **Short vs. Long** A setting of 1 specifies long, and 2 specifies short. A setting of 0 or 3 indicates that the default will be used (short for Windows 2000 Professional and Windows XP, long for Windows Server systems).
- **Variable vs. Fixed** A setting of 1 means to vary the quantum for the foreground process, and 2 means that quantum values don't change for foreground processes. A setting of 0 or 3 means that the default (which is variable for Windows 2000 Professional and Windows XP and fixed for Windows Server systems) will be used.
- **Foreground Quantum Boost** This field (stored in the kernel variable *PsPrioritySeparation*) must have a value of 0, 1, or 2. (A setting of 3 is invalid and treated as 2.) It is used as an index into a three-element byte array named *PspForegroundQuantum* to obtain the quantum for the threads in the foreground process. The quantum for threads in background processes is taken from the first entry in this quantum table. Table 6-16 shows the possible settings for *PspForegroundQuantum*.

Table 6-15 **Quantum Values**

	Short			Long		
Variable	6	12	18	12	24	36
Fixed	18	18	18	36	36	36

Note that when you're using the Performance Options dialog box described earlier, you can choose from only two combinations: short quantums with foreground quantums tripled, or long quantums with no quantum changes for foreground threads. However, you can select other combinations by modifying the Win32PrioritySeparation registry value directly.

Scheduling Scenarios

Windows bases the question of “Who gets the CPU?” on thread priority; but how does this approach work in practice? The following sections illustrate just how priority-driven preemptive multitasking works on the thread level.

Voluntary Switch

First a thread might voluntarily relinquish use of the processor by entering a wait state on some object (such as an event, a mutex, a semaphore, an I/O completion port, a process, a thread, a window message, and so on) by calling one of the Windows wait functions (such as *WaitForSingleObject* or *WaitForMultipleObjects*). Waiting for objects is described in more detail in Chapter 3.

Figure 6-20 illustrates a thread entering a wait state and Windows selecting a new thread to run.

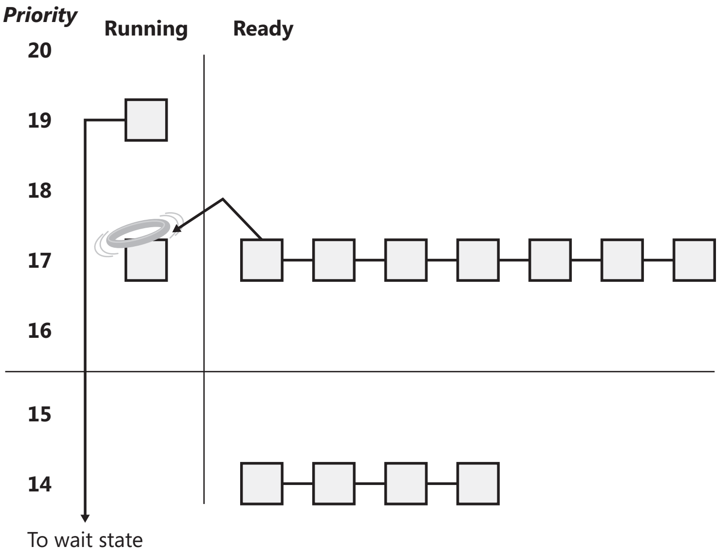


Figure 6-20 Voluntary switching

In Figure 6-20, the top block (thread) is voluntarily relinquishing the processor so that the next thread in the ready queue can run (as represented by the halo it has when in the Running column). Although it might appear from this figure that the relinquishing thread’s priority is being reduced, it’s not—it’s just being moved to the wait queue of the objects the thread is waiting for. What about any remaining quantum for the thread? The quantum value isn’t reset when a thread enters a wait state—in fact, as explained earlier, when the wait is satisfied, the thread’s quantum value is decremented by 1 quantum unit, equivalent to one-third of a clock interval (except for threads running at priority 14 or higher, which have their quantum reset after a wait to a full turn).

Preemption

In this scheduling scenario, a lower-priority thread is preempted when a higher-priority thread becomes ready to run. This situation might occur for a couple of reasons:

- A higher-priority thread's wait completes. (The event that the other thread was waiting for has occurred.)
- A thread priority is increased or decreased.

In either of these cases, Windows must determine whether the currently running thread should still continue to run or whether it should be preempted to allow a higher-priority thread to run.

> **Note** Threads running in user mode can preempt threads running in kernel mode—the mode in which the thread is running doesn't matter. The thread priority is the determining factor.

When a thread is preempted, it is put at the head of the ready queue for the priority it was running at. Figure 6-21 illustrates this situation.

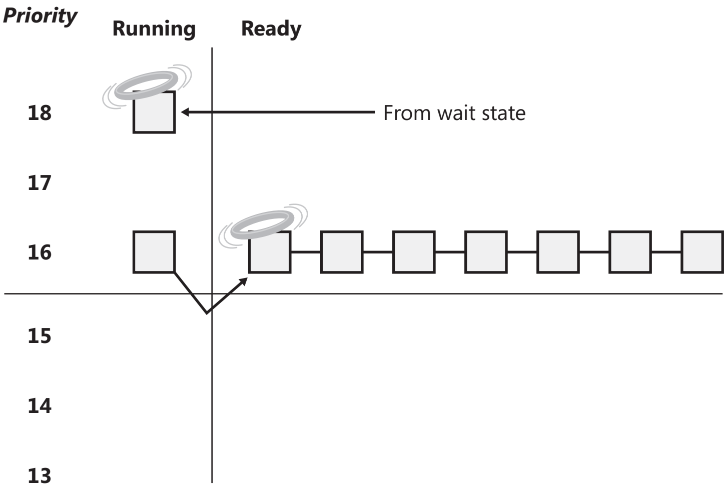


Figure 6-21 Preemptive thread scheduling

In Figure 6-21, a thread with priority 18 emerges from a wait state and repossesses the CPU, causing the thread that had been running (at priority 16) to be bumped to the head of the ready queue. Notice that the bumped thread isn't going to the end of the queue but to the beginning; when the preempting thread has finished running, the bumped thread can complete its quantum.

Quantum End

When the running thread exhausts its CPU quantum, Windows must determine whether the thread's priority should be decremented and then whether another thread should be scheduled on the processor.

If the thread priority is reduced, Windows looks for a more appropriate thread to schedule. (For example, a more appropriate thread would be a thread in a ready queue with a higher priority than the new priority for the currently running thread.) If the thread priority isn't reduced and there are other threads in the ready queue at the same priority level, Windows selects the next thread in the ready queue at that same priority level and moves the previously running thread to the tail of that queue (giving it a new quantum value and changing its state from running to ready). This case is illustrated in Figure 6-22. If no other thread of the same priority is ready to run, the thread gets to run for another quantum.

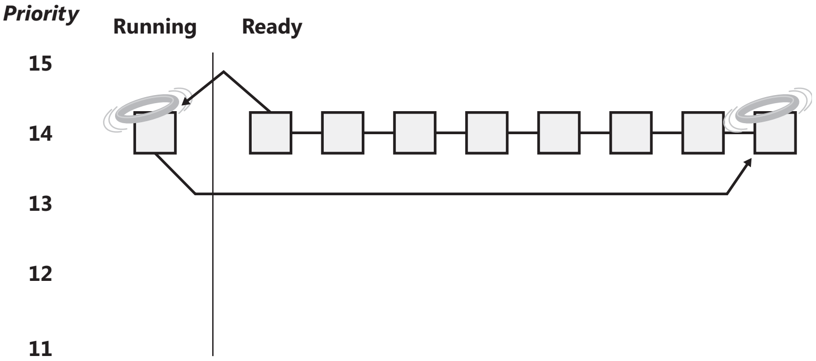


Figure 6-22 Quantum end thread scheduling

Termination

When a thread finishes running (either because it returned from its main routine, called *ExitThread*, or was killed with *TerminateThread*), it moves from the running state to the terminated state. If there are no handles open on the thread object, the thread is removed from the process thread list and the associated data structures are deallocated and released.

Context Switching

A thread's context and the procedure for context switching vary depending on the processor's architecture. A typical context switch requires saving and reloading the following data:

- Instruction pointer
- User and kernel stack pointers
- A pointer to the address space in which the thread runs (the process's page table directory)

The kernel saves this information from the old thread by pushing it onto the current (old thread's) kernel-mode stack, updating the stack pointer, and saving the stack pointer in the old thread's KTHREAD block. The kernel stack pointer is then set to the new thread's kernel stack, and the new thread's context is loaded. If the new thread is in a different process, it loads the address of its page table directory into a special processor register so that its address space is available. (See the description of address translation in Chapter 7.) If a kernel APC that needs to be delivered is pending, an interrupt at IRQL 1 is requested. Otherwise, control passes to the new thread's restored instruction pointer and the new thread resumes execution.

Idle Thread

When no runnable thread exists on a CPU, Windows dispatches the per-CPU idle thread. Each CPU is allotted one idle thread because on a multiprocessor system one CPU can be executing a thread while other CPUs might have no threads to execute.

Various Windows process viewer utilities report the idle process using different names. Task Manager and Process Explorer call it "System Idle Process," Process Viewer reports it as "Idle," Pstat calls it "Idle Process," Process Explode and Tlist call it "System Process," and Qslice calls it "SystemProcess." Windows reports the priority of the idle thread as 0. In reality, however, the idle threads don't have a priority level because they run only when there are no real threads to run. (Remember, only one thread per Windows system is actually running at priority 0—the zero page thread, explained in Chapter 7.)

The idle loop runs at DPC/dispatch level, polling for work to do, such as delivering deferred procedure calls (DPCs) or looking for threads to dispatch to. Although some details of the flow vary between architectures, the basic flow of control of the idle thread is as follows:

1. Enables and disables interrupts (allowing any pending interrupts to be delivered).
2. Checks whether any DPCs (described in Chapter 3) are pending on the processor. If DPCs are pending, clears the pending software interrupt and delivers them.
3. Checks whether a thread has been selected to run next on the processor, and if so, dispatches that thread.
4. Calls the HAL processor idle routine (in case any power management functions need to be performed).

In Windows Server 2003, the idle thread also scans for threads waiting to run on other processors. (This is explained in the upcoming multiprocessor scheduling section.)

Priority Boosts

In five cases, Windows can boost (increase) the current priority value of threads:

- On completion of I/O operations
- After waiting for executive events or semaphores

- After threads in the foreground process complete a wait operation
- When GUI threads wake up because of windowing activity
- When a thread that's ready to run hasn't been running for some time (CPU starvation)

The intent of these adjustments is to improve overall system throughput and responsiveness as well as resolve potentially unfair scheduling scenarios. Like any scheduling algorithms, however, these adjustments aren't perfect, and they might not benefit all applications.

Note Windows never boosts the priority of threads in the real-time range (16 through 31). Therefore, scheduling is always predictable with respect to other threads in the real-time range. Windows assumes that if you're using the real-time thread priorities, you know what you're doing.

Priority Boosting after I/O Completion

Windows gives temporary priority boosts upon completion of certain I/O operations so that threads that were waiting for an I/O will have more of a chance to run right away and process whatever was being waited for. Recall that 1 quantum unit is deducted from the thread's remaining quantum when it wakes up so that I/O bound threads aren't unfairly favored. Although you'll find recommended boost values in the DDK header files (by searching for "#define IO" in Wdm.h or Ntddk.h), the actual value for the boost is up to the device driver. (These values are listed in Table 6-17.) It is the device driver that specifies the boost when it completes an I/O request on its call to the kernel function *IoCompleteRequest*. In Table 6-17, notice that I/O requests to devices that warrant better responsiveness have higher boost values.

Table 6-16 Recommended Boost Values

Device	Boost
Disk, CD-ROM, parallel, video	1
Network, mailslot, named pipe, serial	2
Keyboard, mouse	6
Sound	8

The boost is always applied to a thread's base priority, not its current priority. As illustrated in Figure 6-23, after the boost is applied, the thread gets to run for one quantum at the elevated priority level. After the thread has completed its quantum, it decays one priority level and then runs another quantum. This cycle continues until the thread's priority level has decayed back to its base priority. A thread with a higher priority can still preempt the boosted thread, but the interrupted thread gets to finish its time slice at the boosted priority level before it decays to the next lower priority.

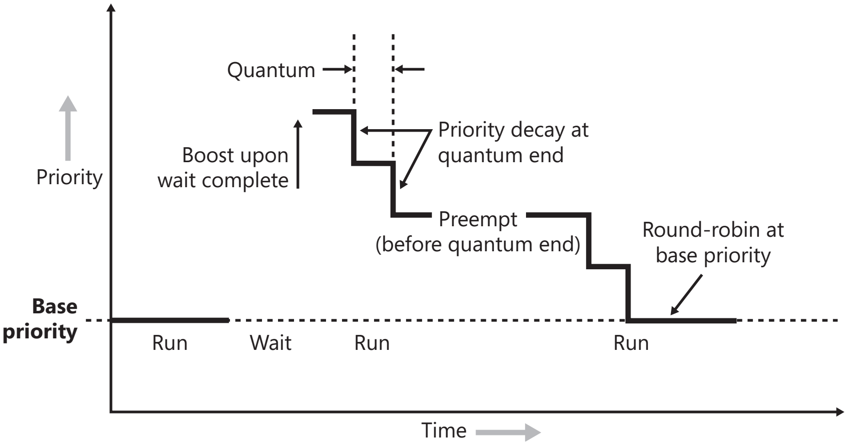

Quantum
Priority
Boost upon wait complete
Priority decay at quantum end
Preempt (before quantum end)
Round-robin at base priority
Base priority
Run
Wait
Run
Run
Time

Figure 6-23 Priority boosting and decay

As noted earlier, these boosts apply only to threads in the dynamic priority range (0 through 15). No matter how large the boost is, the thread will never be boosted beyond level 15 into the real-time priority range. In other words, a priority 14 thread that receives a boost of 5 will go up to priority 15. A priority 15 thread that receives a boost will remain at priority 15.

Boosts after Waiting for Events and Semaphores

When a thread that was waiting for an executive event or a semaphore object has its wait satisfied (because of a call to the function *SetEvent*, *PulseEvent*, or *ReleaseSemaphore*), it receives a boost of 1. (See the value for EVENT_ INCREMENT and SEMAPHORE_INCREMENT in the DDK header files.) Threads that wait for events and semaphores warrant a boost for the same reason that threads that wait for I/O operations do—threads that block on events are requesting CPU cycles less frequently than CPU-bound threads. This adjustment helps balance the scales.

This boost operates the same as the boost that occurs after I/O completion, as described in the previous section:

- The boost is always applied to the base priority (not the current priority).
- The priority will never be boosted over 15.
- The thread gets to run at the elevated priority for its remaining quantum (as described earlier, quantums are reduced by 1 when threads exit a wait) before decaying one priority level at a time until it reaches its original base priority.

A special boost is applied to threads that are awoken as a result of setting an event with the special functions *NtSetEventBoostPriority* (used in Ntdll.dll for critical sections) and *KeSetEventBoostPriority* (used for executive resources and push locks). If a thread waiting for an event is woken up as a result of the special event boost function and its priority is 13 or below, it will have its priority boosted to be the setting thread's priority plus one. If its quantum is less than 4 quantum units, it is set to 4 quantum units. This boost is removed at quantum end.

Priority Boosts for Foreground Threads after Waits

Whenever a thread in the foreground process completes a wait operation on a kernel object, the kernel function *KiUnwaitThread* boosts its current (not base) priority by the current value of *PsPrioritySeparation*. (The windowing system is responsible for determining which process is considered to be in the foreground.) As described in the section on quantum controls, *PsPrioritySeparation* reflects the quantum-table index used to select quantums for the threads of foreground applications. However, in this case, it is being used as a priority boost value.

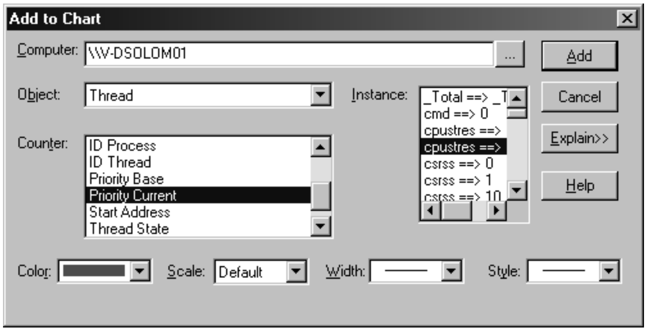
The reason for this boost is to improve the responsiveness of interactive applications—by giving the foreground application a small boost when it completes a wait, it has a better chance of running right away, especially when other processes at the same base priority might be running in the background.

Unlike other types of boosting, this boost applies to all Windows systems, and you *can't* disable this boost, even if you've disabled priority boosting using the Windows *SetThreadPriorityBoost* function.

EXPERIMENT: Watching Foreground Priority Boosts and Decays

Using the CPU Stress tool (in the resource kit and the Platform SDK), you can watch priority boosts in action. Take the following steps:

1. Open the System utility in Control Panel (or right-click My Computer and select Properties), click the Advanced tab, and click the Performance Options button. Select the Applications option. This causes *PsPrioritySeparation* to get a value of 2.
2. Run Cpustres.exe.
3. Run the Windows NT 4 Performance Monitor (Perfmon4.exe in the Windows 2000 resource kits). This older version of the Performance tool is needed for this experiment because it can query performance counter values at a frequency faster than the Windows Performance tool (which has a maximum interval of once per second).
4. Click the Add Counter toolbar button (or press Ctrl+I) to bring up the Add To Chart dialog box.
5. Select the Thread object, and then select the Priority Current counter.
6. In the Instance box, scroll down the list until you see the cpustres process. Select the second thread (thread 1). (The first thread is the GUI thread.) You should see something like this:



Add to Chart

Computer: \\V-DSOLOM01

Object: Thread

Instance: _Total ==> _T
cmd ==> 0
cpustres ==>
cpustres ==>
csrss ==> 0
csrss ==> 1
csrss ==> 10

Counter: ID Process
ID Thread
Priority Base
Priority Current
Start Address
Thread State

Color:

Scale: Default

Width:

Style:

7. Click the Add button, and then click the Done button.

8. Select Chart from the Options menu. Change the Vertical Maximum to 16 and the Interval to 0.010, as follows, and click OK:

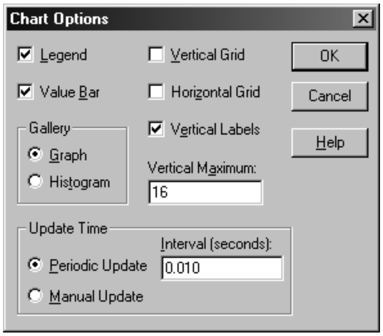


Chart Options

☑ Legend

☑ Value Bar

□ Vertical Grid

□ Horizontal Grid

☑ Vertical Labels

Gallery

● Graph

○ Histogram

Vertical Maximum: 16

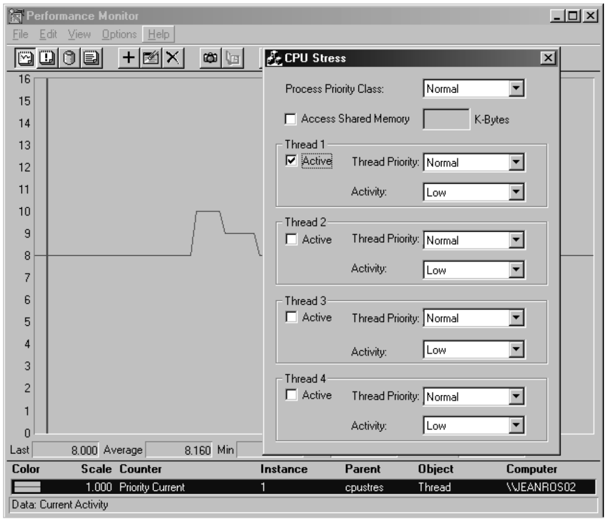
Update Time

● Periodic Update

○ Manual Update

Interval (seconds): 0.010

9. Now bring the Cpustres process to the foreground. You should see the priority of the Cpustres thread being boosted by 2 and then decaying back to the base priority as follows:



CPU Stress

Process Priority Class: Normal

□ Access Shared Memory K-Bytes

Thread 1

☑ Active Thread Priority: Normal

Activity: Low

Thread 2

□ Active Thread Priority: Normal

Activity: Low

Thread 3

□ Active Thread Priority: Normal

Activity: Low

Thread 4

□ Active Thread Priority: Normal

Activity: Low

Last 8.000 Average 8.160 Min

Color	Scale	Counter	Instance	Parent	Object	Computer
▭	1.000	Priority Current	1	cpustres	Thread	\\JEANROS02

Data: Current Activity

10. The reason Cpustres receives a boost of 2 periodically is because the thread you're monitoring is sleeping about 75 percent of the time and then waking up—the boost is applied when the thread wakes up. To see the thread get boosted more frequently, increase the Activity level from Low to Medium to Busy. If you set the Activity level to Maximum, you won't see any boosts because Maximum in Cpustres puts the thread into an infinite loop. Therefore, the thread doesn't invoke any wait functions and therefore doesn't receive any boosts.
11. When you've finished, exit Performance Monitor and CPU Stress.

Priority Boosts after GUI Threads Wake Up

Threads that own windows receive an additional boost of 2 when they wake up because of windowing activity, such as the arrival of window messages. The windowing system (Win32k.sys) applies this boost when it calls *KeSetEvent* to set an event used to wake up a GUI thread. The reason for this boost is similar to the previous one—to favor interactive applications.

EXPERIMENT: Watching Priority Boosts on GUI Threads

You can also see the windowing system apply its boost of 2 for GUI threads that wake up to process window messages by monitoring the current priority of a GUI application and moving the mouse across the window. Just follow these steps:

1. Open the System utility in Control Panel (or right-click My Computer and select Properties), click the Advanced tab, and click the Performance Options button. If you're running Windows XP or Windows Server 2003 select the Advanced tab and ensure that the Programs option is selected; if you're running Windows 2000 ensure that the Applications option is selected. This causes *PsPrioritySeparation* to get a value of 2.
2. Run Notepad from the Start menu by selecting Programs/Accessories/Notepad.
3. Run the Windows NT 4 Performance Monitor (Perfmon4.exe in the Windows 2000 resource kits). This older version of the Performance tool is needed for this experiment because it can query performance counter values at a faster frequency. (The Windows Performance tool has a maximum interval of once per second.)
4. Click the Add Counter toolbar button (or press Ctrl+I) to bring up the Add To Chart dialog box.
5. Select the Thread object, and then select the Priority Current counter.
6. In the Instance box, scroll down the list until you see Notepad thread 0. Click it, click the Add button, and then click the Done button.
7. As in the previous experiment, select Chart from the Options menu. Change the Vertical Maximum to 16 and the Interval to 0.010, and click OK.

8. You should see the priority of thread 0 in Notepad at 8, 9, or 10. Because Notepad entered a wait state shortly after it received the boost of 2 that threads in the foreground process receive, it might not yet have decayed from 10 to 9 and then to 8.
9. With Performance Monitor in the foreground, move the mouse across the Notepad window. (Make both windows visible on the desktop.) You'll see that the priority sometimes remains at 10 and sometimes at 9, for the reasons just explained. (The reason you won't likely catch Notepad at 8 is that it runs so little after receiving the GUI thread boost of 2 that it never experiences more than one priority level of decay before waking up again because of additional windowing activity and receiving the boost of 2 again.)
10. Now bring Notepad to the foreground. You should see the priority rise to 12 and remain there (or drop to 11, because it might experience the normal priority decay that occurs for boosted threads on the quantum end) because the thread is receiving two boosts: the boost of 2 applied to GUI threads when they wake up to process windowing input and an additional boost of 2 because Notepad is in the foreground.
11. If you then move the mouse over Notepad (while it's still in the foreground), you might see the priority drop to 11 (or maybe even 10) as it experiences the priority decay that normally occurs on boosted threads as they complete quantums. However, the boost of 2 that is applied because it's the foreground process remains as long as Notepad remains in the foreground.
12. When you've finished, exit Performance Monitor and Notepad.

Priority Boosts for CPU Starvation

Imagine the following situation: you have a priority 7 thread that's running, preventing a priority 4 thread from ever receiving CPU time; however, a priority 11 thread is waiting for some resource that the priority 4 thread has locked. But because the priority 7 thread in the middle is eating up all the CPU time, the priority 4 thread will never run long enough to finish whatever it's doing and release the resource blocking the priority 11 thread. What does Windows do to address this situation? Once per second, the *balance set manager* (a system thread that exists primarily to perform memory management functions and is described in more detail in Chapter 7) scans the ready queues for any threads that have been in the ready state (that is, haven't run) for approximately 4 seconds. If it finds such a thread, the balance set manager boosts the thread's priority to 15. On Windows 2000 and Windows XP, the thread quantum is set to twice the process quantum. On Windows Server 2003, the quantum is set to 4 quantum units. Once the quantum is expired, the thread's priority decays immediately to its original base priority. If the thread wasn't finished and a higher priority thread is ready to run, the decayed thread will return to the ready queue, where it again becomes eligible for another boost if it remains there for another 4 seconds.

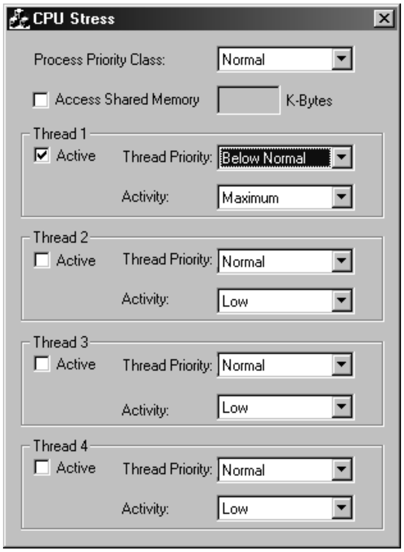
The balance set manager doesn't actually scan all ready threads every time it runs. To minimize the CPU time it uses, it scans only 16 ready threads; if there are more threads at that priority level, it remembers where it left off and picks up again on the next pass. Also, it will boost only 10 threads per pass—if it finds 10 threads meriting this particular boost (which would indicate an unusually busy system), it stops the scan at that point and picks up again on the next pass.

Will this algorithm always solve the priority inversion issue? No—it's not perfect by any means. But over time, CPU-starved threads should get enough CPU time to finish whatever processing they were doing and reenter a wait state.

EXPERIMENT: Watching Priority Boosts for CPU Starvation

Using the CPU Stress tool (in the resource kit and the Platform SDK), you can watch priority boosts in action. In this experiment, we'll see CPU usage change when a thread's priority is boosted. Take the following steps:

1. Run Cpustres.exe. Change the activity level of the active thread (by default, Thread 1) from Low to Maximum. Change the thread priority from Normal to Below Normal. The screen should look like this:



CPU Stress

Process Priority Class: Normal

□ Access Shared Memory K-Bytes

Thread 1
☑ Active Thread Priority: Below Normal
Activity: Maximum

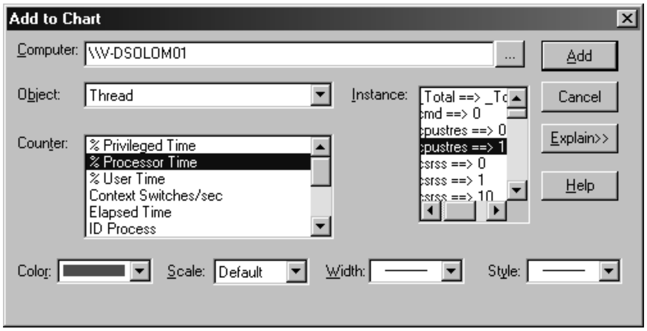
Thread 2
□ Active Thread Priority: Normal
Activity: Low

Thread 3
□ Active Thread Priority: Normal
Activity: Low

Thread 4
□ Active Thread Priority: Normal
Activity: Low

2. Run the Windows NT 4 Performance Monitor (Perfmon4.exe in the Windows 2000 resource kits). Again, you need the older version for this experiment because it can query performance counter values at a frequency faster than once per second.
3. Click the Add Counter toolbar button (or press Ctrl+I) to bring up the Add To Chart dialog box.
4. Select the Thread object, and then select the % Processor Time counter.

5. In the Instance box, scroll down the list until you see the cpustres process. Select the second thread (thread 1). (The first thread is the GUI thread.) You should see something like this:



Add to Chart

Computer: \\V-DSOLOM01

Object: Thread

Instance: _Total ==> _To; cmd ==> 0; cpustres ==> 0; cpustres ==> 1; csrss ==> 0; csrss ==> 1; csrss ==> 10

Counter: % Privileged Time; % Processor Time; % User Time; Context Switches/sec; Elapsed Time; ID Process

Color: ▇ Scale: Default Width: —— Style: ——

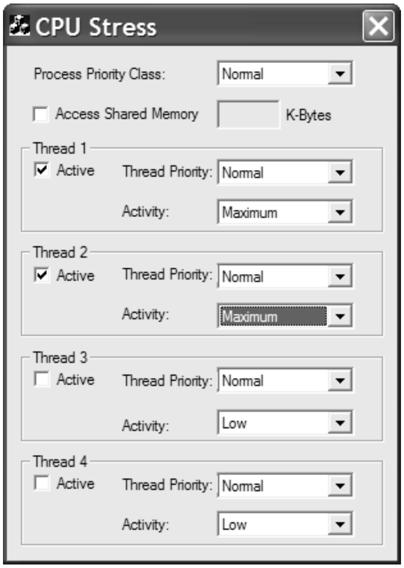
6. Click the Add button, and then click the Done button.

7. Raise the priority of Performance Monitor to real-time by running Task Manager, clicking the Processes tab, and selecting the Perfmon4.exe process. Right-click the process, select Set Priority, and then select Realtime. (If you receive a Task Manager Warning message box warning you of system instability, click the Yes button.)

8. Run another copy of CPU Stress. In this copy, change the activity level of Thread 1 from Low to Maximum.

9. Now switch back to Performance Monitor. You should see CPU activity every 4 or so seconds because the thread is boosted to priority 15.

When you've finished, exit Performance Monitor and the two copies of CPU Stress.



CPU Stress

Process Priority Class: Normal

□ Access Shared Memory K-Bytes

Thread 1
☑ Active Thread Priority: Normal
Activity: Maximum

Thread 2
☑ Active Thread Priority: Normal
Activity: Maximum

Thread 3
□ Active Thread Priority: Normal
Activity: Low

Thread 4
□ Active Thread Priority: Normal
Activity: Low

EXPERIMENT: “Listening” to Priority Boosting

To “hear” the effect of priority boosting for CPU starvation, perform the following steps on a system with a sound card:

1. Run Windows Media Player (or some other audio playback program), and begin playing some audio content.
2. Run Cpustres from the Windows 2000 resource kits, and set the activity level of thread 1 to maximum.
3. Raise the priority of thread 1 from Normal to Time Critical.
4. You should hear the music playback stop as the compute-bound thread begins consuming all available CPU time.
5. Every so often, you should hear bits of sound as the starved thread in the audio playback process gets boosted to 15 and runs enough to send more data to the sound card.
6. Stop Cpustres and Windows Media Player.

Multiprocessor Systems

On a uniprocessor system, scheduling is relatively simple: the highest priority thread that wants to run is always running. On a multiprocessor system, it is more complex, as Windows attempts to schedule threads on the most optimal processor for the thread, taking into account the thread’s preferred and previous processors, as well as the configuration of the multiprocessor system. Therefore, while Windows attempts to schedule the highest priority runnable threads on all available CPUs, it only guarantees to be running the (single) highest priority thread somewhere.

Before we describe the specific algorithms used to choose which threads run where and when, let’s examine the additional information Windows maintains to track thread and processor state on multiprocessor systems and the two new types of multiprocessor systems supported by Windows (hyperthreaded and NUMA).

Multiprocessor Dispatcher Database

As explained in the “Dispatcher Database” section earlier in the chapter, the dispatcher database refers to the information maintained by the kernel to perform thread scheduling. As shown in Figure 6-16, on multiprocessor Windows 2000 and Windows XP systems, the ready queues and ready summary have the same structure as they do on uniprocessor systems. In addition to the ready queues and the ready summary, Windows maintains two bitmasks that track the state of the processors on the system. (How these bitmasks are used is explained in the upcoming section “Multiprocessor Thread-Scheduling Algorithms”.) Following are the two bitmasks that Windows maintains:

- The *active processor mask (KeActiveProcessors)*, which has a bit set for each usable processor on the system (This might be less than the number of actual processors if the licensing limits of the version of Windows running supports less than the number of available physical processors.)
- The *idle summary* (*KiIdleSummary*), in which each set bit represents an idle processor

Whereas on uniprocessor systems, the dispatcher database is locked by raising IRQL (to DPC/dispatch level on Windows 2000 and Windows XP and to both DPC/dispatch level and Synch level on Windows Server 2003), on multiprocessor systems more is required, because each processor could, at the same time, raise IRQL and attempt to operate on the dispatcher database. (This is true for any systemwide structure accessed from high IRQL.) (See Chapter 3 for a general description of kernel synchronization and spinlocks.) On Windows 2000 and Windows XP, two kernel spinlocks are used to synchronize access to thread dispatching: the *dispatcher* spinlock (*KiDispatcherLock*) and the *context swap* spinlock (*KiContextSwapLock*). The former is held while changes are made to structures that might affect which thread should run. The latter is held after the decision is made but during the thread context swap operation itself.

To improve scalability including thread dispatching concurrency, Windows Server 2003 multiprocessor systems have per-processor dispatcher ready queues, as illustrated in Figure 6-24. In this way, on Windows Server 2003, each CPU can check its own ready queues for the next thread to run without having to lock the systemwide ready queues.

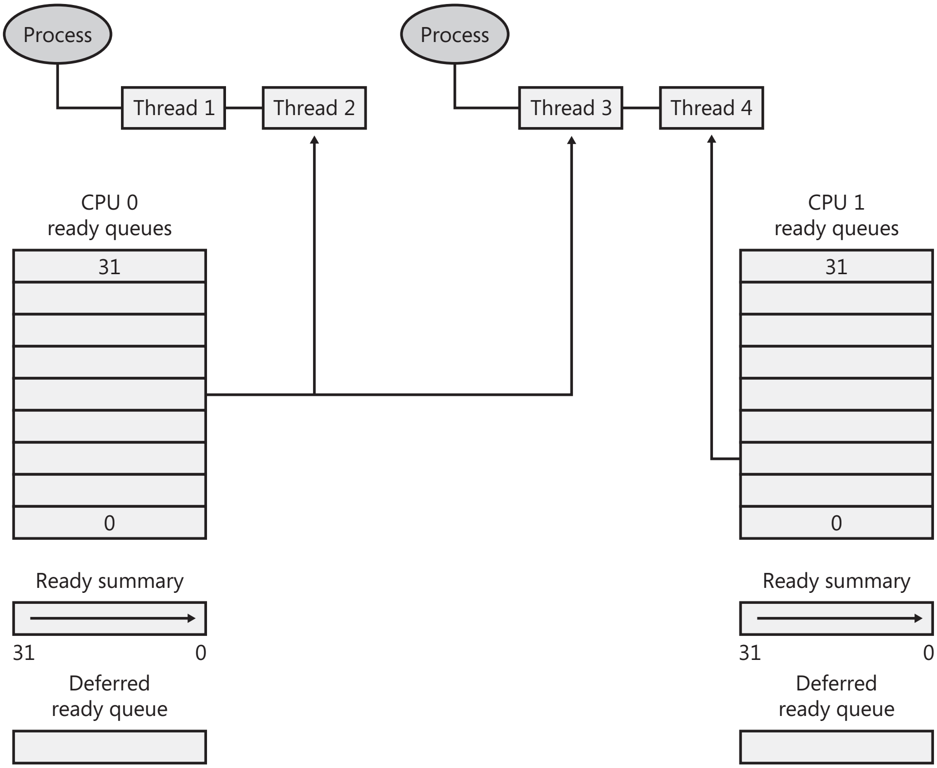


Figure 6-24 Windows Server 2003 multiprocessor dispatcher database

The per-processor ready queues, as well as the per-processor ready summary are part of the processor control block (PRCB) structure. (To see the fields in the PRCB, type **dt nt!_prcb** in the Kernel Debugger.) Because on a multiprocessor system one processor might need to modify another processor's per-CPU scheduling data structures (such as inserting a thread that would like to run on a certain processor), these structures are synchronized by using a new per-PRCB queued spinlock, which is held at IRQL SYNCH_LEVEL. (See Table 6-18 for the various values of SYNCH_LEVEL). Thus, thread selection can occur while locking only an individual processor's PRCB, in contrast to doing this on Windows 2000 and Windows XP, where the systemwide dispatcher spinlock had to be held.

Table 6-17 **IRQL SYNCH_LEVEL on Multiprocessor Systems**

Windows Version	IRQL
Windows 2000	2
Windows XP on x86	2
Windows Server 2003 on x86	27
Windows XP on x64	12
Windows XP on IA-64	2
Windows Server 2003 on x64 & IA-64	12

There is also a per-CPU list of threads in the deferred ready state. These represent threads that are ready to run but have not yet been readied for execution; the actual ready operation has been deferred to a more appropriate time. Because each processor manipulates only its own per-processor deferred ready list, this list is not synchronized by the PRCB spinlock. The deferred ready thread list is processed before exiting the thread dispatcher, before performing a context switch, and after processing a DPC. Threads on the deferred ready list are either dispatched immediately or are moved to the per-process ready queue for their priority level.

Note that the systemwide dispatcher spinlock still exists and is used on Windows Server 2003, but it is held only for the time needed to modify systemwide state that might affect which thread runs next. For example, changes to synchronization objects (mutexes, events, and semaphores) and their wait queues require holding the dispatcher lock to prevent more than one processor from changing the state of such objects (and the consequential action of possibly readying threads for execution). Other examples include changing the priority of a thread, timer expiration, and swapping of thread kernel stacks.

Finally, synchronization of thread context switching has also been improved on Windows Server 20003, as it is now synchronized by using a per-thread spinlock, whereas in Windows 2000 and Windows XP context switching was synchronized by holding a systemwide context swap spinlock.

Hyperthreaded Systems

As described in the “Symmetric Multiprocessing” section in Chapter 2, Windows XP and Windows Server 2003 support hyperthreaded multiprocessor systems in two primary ways:

1. Logical processors do not count against physical processor licensing limits. For example, Windows XP Home Edition, which has a licensed processor limit of 1, will use both logical processors on a single processor hyperthreaded system.
2. When choosing a processor for a thread, if there is a physical processor with all logical processors idle, a logical processor from that physical processor will be selected, as opposed to choosing an idle logical processor on a physical processor that has another logical processor running a thread.

EXPERIMENT: Viewing Hyperthreading Information

You can examine the information Windows maintains for hyperthreaded processors using the *!smt* command in the kernel debugger. The following output is from a dual processor hyperthreaded Xeon system (four logical processors):

```
lkd> !smt
SMT Summary:
------------
   KeActiveProcessors: ****---------------------------- (0000000f)
        KiIdleSummary: -***---------------------------- (0000000e)
No PRCB     Set Master SMT Set                                     #LP IAID
 0 ffdff120 Master     *-*----------------------------- (00000005)   2  00
 1 f771f120 Master     -*-*---------------------------- (0000000a)   2  06
 2 f7727120 ffdff120   *-*----------------------------- (00000005)   2  01
 3 f772f120 f771f120   -*-*---------------------------- (0000000a)   2  07

   Number of licensed physical processors: 2
```

Logical processor 0 and 1 are on separate physical processors (as indicated by the term “Master”).

NUMA Systems

Another type of multiprocessor system supported by Windows XP and Windows Server 2003 are those with nonuniform memory access (NUMA) architectures. In a NUMA system, processors are grouped together in smaller units called nodes. Each node has its own processors and memory, and is connected to the larger system through a cache-coherent interconnect bus. These systems are called “nonuniform” because each node has its own local high-speed memory. While any processor in any node can access all of memory, node-local memory is much faster to access.

The kernel maintains information about each node in a NUMA system in a data structure called KNODE. The kernel variable *KeNodeBlock* is an array of pointers to the KNODE structures for each node. The format of the KNODE structure can be shown using the *dt* command in the kernel debugger, as shown here:

```
lkd> dt nt!_knode
nt!_KNODE
   +0x000 ProcessorMask    : Uint4B
   +0x004 Color            : Uint4B
   +0x008 MmShiftedColor   : Uint4B
   +0x00c FreeCount        : [2] Uint4B
   +0x018 DeadStackList    : _SLIST_HEADER
   +0x020 PfnDereferenceSListHead : _SLIST_HEADER
   +0x028 PfnDeferredList  : Ptr32 _SINGLE_LIST_ENTRY
   +0x02c Seed             : UChar
   +0x02d Flags            : _flags
```

EXPERIMENT: Viewing NUMA Information

You can examine the information Windows maintains for each node in a NUMA system using the *!numa* command in the kernel debugger. The following partial output is from a 32-processor NUMA system by NEC with 4 processors per node:

```
21: kd> !numa
NUMA Summary:
------------
    Number of NUMA nodes : 8
    Number of Processors : 32
    MmAvailablePages     : 0x00F70D2C
    KeActiveProcessors   : ********************************---------------------------
----- (00000000ffffffff)

    NODE 0 (E00000008428AE00):
        ProcessorMask    : ****-------------------------------------------------------
-----
        Color            : 0x00000000
        MmShiftedColor   : 0x00000000
        Seed             : 0x00000000
        Zeroed Page Count: 0x00000000001CF330
        Free Page Count  : 0x0000000000000000

    NODE 1 (E00001597A9A2200):
        ProcessorMask    : ----****---------------------------------------------------
-----
        Color            : 0x00000001
        MmShiftedColor   : 0x00000040
        Seed             : 0x00000006
        Zeroed Page Count: 0x00000000001F77A0
        Free Page Count  : 0x0000000000000004
```

The following partial output is from a 64-processor NUMA system from Hewlett Packard with 4 processors per node:

```
26: kd> !numa
NUMA Summary:
------------
    Number of NUMA nodes : 16
    Number of Processors : 64
    MmAvailablePages     : 0x03F55E67
    KeActiveProcessors   : ***********************************************************
***** (ffffffffffffffff)

    NODE 0 (E000000084261900):
        ProcessorMask    : ****-------------------------------------------------------
-----
        Color            : 0x00000000
        MmShiftedColor   : 0x00000000
        Seed             : 0x00000001
        Zeroed Page Count: 0x00000000003F4430
        Free Page Count  : 0x0000000000000000

    NODE 1 (E0000145FF992200):
        ProcessorMask    : ----****---------------------------------------------------
-----
        Color            : 0x00000001
        MmShiftedColor   : 0x00000040
        Seed             : 0x00000007
        Zeroed Page Count: 0x00000000003ED59A
        Free Page Count  : 0x0000000000000000
```

Applications that want to gain the most performance out of NUMA systems can set the affinity mask to restrict a process to the processors in a specific node. This information can be obtained using the functions listed in Table 6-18. Functions that can alter thread affinity are listed in Table 6-13.

Table 6-18 NUMA-Related Functions

Function	Description
GetNumaHighestNodeNumber	Retrieves the node that currently has the highest number.
GetNumaNodeProcessorMask	Retrieves the processor mask for the specified node.
GetNumaProcessorNode	Retrieves the node number for the specified processor.

How the scheduling algorithms take into account NUMA systems will be covered in the upcoming section "Multiprocessor Thread-Scheduling Algorithms" (and the optimizations in the memory manager to take advantage of node-local memory are covered in Chapter 7).

Affinity

Each thread has an *affinity mask* that specifies the processors on which the thread is allowed to run. The thread affinity mask is inherited from the process affinity mask. By default, all processes (and therefore all threads) begin with an affinity mask that is equal to the set of active processors on the system—in other words, the system is free to schedule all threads on any available processor.

However, to optimize throughput and/or partition workloads to a specific set of processors, applications can choose to change the affinity mask for a thread. This can be done at several levels:

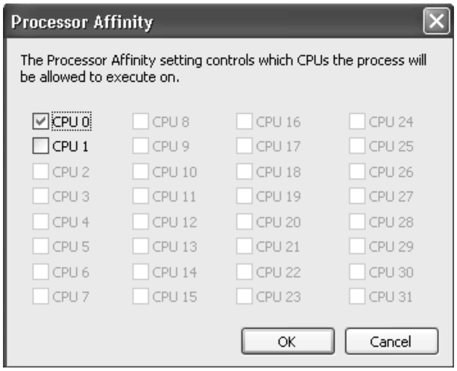
- Calling the *SetThreadAffinityMask* function to set the affinity for an individual thread
- Calling the *SetProcessAffinityMask* function to set the affinity for all the threads in a process. Task Manager and Process Explorer provide a GUI interface to this function if you right-click a process and choose Set Affinity. The Psexec tool (from *www.sysinternals.com*) provides a command-line interface to this function. (See the –*a* switch.)
- By making a process a member of a job that has a jobwide affinity mask set using the *SetInformationJobObject* function (Jobs are described in the upcoming "Job Objects" section.)
- By specifying an affinity mask in the image header using, for example, the Imagecfg tool in the Windows 2000 Server Resource Kit Supplement 1 (For more information on the detailed format of Windows images, see the article "Portable Executable and Common Object File Format Specification" in the MSDN Library.)

You can also set the "uniprocessor" flag for an image (using the Imagecfg –*u* switch). If this flag is set, the system chooses a single processor at process creation time and assigns that as the process affinity mask, starting with the first processor and then going round-robin across all the processors. For example, on a dual-processor system, the first time you run an image marked as uniprocessor, it is assigned to CPU 0; the second time, CPU 1; the third time, CPU 0; the fourth time, CPU 1; and so on. This flag can be useful as a temporary workaround for programs that have multithreaded synchronization bugs that, as a result of race conditions, surface on multiprocessor systems but that don't occur on uniprocessor systems. (This has actually saved the authors of this book on two different occasions.)

EXPERIMENT: Viewing and Changing Process Affinity

In this experiment, you will modify the affinity settings for a process and see that process affinity is inherited by new processes:

1. Run the Command Prompt (cmd.exe).
2. Run Task Manager or Process Explorer, and find the cmd.exe process in the process list.
3. Right-click the process, and select Affinity. A list of processors should be displayed. For example, on a dual-processor system you will see this:



Processor Affinity

The Processor Affinity setting controls which CPUs the process will be allowed to execute on.

☑ CPU 0　□ CPU 8　□ CPU 16　□ CPU 24
□ CPU 1　□ CPU 9　□ CPU 17　□ CPU 25
□ CPU 2　□ CPU 10　□ CPU 18　□ CPU 26
□ CPU 3　□ CPU 11　□ CPU 19　□ CPU 27
□ CPU 4　□ CPU 12　□ CPU 20　□ CPU 28
□ CPU 5　□ CPU 13　□ CPU 21　□ CPU 29
□ CPU 6　□ CPU 14　□ CPU 22　□ CPU 30
□ CPU 7　□ CPU 15　□ CPU 23　□ CPU 31

4. Select a subset of the available processors on the system, and press OK. The process's threads are now restricted to run on the processors you just selected.

5. Now run Notepad.exe from the Command Prompt (by typing **Notepad.exe**).

6. Go back to Task Manager or Process Explorer and find the new Notepad process. Right-click it, and choose Affinity. You should see the same list of processors you chose for the Command Prompt process. This is because processes inherit their affinity settings from their parent.

EXPERIMENT: Changing the Image Affinity

In this experiment (which requires access to a multiprocessor system), you will change the affinity mask of a program to force it to run on the first processor:

1. Make a copy of Cpustres.exe from the Windows 2000 resource kits. For example, assuming there is a c:\temp folder on your system, from the command prompt, type the following:

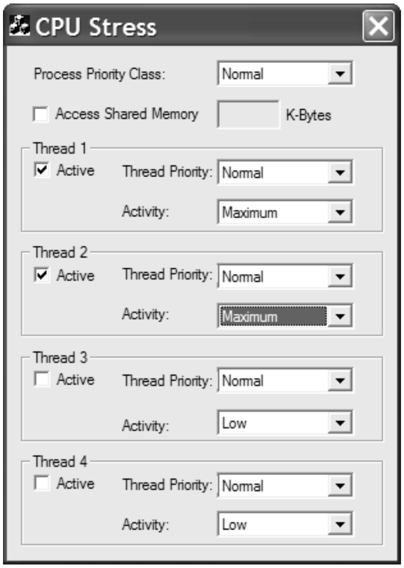
```
copy c:\program files\resource kit\cpustres.exe c:\temp\cpustres.exe
```

2. Set the image affinity mask to force the process's threads to run on CPU 0 by typing the following in the command prompt (assuming that the path to the resource kit tools is in your path):

```
imagecfg -a 1 c:\temp\cpustres.exe
```

3. Now run the modified Cpustres from the c:\temp folder.

4. Enable two worker threads, and set the activity level for both threads to Maximum (not Busy). The Cpustres screen should look like this:



CPU Stress

Process Priority Class: Normal

□ Access Shared Memory K-Bytes

Thread 1
☑ Active Thread Priority: Normal
Activity: Maximum

Thread 2
☑ Active Thread Priority: Normal
Activity: Maximum

Thread 3
□ Active Thread Priority: Normal
Activity: Low

Thread 4
□ Active Thread Priority: Normal
Activity: Low

5. Find the Cpustres process in Process Explorer or Task Manager, right click, and choose Set Affinity. The affinity settings should show the process bound to CPU 0.

6. Examine the systemwide CPU usage by clicking Show, System Information (if running Process Explorer) or by clicking the Performance tab (if running Task Manager). Assuming there are no other compute-bound processes, you should see the total percentage of CPU time consumed, approximately 1/# CPUs (for example, 50% on a dual-CPU system, 25% on a four-CPU system), because the two threads in Cpustres are forced to run on a single processor, leaving the other processor(s) idle.

7. Finally, change the affinity mask of the Cpustres process to permit it to run on all CPUs. Go back and examine the systemwide CPU usage. You should see 100% on a dual-CPU system, 50% on a four-CPU system, and so forth.

Windows won't move a running thread that could run on a different processor from one CPU to a second processor to permit a thread with an affinity for the first processor to run on the first processor. For example, consider this scenario: CPU 0 is running a priority 8 thread that can run on any processor, and CPU 1 is running a priority 4 thread that can run on any processor. A priority 6 thread that can run on only CPU 0 becomes ready. What happens? Windows won't move the priority 8 thread from CPU 0 to CPU 1 (preempting the priority 4 thread) so that the priority 6 thread can run; the priority 6 thread has to wait.

Therefore, changing the affinity mask for a process or a thread can result in threads getting less CPU time than they normally would, as Windows is restricted from running the thread on certain processors. Therefore, setting affinity should be done with extreme care—in most cases, it is optimal to let Windows decide which threads run where.

Ideal and Last Processor

Each thread has two CPU numbers stored in the kernel thread block:

- *Ideal processor,* or the preferred processor that this thread should run on
- *Last processor,* or the processor on which the thread last ran

The ideal processor for a thread is chosen when a thread is created using a seed in the process block. The seed is incremented each time a thread is created so that the ideal processor for each new thread in the process will rotate through the available processors on the system. For example, the first thread in the first process on the system is assigned an ideal processor of 0. The second thread in that process is assigned an ideal processor of 1. However, the next process in the system has its first thread's ideal processor set to 1, the second to 2, and so on. In that way, the threads within each process are spread evenly across the processors.

Note that this assumes the threads within a process are doing an equal amount of work. This is typically not the case in a multithreaded process, which normally has one or more housekeeping threads and then a number of worker threads. Therefore, a multithreaded application that wants to take full advantage of the platform might find it advantageous to specify the ideal processor numbers for its threads by using the *SetThreadIdealProcessor* function.

On hyperthreaded systems, the next ideal processor is the first logical processor on the next physical processor. For example, on a dual-processor hyperthreaded system with four logical processors, if the ideal processor for the first thread is assigned to logical processor 0, the second thread would be assigned to logical processor 2, the third thread to logical processor 1, the fourth thread to logical process 3, and so forth. In this way, the threads are spread evenly across the physical processors.

On NUMA systems, when a process is created, an ideal node for the process is selected. The first process is assigned to node 0, the second process to node 1, and so on. Then, the ideal processors for the threads in the process are chosen from the process's ideal node. The ideal processor for the first thread in a process is assigned to the first processor in the node. As additional threads are created in processes with the same ideal node, the next processor is used for the next thread's ideal processor, and so on.

Multiprocessor Thread-Scheduling Algorithms

Now that we've described the types of multiprocessor systems supported by Windows as well as the thread affinity and ideal processor settings, we're ready to examine how this information is used to determine which threads run where. There are two basic decisions to describe:

- Choosing a processor for a thread that wants to run
- Choosing a thread on a processor that needs something to do

Choosing a Processor for a Thread When There Are Idle Processors

When a thread becomes ready to run, Windows first tries to schedule the thread to run on an idle processor. If there is a choice of idle processors, preference is given first to the thread's ideal processor, then to the thread's previous processor, and then to the currently executing processor (that is, the CPU on which the scheduling code is running).

On Windows 2000, if none of these CPUs are idle, the first idle processor the thread can run on is selected by scanning the idle processor mask from highest to lowest CPU number.

On Windows XP and Windows Server 2003, the idle processor selection is more sophisticated. First, the idle processor set is set to the idle processors that the thread's affinity mask permits it to run on. If the system is NUMA and there are idle CPUs in the node containing the thread's ideal processor, the list of idle processors is reduced to that set. If this eliminates all idle processors, the reduction is not done. Next, if the system is running hyperthreaded processors and there is a physical processor with all logical processors idle, the list of idle processors is reduced to that set. If that results in an empty set of processors, the reduction is not done.

If the current processor (the processor trying to determine what to do with the thread that wants to run) is in the remaining idle processor set, the thread is scheduled on it. If the current processor is not in the remaining set of idle processors, it is a hyperthreaded system, and there is an idle logical processor on the physical processor containing the ideal processor for the thread, the idle processors are reduced to that set. If not, the system checks whether there are any idle logical processors on the physical processor containing the thread's previous processor. If that set is nonzero, the idle processors are reduced to that list.

In the set of idle processors remaining, any CPUs in a sleep state are eliminated from consideration. (Again, this reduction is not performed if that would eliminate all possible processors.) Finally, the lowest numbered CPU in the remaining set is selected as the processor to run the thread on.

Regardless of the Windows version, once a processor has been selected for the thread to run on, that thread is put in the Standby state and the idle processor's PRCB is updated to point to this thread. When the idle loop on that processor runs, it will see that a thread has been selected to run and will dispatch that thread.

Choosing a Processor for a Thread When There Are No Idle Processors

If there are no idle processors when a thread wants to run, Windows compares the priority of the thread running (or the one in the standby state) on the thread's ideal processor to determine whether it should preempt that thread. On Windows 2000, a thread's affinity mask can exclude the ideal processor. (This condition is not allowed as of Windows XP.) If that is the case, Windows 2000 selects the thread's previous processor. If that processor is not in the thread's affinity mask, the highest processor number that the thread can run on is selected.

If the thread's ideal processor already has a thread selected to run next (waiting in the standby state to be scheduled) and that thread's priority is less than the priority of the thread being readied for execution, the new thread preempts that first thread out of the standby state and becomes the next thread for that CPU. If there is already a thread running on that CPU, Windows checks whether the priority of the currently running thread is less than the thread being readied for execution. If so, the currently running thread is marked to be preempted and Windows queues an interprocessor interrupt to the target processor to preempt the currently running thread in favor of this new thread.

Note Windows doesn't look at the priority of the current and next threads on all the CPUs—just on the one CPU selected as just described. If no thread can be preempted on that one CPU, the new thread is put in the ready queue for its priority level, where it awaits its turn to get scheduled. Therefore, Windows does not guarantee to be running all the highest priority threads, but it will always run the highest priority thread.

If the ready thread cannot be run right away, it is moved into the ready state where it awaits its turn to run. Note that in Windows Server 2003, threads are always put on their ideal processor's per-processor ready queues.

Selecting a Thread to Run on a Specific CPU (Windows 2000 and Windows XP)

In several cases (such as when a thread enters a wait state, lowers its priority, changes its affinity, or delays or yields execution), Windows must find a new thread to run on the CPU that the currently executing thread is running on. As described earlier, on a single-processor system, Windows simply picks the first thread in the highest-priority nonempty ready queue. On a multiprocessor system, however, Windows 2000 and Windows XP don't simply pick the first thread in the ready queue. Instead, they look for a thread in the highest-priority nonempty read queue that meets one of the following conditions:

- Ran last on the specified processor
- Has its ideal processor set to the specified processor
- Has been ready to run for longer than 3 clock ticks
- Has a priority greater than or equal to 24

Threads that don't have the specified processor in their hard affinity mask are skipped, obviously. If there are no threads that meet one of these conditions, Windows picks the thread at the head of the ready queue it began searching from.

Why does it matter which processor a thread was last running on? As usual, the answer is speed–giving preference to the last processor a thread executed on maximizes the chances that thread data remains in the secondary cache of the processor in question.

Selecting a Thread to Run on a Specific CPU (Windows Server 2003)

Because each processor in Windows Server 2003 has its own list of threads waiting to run on that processor, when a thread finishes running, the processor can simply check its per-processor ready queue for the next thread to run. If the per-processor ready queues are empty, the idle thread for that processor is scheduled. The idle thread then begins scanning other processor's ready queues for threads it can run. Note that on NUMA systems, the idle thread first looks at processors on its node before looking at other nodes' processors.

Job Objects

A *job object* is a nameable, securable, shareable kernel object that allows control of one or more processes as a group. A job object's basic function is to allow groups of processes to be managed and manipulated as a unit. A process can be a member of only one job object. By default, its association with the job object can't be broken and all processes created by the process and its descendents are associated with the same job object as well. The job object also records basic accounting information for all processes associated with the job and for all processes that were associated with the job but have since terminated. Table 6-20 lists the Windows functions to create and manipulate job objects.

Table 6-19 Windows API Functions for Jobs

Function	Description
CreateJobObject	Creates a job object (with an optional name)
OpenJobObject	Opens an existing job object by name
AssignProcessToJobObject	Adds a process to a job
TerminateJobObject	Terminates all processes in a job
SetInformationJobObject	Sets limits
QueryInformationJobObject	Retrieves information about the job, such as CPU time, page fault count, number of processes, list of process IDs, quotas or limits, and security limits

The following are some of the CPU-related and memory-related limits you can specify for a job:

- **Maximum number of active processes** Limits the number of concurrently existing processes in the job.
- **Jobwide user-mode CPU time limit** Limits the maximum amount of user-mode CPU time that the processes in the job can consume (including processes that have run and exited). Once this limit is reached, by default all the processes in the job will be terminated with an error code and no new processes can be created in the job (unless the limit is reset). The job object is signaled, so any threads waiting for the job will be released. You can change this default behavior with a call to *EndOfJobTimeAction*.

- **Per-process user-mode CPU time limit** Allows each process in the job to accumulate only a fixed maximum amount of user-mode CPU time. When the maximum is reached, the process terminates (with no chance to clean up).
- **Job scheduling class** Sets the length of the time slice (or quantum) for threads in processes in the job. This setting applies only to systems running with long, fixed quantums (the default for Windows Server systems). The value of the job-scheduling class determines the quantum as shown here:

Scheduling Class	Quantum Units
0	6
1	12
2	18
3	24
4	30
5	36
6	42
7	48
8	54
9	Infinite if real-time; 60 otherwise

- **Job processor affinity** Sets the processor affinity mask for each process in the job. (Individual threads can alter their affinity to any subset of the job affinity, but processes can't alter their process affinity setting.)
- **Job process priority class** Sets the priority class for each process in the job. Threads can't increase their priority relative to the class (as they normally can). Attempts to increase thread priority are ignored. (No error is returned on calls to *SetThreadPriority*, but the increase doesn't occur.)
- **Default working set minimum and maximum** Defines the specified working set minimum and maximum for each process in the job. (This setting isn't jobwide—each process has its own working set with the same minimum and maximum values.)
- **Process and job committed virtual memory limit** Defines the maximum amount of virtual address space that can be committed by either a single process or the entire job.

Jobs can also be set to queue an entry to an I/O completion port object, which other threads might be waiting for, with the Windows *GetQueuedCompletionStatus* function.

You can also place security limits on processes in a job. You can set a job so that each process runs under the same jobwide access token. You can then create a job to restrict processes from impersonating or creating processes that have access tokens that contain the local administrator's group. In addition, you can apply security filters so that when threads in processes contained in a job impersonate client threads, certain privileges and security IDs (SIDs) can be eliminated from the impersonation token.

Finally, you can also place user-interface limits on processes in a job. Such limits include being able to restrict processes from opening handles to windows owned by threads outside the job, reading and/or writing to the clipboard, and changing the many user-interface system parameters via the Windows *SystemParametersInfo* function.

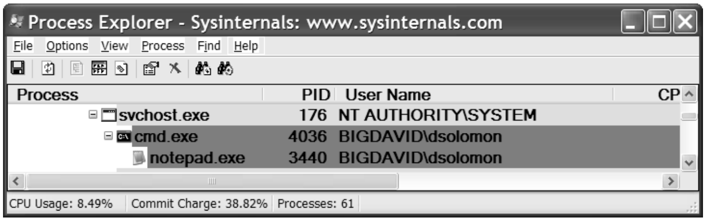
Windows 2000 Datacenter Server has a tool called the Process Control Manager that allows an administrator to define job objects, the various quotas and limits that can be specified for a job, and which processes, if run, should be added to the job. A service component monitors process activity and adds the specified processes to the jobs. Note that this tool is no longer shipped with Windows Server 2003 Datacenter Edition, but will remain on the system if a Windows 2000 Datacenter Server is upgraded to Windows Server 2003 Datacenter Edition.

EXPERIMENT: Viewing the Job Object

You can view named job objects with the Performance tool. (See the Job Object and Job Object Details performance objects.) You can view unnamed jobs with the kernel debugger *!job* or *dt nt!_ejob* commands.

To see whether a process is associated with a job, you can use the kernel debugger *!process* command, or on Windows XP and Windows Server 2003, Process Explorer. Follow these steps to create and view an unnamed job object:

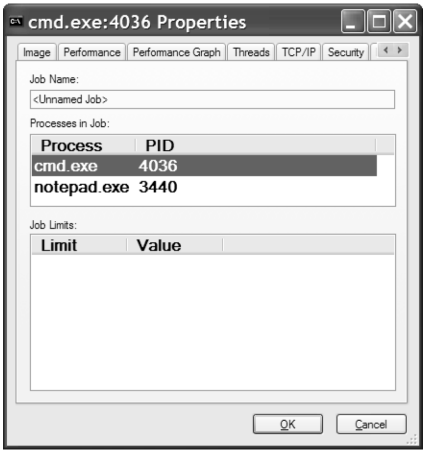
1. From the command prompt, use the *runas* command to create a process running the command prompt (Cmd.exe). For example, type **runas /user:<domain>\<username> cmd**. You'll be prompted for your password. Enter your password, and a command prompt window will appear. The Windows service that executes runas commands creates an unnamed job to contain all processes (so that it can terminate these processes at logoff time).
2. From the command prompt, run Notepad.exe.
3. Then run Process Explorer and notice that the Cmd.exe and Notepad.exe processes are highlighted as part of a job. (You can configure the colors used to highlight processes that are members of a job by clicking Options, Configure Highlighting.) Here is a screen shot showing these two processes:



```
Process                         PID   User Name                    CP
  svchost.exe                   176   NT AUTHORITY\SYSTEM
    cmd.exe                    4036   BIGDAVID\dsolomon
      notepad.exe              3440   BIGDAVID\dsolomon
CPU Usage: 8.49%   Commit Charge: 38.82%   Processes: 61
```

4. Double-click either the Cmd.exe or Notepad.exe process to bring up the process properties. You will see a Job tab on the process properties dialog box.

5. Click the Job tab to view the details about the job. In this case, there are no quotas associated with the job, but there are two member processes:



Job Name:

<Unnamed Job>

Processes in Job:

Process	PID
cmd.exe	4036
notepad.exe	3440

Job Limits:

Limit	Value

6. Now run the kernel debugger on the live system (either WinDbg in local kernel debugging mode or LiveKd if you are on Windows 2000), display the process list with *!process*, and find the recently created process running Cmd.exe. Then display the process block by using *!process <process ID>*, find the address of the job object, and finally display the job object with the *!job* command. Here's some partial debugger output of these commands on a live system:

```
lkd> !process 0 0
**** NT ACTIVE PROCESS DUMP ****
    .
    .
PROCESS 8567b758  SessionId: 0  Cid: 0fc4    Peb: 7ffdf000  ParentCid: 00b0
    DirBase: 1b3fb000  ObjectTable: e18dd7d0  HandleCount:  19.
    Image: cmd.exe

PROCESS 856561a0  SessionId: 0  Cid: 0d70    Peb: 7ffdf000  ParentCid: 0fc4
    DirBase: 2e341000  ObjectTable: e19437c8  HandleCount:  16.
    Image: notepad.exe

lkd> !process 0fc4
Searching for Process with Cid == fc4
PROCESS 8567b758  SessionId: 0  Cid: 0fc4    Peb: 7ffdf000  ParentCid: 00b0
    DirBase: 1b3fb000  ObjectTable: e18dd7d0  HandleCount:  19.
    Image: cmd.exe
    BasePriority                      8
    .
    .
    Job                               85557988

lkd> !job 85557988
Job at 85557988
  TotalPageFaultCount      0
  TotalProcesses           2
  ActiveProcesses          2
  TotalTerminatedProcesses 0
```

```
LimitFlags               0
MinimumWorkingSetSize    0
MaximumWorkingSetSize    0
ActiveProcessLimit       0
PriorityClass            0
UIRestrictionsClass      0
SecurityLimitFlags       0
Token                    00000000
```

7. Finally, use the *dt* command to display the job object and notice the additional fields shown about the job:

```
lkd> dt nt!_ejob 85557988
nt!_EJOB
   +0x000 Event            : _KEVENT
   +0x010 JobLinks         : _LIST_ENTRY [ 0x805455c8 - 0x85797888 ]
   +0x018 ProcessListHead  : _LIST_ENTRY [ 0x8567b8dc - 0x85656324 ]
   +0x020 JobLock          : _ERESOURCE
   +0x058 TotalUserTime    : _LARGE_INTEGER 0x0
   +0x060 TotalKernelTime  : _LARGE_INTEGER 0x0
   +0x068 ThisPeriodTotalUserTime : _LARGE_INTEGER 0x0
   +0x070 ThisPeriodTotalKernelTime : _LARGE_INTEGER 0x0
   +0x078 TotalPageFaultCount : 0
   +0x07c TotalProcesses   : 2
   +0x080 ActiveProcesses  : 2
   +0x084 TotalTerminatedProcesses : 0
   +0x088 PerProcessUserTimeLimit : _LARGE_INTEGER 0x0
   +0x090 PerJobUserTimeLimit : _LARGE_INTEGER 0x0
   +0x098 LimitFlags       : 0
   +0x09c MinimumWorkingSetSize : 0
   +0x0a0 MaximumWorkingSetSize : 0
   +0x0a4 ActiveProcessLimit : 0
   +0x0a8 Affinity         : 0
   +0x0ac PriorityClass    : 0 ''
   +0x0b0 UIRestrictionsClass : 0
   +0x0b4 SecurityLimitFlags : 0
   +0x0b8 Token            : (null)
   +0x0bc Filter           : (null)
   +0x0c0 EndOfJobTimeAction : 0
   +0x0c4 CompletionPort   : 0x8619d8c0
   +0x0c8 CompletionKey    : (null)
   +0x0cc SessionId        : 0
   +0x0d0 SchedulingClass  : 5
   +0x0d8 ReadOperationCount : 0
   +0x0e0 WriteOperationCount : 0
   +0x0e8 OtherOperationCount : 0
   +0x0f0 ReadTransferCount : 0
   +0x0f8 WriteTransferCount : 0
   +0x100 OtherTransferCount : 0
   +0x108 IoInfo           : _IO_COUNTERS
   +0x138 ProcessMemoryLimit : 0
   +0x13c JobMemoryLimit   : 0
   +0x140 PeakProcessMemoryUsed : 0x256
   +0x144 PeakJobMemoryUsed : 0x1f6
   +0x148 CurrentJobMemoryUsed : 0x1f6
   +0x14c MemoryLimitsLock : _FAST_MUTEX
   +0x16c JobSetLinks      : _LIST_ENTRY [ 0x85557af4 - 0x85557af4 ]
   +0x174 MemberLevel      : 0    +0x178 JobFlags         : 0
```

Conclusion

In this chapter, we've examined the structure of processes and threads and jobs, seen how they are created, and looked at how Windows decides which threads should run and for how long.

Many references in this chapter are to topics related to memory management. Because threads run inside processes and processes in large part define an address space, the next logical topic is how Windows performs virtual and physical memory management—the subjects of Chapter 7.

Chapter 7

Memory Management

In this chapter, you'll learn how Microsoft Windows implements virtual memory and how it manages the subset of virtual memory kept in physical memory. We'll also describe the internal structure and components that make up the memory manager, including key data structures and algorithms. Before examining these mechanisms, let's review the basic services provided by the memory manager and key concepts such as reserved memory versus committed memory and shared memory.

Introduction to the Memory Manager

By default, the virtual size of a process on 32-bit Windows is 2 GB. If the image is marked specially as large address space aware, and the system is booted with a special switch (described later in this chapter), a 32-bit process can grow to 3 GB on 32-bit Windows and up to 4 GB on 64-bit Windows. The process virtual address space size on 64-bit Windows is 7152 GB on IA-64 systems and 8192 GB on x64 systems. (This value could be increased in future releases.)

As you saw in Chapter 2 (specifically in Table 2-4), the maximum amount of physical memory supported by Windows ranges from 2 GB to 1024 GB, depending on which version and edition of Windows you are running. Because the virtual address space might be larger or smaller than the physical memory on the machine, the memory manager has two primary tasks:

- Translating, or mapping, a process's virtual address space into physical memory so that when a thread running in the context of that process reads or writes to the virtual address space, the correct physical address is referenced. (The subset of a process's virtual address space that is physically resident is called the *working set*. Working sets are described in more detail later in this chapter.)
- Paging some of the contents of memory to disk when it becomes overcommitted—that is, when running threads or system code try to use more physical memory than is currently available—and bringing the contents back into physical memory when needed.

In addition to providing virtual memory management, the memory manager provides a core set of services on which the various Windows environment subsystems are built. These services include memory mapped files (internally called *section objects*), copy-on-write memory, and support for applications using large, sparse address spaces. In addition, the memory manager provides a way for a process to allocate and use larger amounts of physical memory than can be mapped into the process virtual address space (for example, on 32-bit systems with more than 4 GB of physical memory). This is explained in the section "Address Windowing Extensions" later in this chapter.

Memory Manager Components

The memory manager is part of the Windows executive and therefore exists in the file Ntoskrnl.exe. No parts of the memory manager exist in the HAL. The memory manager consists of the following components:

- A set of executive system services for allocating, deallocating, and managing virtual memory, most of which are exposed through the Windows API or kernel-mode device driver interfaces
- A translation-not-valid and access fault trap handler for resolving hardware-detected memory management exceptions and making virtual pages resident on behalf of a process
- Several key components that run in the context of six different kernel-mode system threads:
    - ❑ The *working set manager* (priority 16), which the balance set manager (a system thread that the kernel creates) calls once per second as well as when free memory falls below a certain threshold, drives the overall memory management policies, such as working set trimming, aging, and modified page writing.
    - ❑ The *process/stack swapper* (priority 23) performs both process and kernel thread stack inswapping and outswapping. The balance set manager and the thread-scheduling code in the kernel awaken this thread when an inswap or outswap operation needs to take place.
    - ❑ The *modified page writer* (priority 17) writes dirty pages on the modified list back to the appropriate paging files. This thread is awakened when the size of the modified list needs to be reduced. (See the section "Modified Page Writer" to find out how you can change this default value.)
    - ❑ The *mapped page writer* (priority 17) writes dirty pages in mapped files to disk. It is awakened when the size of the modified list needs to be reduced or if pages for mapped files have been on the modified list for more than 5 minutes. This second modified page writer thread is necessary because it can generate page faults that result in requests for free pages. If there were no free pages and there was only one modified page writer thread, the system could deadlock waiting for free pages.

- The *dereference segment thread* (priority 18) is responsible for cache reduction as well as page file growth and shrinkage. (For example, if there is no virtual address space for paged pool growth, this thread trims the page cache so that the paged pool used to anchor it can be freed for reuse.)
- The *zero page thread* (priority 0) zeroes out pages on the free list so that a cache of zero pages is available to satisfy future demand-zero page faults. (Memory zeroing in some cases is done by a faster function called *MiZeroInParallel*. See the note in the section “Page List Dynamics.”)

Each of these components is covered in more detail later in the chapter.

Internal Synchronization

Like all other components of the Windows executive, the memory manager is fully reentrant and supports simultaneous execution on multiprocessor systems—that is, it allows two threads to acquire resources in such a way that they don’t corrupt each other’s data. To accomplish the goal of being fully reentrant, the memory manager uses several different internal synchronization mechanisms to control access to its own internal data structures, such as spinlocks and executive resources. (Synchronization objects are discussed in Chapter 3.)

Systemwide resources to which the memory manager must synchronize access include the page frame number (PFN) database (controlled by a spinlock), section objects and the system working set (controlled by pushlocks), and page file creation (controlled by a mutex). In Windows XP and Windows Server 2003, a number of these locks have been either removed completely or optimized, resulting in much less contention. For example, in Windows 2000, spinlocks were used to synchronize changes to system address space and memory commitment; however to improve scalability, these spinlocks have been removed as of Windows XP. Per-process memory management data structures that require synchronization include the working set lock (held while changes are being made to the working set list) and the address space lock (held whenever the address space is being changed). Working set synchronization in Windows 2000 was implemented as a mutex; in Windows XP and later, a pushlock is used, improving parallelism and scalability, because pushlocks support both shared and exclusive access.

Other operations that no longer involve acquiring locks include charging nonpaged and paged pool quotas, charging commitment of pages, and allocating and mapping physical memory allocated through the address windowing extensions (AWE) functions. Also, the lock that synchronizes access to the structures that describe physical memory (the PFN database) is now acquired less and when acquired, held for less time. These changes translate into greater parallelism and scalability on multiprocessor systems because the number of times the memory manager might have to block while another CPU is making a change to a global structure has been reduced or eliminated.

Configuring the Memory Manager

Like most of Windows, the memory manager attempts to automatically provide optimal system performance for varying workloads on systems of varying sizes and types. While there are a limited number of registry values you can add and/or modify under the key HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management to override some of these default performance calculations, in general, the memory manager's default computations will be sufficient for the majority of workloads.

Many of the thresholds and limits that control memory manager policy decisions are computed at system boot time on the basis of memory size and product type. (Windows 2000 Professional and Windows XP Professional and Home editions are optimized for desktop interactive use, and Windows Server systems are optimized for running server applications.) These values are stored in various kernel variables and later used by the memory manager. To find some of these, search for global variables in Ntoskrnl.exe that have names beginning with *Mm* that contain the word "maximum" or "minimum." (See the experiment "Peering into Undocumented Interfaces" in Chapter 2.)

Warning Although you'll find references to some of these variables, you shouldn't change them. Windows has been tested to operate properly with the current possible permutations of these values that can be computed. Changing the value of these kernel variables on a running system can result in unpredictable system behavior, including system hangs or even crashes.

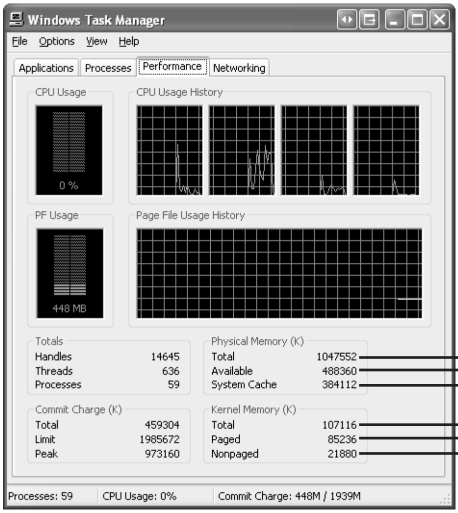
Examining Memory Usage

The Memory and Process performance counter objects provide access to most of the details about system and process memory utilization. Throughout the chapter, we'll include references to specific performance counters that contain information related to the component being described.

Besides the Performance tool, a number of tools in the Windows Support Tools and Windows resource kits display different subsets of memory usage information. We've included relevant examples and experiments throughout the chapter. One word of caution, however—different utilities use varying and sometimes inconsistent or confusing names when displaying memory information. The following experiment illustrates this point. (We'll explain the terms used in this example in subsequent sections.)

EXPERIMENT: Viewing System Memory Information

The Performance tab in the Windows Task Manager, shown in the following screen shot from Windows XP, displays basic system memory information. This information is a subset of the detailed memory information available through the performance counters.



```
Totals                    Physical Memory (K)
Handles        14645      Total          1047552
Threads          636      Available       488360
Processes         59      System Cache    384112

Commit Charge (K)         Kernel Memory (K)
Total         459304      Total           107116
Limit        1985672      Paged            85236
Peak          973160      Nonpaged         21880

Processes: 59    CPU Usage: 0%    Commit Charge: 448M / 1939M
```

Actual physical memory on machine

Total size of standby, free, and zero lists

Standby list + size of system working set

Total of next two values

Paged pool virtual size

Nonpaged pool physical size

Both Pmon.exe (in the Windows Support Tools) and Pstat.exe (in the Platform SDK) display system and process memory information. The annotations in the following output from Pstat explain the information reported. (For an explanation of the commit total and limit, see Table 7-15.)

Total physical memory

Total of standby, free, zero lists

Total of all process working sets (*not* the total process memory usage, which would double count shared pages)

Resident system code

Resident paged pool

```
Memory:  64692K Avail: 3676K TotalWs:  62836K InRam Kernel: 3788K    P: 9604K
Commit: 111848K/85808K  Limit: 182400K Peak: 113732K  Pool N: 2344K    P:14448K
```

System working set size (*not* just the size of the file cache)

Nonpaged pool physical size

Paged pool virtual size

```
User Time    Kernel Time    Ws    Faults  Commit Pri  Hnd  Thd  Pid   Name
                         20032    194378                               File Cache

0:00:00.000  8:03:55.734   16        1      0    0     0    2    0  Idle Process
0:00:00.000  0:01:24.421   32     1601     36    8   140   27    2  System
0:00:00.140  0:00:00.468  200     2008    164   11    35    6   20  smss.exe
0:00:00.640  0:00:02.843  716     8625   1588   13   145    9   30  csrss.exe
```

To see the specific usage of paged and nonpaged pool, use the Poolmon utility, described in the "Monitoring Pool Usage" section.

Finally, the *!vm* command in the kernel debugger shows the basic memory management information available through the memory-related performance counters. This command can be useful if you're looking at a crash dump or hung system. Here's an example of its output:

```
kd> !vm

*** Virtual Memory Usage ***
        Physical Memory:    32620   (  130480 Kb)
        Page File: \??\C:\pagefile.sys
           Current:    204800Kb Free Space:    101052Kb
           Minimum:    204800Kb Maximum:       204800Kb
        Available Pages:     3604   (   14416 Kb)
        ResAvail Pages:     24004   (   96016 Kb)
        Modified Pages:       768   (    3072 Kb)
        NonPagedPool Usage:  1436   (    5744 Kb)
        NonPagedPool Max:   12940   (   51760 Kb)
        PagedPool 0 Usage:   6817   (   27268 Kb)
        PagedPool 1 Usage:    982   (    3928 Kb)
        PagedPool 2 Usage:    984   (    3936 Kb)
        PagedPool Usage:     8783   (   35132 Kb)
        PagedPool Maximum:  26624   (  106496 Kb)
        Shared Commit:       1361   (    5444 Kb)
        Special Pool:           0   (       0 Kb)
        Free System PTEs:  189291   (  757164 Kb)
        Shared Process:      3165   (   12660 Kb)
        PagedPool Commit:    8783   (   35132 Kb)
        Driver Commit:       1098   (    4392 Kb)
        Committed pages:    45113   (  180452 Kb)
        Commit limit:       79556   (  318224 Kb)

        Total Private:      30536   (  122144 Kb)
            IEXPLORE.EXE       3028 (   12112 Kb)
            svchost.exe        2128 (    8512 Kb)
            WINWORD.EXE        1971 (    7884 Kb)
            POWERPNT.EXE       1905 (    7620 Kb)
            Acrobat.exe        1761 (    7044 Kb)
            winlogon.exe       1361 (    5444 Kb)
            explorer.exe       1300 (    5200 Kb)
            livekd.exe         1015 (    4060 Kb)
            hh.exe              960 (    3840 Kb)
```

EXPERIMENT: Accounting for Physical Memory Use

By combining information available from performance counters with output from kernel debugger commands, you can come close to accounting for physical memory usage on a machine running Windows. To examine the memory usage information available through performance counters, run the Performance tool and add the counters to view the following information. (You'll see the results more easily if you change the vertical scale maximum on the graph to 1000.)

- **Total process working set size** To view this information, select the Process performance object and the Working Set counter for the _Total process instance. This number will be larger than the actual total process memory utilization because shared pages are counted in each process working set. To get a more accurate picture of process memory utilization, subtract free memory (available bytes), operating system memory used (nonpaged pool, resident paged pool, and resident operating system and driver code), and the size of the modified list from the total physical memory on the machine. What you're left with is the memory being used by processes. Comparing this value against the total process working set size as reported by the Performance tool gives you some indication of the amount of sharing occurring between processes. Although examining process physical memory usage is interesting, of more concern is the private committed virtual memory usage by processes, because memory leaks show up as an increasing private virtual size, not an increasing working set size. At some point, the memory manager will stop the process from growing in physical size, though it can continue to grow in its virtual size until the systemwide commit limit—the maximum amount of private committed virtual memory available on the system—is reached (or less if the process is a member of a job with a job-wide or process virtual memory limit). For more information, see the section "Page Files."
- **Total system working set size** To view this information, select the Memory processor object and the Cache Bytes counter. As explained in the section "System Working Set," the total system working set size includes more than just the cache size—it includes the subset of paged pool, pageable operating system code, and pageable driver code that is resident and in the system working set.
- **Size of nonpaged pool** View this information by adding the Memory: Pool Nonpaged Bytes counter.
- **Size of the free, zero, and standby lists** View the sizes of these lists by adding the Memory: Available Bytes counter. (Use the *!memusage* kernel debugger command to get the size of each of the lists separately.)

Your graph or report now contains a representation of all of physical memory except for two components that you can't obtain from a performance counter:

- Nonpaged operating system and driver code
- The modified and modified-no-write paging lists

Although you can easily obtain the size of both the modified and modified-no-write lists by using the kernel debugger *!memusage* command, there's no easy way to get the size of the nonpaged operating system and driver code.

Services the Memory Manager Provides

The memory manager provides a set of system services to allocate and free virtual memory, share memory between processes, map files into memory, flush virtual pages to disk, retrieve information about a range of virtual pages, change the protection of virtual pages, and lock the virtual pages into memory.

Like other Windows executive services, the memory management services allow their caller to supply a process handle, indicating the particular process whose virtual memory is to be manipulated. The caller can thus manipulate either its own memory or (with the proper permissions) the memory of another process. For example, if a process creates a child process, by default it has the right to manipulate the child process's virtual memory. Thereafter, the parent process can allocate, deallocate, read, and write memory on behalf of the child process by calling virtual memory services and passing a handle to the child process as an argument. This feature is used by subsystems to manage the memory of their client processes, and it is also key for implementing debuggers because debuggers must be able to read and write to the memory of the process being debugged.

Most of these services are exposed through the Windows API. The Windows API has three groups of functions for managing memory in applications: page granularity virtual memory functions (*Virtualxxx*), memory-mapped file functions (*CreateFileMapping*, *MapViewOfFile*), and heap functions (*Heapxxx* and the older interfaces *Localxxx* and *Globalxxx*). (We'll describe the heap manager later in this section.)

The memory manager also provides a number of services, such as allocating and deallocating physical memory and locking pages in physical memory for direct memory access (DMA) transfers, to other kernel-mode components inside the executive as well as to device drivers. These functions begin with the prefix *Mm*. In addition, though not strictly part of the memory manager, executive support routines that begin with *Ex* are used to allocate and deallocate from the system heaps (paged and nonpaged pool) as well as to manipulate look-aside lists. We'll touch on these topics later in this chapter, in the section "System Memory Pools."

Although we'll be referring to Windows functions and kernel-mode memory management and memory allocation routines provided for device drivers, we won't cover the interface and programming details but rather the internal operations of these functions. Refer to the Platform Software Development Kit (SDK) and Device Driver Kit (DDK) documentation on MSDN for a complete description of the available functions and their interfaces.

Large and Small Pages

The virtual address space is divided into units called pages. That is because the hardware memory management unit translates virtual to physical addresses at the granularity of a page. Hence, a page is the smallest unit of protection at the hardware level. (The various page protection options are described in the section "Protecting Memory.") There are two page sizes: small and large. The actual sizes vary based on hardware architecture, and they are listed in Table 7-1.

Table 7-1 Page Sizes

Architecture	Small Page Size	Large Page Size
x86	4 KB	4 MB (2 MB on PAE systems)
x64	4 KB	2 MB
IA64	8 KB	16 MB

The advantage of large pages is speed of address translation for references to other data within the large page. This advantage exists because the first reference to any byte within a large page will cause the hardware's translation look-aside buffer (or TLB, which is described in the section "Translation Look-Aside Buffer") to have in its cache the information necessary to translate references to any other byte within the large page. If small pages are used, more TLB entries are needed for the same range of virtual addresses, thus increasing recycling of entries as new virtual addresses require translation. This, in turn, means having to go back to the page table structures when references are made to virtual addresses outside the scope of a small page whose translation has been cached. The TLB is a very small cache, and thus large pages make better use of this limited resource.

To take advantage of large pages, on systems considered to have enough memory (see Table 7-2 for the minimums), Windows maps with large pages the core operating system images (Ntoskrnl.exe and Hal.dll) as well as core operating system data (such as the initial part of nonpaged pool and the data structures that describe the state of each physical memory page). Windows also automatically maps I/O space requests (calls by device drivers to *MmMapIoSpace*) with large pages if the request is of satisfactory large page length and alignment. Lastly, Windows also allows applications to map their images, private memory and pagefile-backed sections with large pages. (See the MEM_LARGE_PAGE flag on the *VirtualAlloc* function.) You can also specify other device drivers to be mapped with large pages by adding a multistring registry value HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management\LargePageDrivers and specifying the names of the drivers as separately null terminated strings.

Table 7-2 Large Page Minimums

Operating System	Minimum Memory to Use Large Pages
Windows 2000	>127 MB
Windows XP, Windows Server 2003	>255 MB

One side-effect of large pages is that because each large page must be mapped with a single protection (because hardware memory protection is on a per-page basis), if a large page contains both read-only code and read/write data, the page must be marked as read/write, which means that the code will be writable. This means device drivers or other kernel mode code could, as a result of a bug, modify what is supposed to be read-only operating system or driver code without causing a memory access violation. However, if small pages are used to map the kernel, the read-only portions of NTOSKRNL.EXE and HAL.DLL will be mapped as read-only pages. Although this reduces efficiency of address translation, if a device driver (or other kernel mode code) attempts to modify a read-only part of the operating system, the system will crash immediately with the finger pointing at the offending instruction, as opposed to allowing the corruption to occur and later the system crashing (in a harder-to-diagnose way) when some other component trips over their corrupted data. If you suspect you are experiencing kernel code corruptions, enable Driver Verifier (described later in this chapter)—this will disable the use of large pages.

Reserving and Committing Pages

Pages in a process address space are free, reserved, or committed. Applications can first *reserve* address space and then *commit* pages in that address space. Or they can reserve and commit in the same function call. These services are exposed through the Windows *VirtualAlloc* and *VirtualAllocEx* functions.

Reserved address space is simply a way for a thread to reserve a range of virtual addresses for future use. Attempting to access reserved memory results in an access violation because the page isn't mapped to any storage that can resolve the reference.

Committed pages are pages that, when accessed, ultimately translate to valid pages in physical memory. Committed pages are either private and not shareable or mapped to a view of a section (which might or might not be mapped by other processes). Sections are described in two upcoming sections: "Shared Memory and Mapped Files" and "Section Objects."

If the pages are private to the process and have never been accessed before, they are created at the time of first access as zero-initialized pages (or *demand zero*). Private committed pages can later be automatically written to the paging file by the operating system if memory demands dictate. Committed pages that are private are inaccessible to any other process unless they're accessed using cross-process memory functions, such as *ReadProcessMemory* or *WriteProcessMemory*. If committed pages are mapped to a portion of a mapped file, they might need to be brought in from disk when accessed unless they've already been read earlier, either by the process accessing the page or by another process that had the same file mapped and had previously accessed the page. (See the section "Shared Memory and Mapped Files" later in this chapter.)

Pages are written to disk through normal modified page writing as pages are moved from the process working set to the modified list and ultimately to disk. (Working sets and the modified list are explained later in this chapter.) Mapped file pages can also be written back to disk as a result of an explicit call to *FlushViewOfFile*.

You can decommit pages and/or release address space with the *VirtualFree* or *VirtualFreeEx* function. The difference between decommittal and release is similar to the difference between reservation and committal—decommitted memory is still reserved, but released memory is neither committed nor reserved. (It's freed.)

Using the two-step process of reserving and committing memory can reduce memory usage by deferring committing pages until needed but keeping the convenience of virtual contiguity.

Reserving memory is a relatively fast and inexpensive operation under Windows because it doesn't consume any committed pages (a precious system resource) or process page file quota (a limit on the number of committed pages a process can consume—not necessarily page file space). All that needs to be updated or constructed is the relatively small internal data structures that represent the state of the process address space. (We'll explain these data structures, called *virtual address descriptors*, or VADs, later in the chapter.)

Reserving and then committing memory is useful for applications that need a potentially large virtually contiguous memory buffer; rather than committing pages for the entire region, the address space can be reserved and then committed later when needed. A utilization of this technique in the operating system is the user-mode stack for each thread. When a thread is created, a stack is reserved. (1 MB is the default; you can override this size with the *CreateThread* function call or on an imagewide basis by using the /STACK linker flag.) By default, the initial page in the stack is committed and the next page is marked as a guard page, which isn't committed, that traps references beyond the end of the committed portion of the stack and expands it.

Locking Memory

In general, it's better to let the memory manager decide which pages remain in physical memory. However, there might be special circumstances where this is necessary. Pages can be locked in memory in two ways:

- Windows applications can call the *VirtualLock* function to lock pages in their process working set. The number of pages a process can lock can't exceed its minimum working set size minus eight pages. Therefore, if a process needs to lock more pages, it can increase its working set minimum with the *SetProcessWorkingSetSize* function (referred to in the section "Working Set Management").
- Device drivers can call the kernel-mode functions *MmProbeAndLockPages*, *MmLockPagableCodeSection*, *MmLockPagableDataSection*, or *MmLockPagableSectionByHandle*. Pages locked using this mechanism remain in memory until explicitly unlocked. Although no quota is imposed on the number of pages a driver can lock in memory, a driver can't lock more pages than the resident available page count will allow.

Allocation Granularity

Windows aligns each region of reserved process address space to begin on an integral boundary defined by the value of the system *allocation granularity*, which can be retrieved from the Windows *GetSystemInfo* function. Currently, this value is 64 KB. This size was chosen so that if support were added for future processors with large page sizes (for example, up to 64 KB) or virtually indexed caches that require systemwide physical-to-virtual page alignment, the risk of requiring changes to applications that made assumptions about allocation alignment would be reduced. (Windows kernel-mode code isn't subject to the same restrictions; it can reserve memory on a single-page granularity.)

Finally, when a region of address space is reserved, Windows ensures that the size and base of the region is a multiple of the system page size, whatever that might be. For example, because x86 systems use 4-KB pages, if you tried to reserve a region of memory 18 KB in size, the actual amount reserved on an x86 system would be 20 KB. If you specified a base address of 3 KB for an 18-KB region, the actual amount reserved would be 24 KB.

Shared Memory and Mapped Files

As is true with most modern operating systems, Windows provides a mechanism to share memory among processes and the operating system. *Shared memory* can be defined as memory that is visible to more than one process or that is present in more than one process virtual address space. For example, if two processes use the same DLL, it would make sense to load the referenced code pages for that DLL into physical memory only once and share those pages between all processes that map the DLL, as illustrated in Figure 7-1.

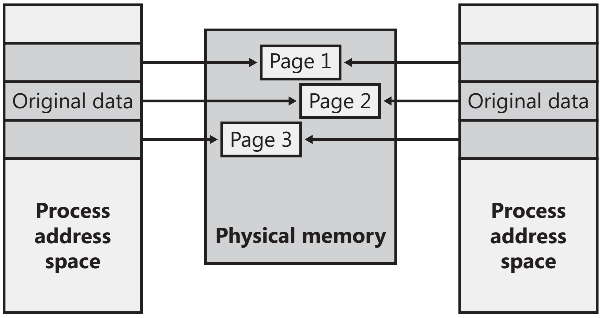


Figure 7-1 Sharing memory between processes

Each process would still maintain its private memory areas in which to store private data, but the program instructions and unmodified data pages could be shared without harm. As we'll explain later, this kind of sharing happens automatically because the code pages in executable images are mapped as execute-only and writable pages are mapped copy-on-write. (See the section "Copy-on-Write" for more information.)

The underlying primitives in the memory manager used to implement shared memory are called *section objects*, which are called *file mapping objects* in the Windows API. The internal structure and implementation of section objects are described in the section "Section Objects" later in this chapter.

This fundamental primitive in the memory manager is used to map virtual addresses, whether in main memory, in the page file, or in some other file that an application wants to access as if it were in memory. A section can be opened by one process or by many; in other words, section objects don't necessarily equate to shared memory.

A section object can be connected to an open file on disk (called a mapped file) or to committed memory (to provide shared memory). Sections mapped to committed memory are called *page file backed sections* because the pages are written to the paging file if memory demands dictate. (Because Windows can run with no paging file, page file backed sections might in fact be “backed” only by physical memory.) As with any other page that is made visible to user mode (such as private committed pages), shared committed pages are always zero-filled when they are first accessed.

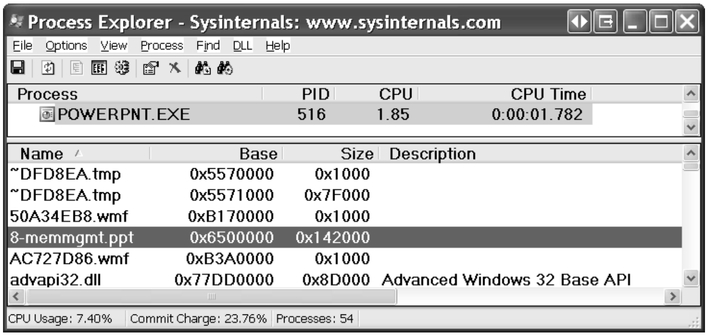
To create a section object, call the Windows *CreateFileMapping* function, specifying the file handle to map it to (or INVALID_HANDLE_VALUE for a page file backed section), and optionally a name and security descriptor. If the section has a name, other processes can open it with *OpenFileMapping*. Or you can grant access to section objects through handle inheritance (by specifying that the handle be inheritable when opening or creating the handle) or handle duplication (by using *DuplicateHandle*). Device drivers can also manipulate section objects with the *ZwOpenSection*, *ZwMapViewOfSection*, and *ZwUnmapViewOfSection* functions.

A section object can refer to files that are much larger than can fit in the address space of a process. (If the paging file backs a section object, sufficient space must exist in the paging file to contain it.) To access a very large section object, a process can map only the portion of the section object that it requires (called a *view* of the section) by calling the *MapViewOfFile* function and then specifying the range to map. Mapping views permits processes to conserve address space because only the views of the section object needed at the time must be mapped into memory.

Windows applications can use mapped files to conveniently perform I/O to files by simply making them appear in their address space. User applications aren’t the only consumers of section objects: the image loader uses section objects to map executable images, DLLs, and device drivers into memory, and the cache manager uses them to access data in cached files. (For information on how the cache manager integrates with the memory manager, see Chapter 11.) How shared memory sections are implemented, both in terms of address translation and the internal data structures, is explained later in this chapter.

EXPERIMENT: Viewing Memory Mapped Files

You can list the memory mapped files in a process by using Process Explorer from Sysinternals. To view the memory mapped files by using Process Explorer, configure the lower pane to show the DLL view. (Click on View, Lower Pane View, DLLs.) Note that this is more than just a list of DLLs—it represents all memory mapped files in the process address space. Some of these are DLLs, one is the image file (EXE) being run, and additional entries might represent memory mapped data files. For example, the following display from Process Explorer shows a Microsoft PowerPoint process that has memory mapped the PowerPoint file being edited into its address space:


Process Explorer - Sysinternals: www.sysinternals.com
File Options View Process Find DLL Help

Process	PID	CPU	CPU Time
POWERPNT.EXE	516	1.85	0:00:01.782

Name	Base	Size	Description
~DFD8EA.tmp	0x5570000	0x1000	
~DFD8EA.tmp	0x5571000	0x7F000	
50A34EB8.wmf	0xB170000	0x1000	
8-memmgmt.ppt	0x6500000	0x142000	
AC727D86.wmf	0xB3A0000	0x1000	
advapi32.dll	0x77DD0000	0x8D000	Advanced Windows 32 Base API

CPU Usage: 7.40% Commit Charge: 23.76% Processes: 54

You can also search for memory mapped files by clicking on Find, DLL. This can be useful when trying to determine which process(es) are using a DLL that you are trying to replace.

Finally, comparing the list of DLLs loaded in a process with another instance of the same program running on another system might help point to DLL configuration issues, such as the wrong version of a DLL getting loaded in a process. This problem is known affectionately as "DLL hell."

Protecting Memory

As explained in Chapter 1, Windows provides memory protection so that no user process can inadvertently or deliberately corrupt the address space of another process or the operating system itself. Windows provides this protection in four primary ways.

First, all systemwide data structures and memory pools used by kernel-mode system components can be accessed only while in kernel mode—user-mode threads can't access these pages. If they attempt to do so, the hardware generates a fault, which in turn the memory manager reports to the thread as an access violation.

Note In contrast, Microsoft Windows 95, Microsoft Windows 98, and Microsoft Windows Millennium Edition have some pages in system address space that are writable from user mode, thus allowing an errant application to corrupt key system data structures and crash the system.

Second, each process has a separate, private address space, protected from being accessed by any thread belonging to another process. The only exceptions are if the process decides to share pages with other processes or if another process has virtual memory read or write access to the process object and thus can use the *ReadProcessMemory* or *WriteProcessMemory* functions. Each time a thread references an address, the virtual memory hardware, in concert with the memory manager, intervenes and translates the virtual address into a physical one. By controlling how virtual addresses are translated, Windows can ensure that threads running in one process don't inappropriately access a page belonging to another process.

Third, in addition to the implicit protection virtual-to-physical address translation offers, all processors supported by Windows provide some form of hardware-controlled memory protection (such as read/write, read-only, and so on); the exact details of such protection vary according to the processor. For example, code pages in the address space of a process are marked read-only and are thus protected from modification by user threads.

Table 7-3 lists the memory protection options defined in the Windows API. (See the *VirtualProtect*, *VirtualProtectEx*, *VirtualQuery*, and *VirtualQueryEx* functions.)

Table 7-3 Memory Protection Options Defined in the Windows API

Attribute	Description
PAGE_NOACCESS	Any attempt to read from, write to, or execute code in this region causes an access violation.
PAGE_READONLY	Any attempt to write to (and on processors with no execute support, execute code in) memory causes an access violation, but reads are permitted.
PAGE_READWRITE	The page is readable and writable, but not executable.
PAGE_EXECUTE*	Any attempt to write to code in memory in this region causes an access violation, but execution (and read on all existing processors) is permitted.
PAGE_EXECUTE_READ*	Any attempt to write to code in memory in this region causes an access violation, but executes and reads are permitted.
PAGE_EXECUTE_READWRITE*	The page is readable, writable, and executable—no action will cause an access violation.
PAGE_WRITECOPY	Any attempt to write to memory in this region causes the system to give the process a private copy of the page. On processors with no execute support, attempts to execute code in memory in this region cause an access violation.
PAGE_EXECUTE_WRITECOPY	Any attempt to write to memory in this region causes the system to give the process a private copy of the page. Reading and executing code in this region is permitted. (No copy is made in this case.)
PAGE_GUARD	Any attempt to read from or write to a guard page raises an EXCEPTION_GUARD_PAGE exception and turns off the guard page status. Guard pages thus act as a one-shot alarm. Note that this flag can be specified with any of the page protections listed in this table except PAGE_NOACCESS.
PAGE_NOCACHE	Use physical memory that is not cached. This is not recommended for general usage. It is useful for device drivers—for example, mapping a video frame buffer with no caching.
PAGE_WRITECOMBINE	Enables write-combined memory accesses. When enabled, the processor might cache memory write requests to optimize performance. For example, if multiple writes are made to the same address, only the most recent write might occur.

* No execute protection is supported by Windows XP Service Pack 2 and Windows Server 2003 Service Pack 1 and later on processors that have the necessary hardware support (for example, the x64, IA-64, and future x86 processors). On earlier versions of Windows and on processors that do not support no execute protection, all page permissions allow execution. The next section contains a more complete description of no execute protection.

And finally, shared memory section objects have standard Windows access-control lists (ACLs) that are checked when processes attempt to open them, thus limiting access of shared memory to those processes with the proper rights. Security also comes into play when a thread creates a section to contain a mapped file. To create the section, the thread must have at least read access to the underlying file object or the operation will fail.

Once a thread has successfully opened a handle to a section, its actions are still subject to the memory manager and the hardware-based page protections described earlier. A thread can change the page-level protection on virtual pages in a section if the change doesn't violate the permissions in the ACL for that section object. For example, the memory manager allows a thread to change the pages of a read-only section to have copy-on-write access but not to have read/write access. The copy-on-write access is permitted because it has no effect on other processes sharing the data.

These four primary memory protection mechanisms are part of the reason that Windows is a robust, reliable operating system that is impervious to and resilient to application errors.

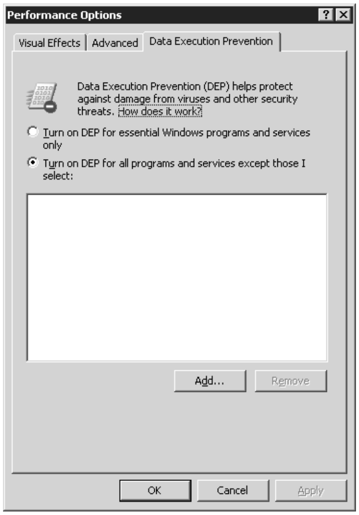
No Execute Page Protection

Although the Windows memory management API has always had page protection bits defined in the programming interface that allow specification of whether or not pages can contain executable code, it is only as of Windows XP Service Pack 2 and Windows Server 2003 Service Pack 1 that this capability is supported on processors that have hardware "no execute" protection, including all AMD64 processors (including AMD Athlon64 and AMD Opteron) and certain exclusively 32-bit AMD processors (selected AMD Sempron processors–details are in AMD's product literature), Intel IA-64, and Intel Pentium 4 and Xeon processors with Intel Extended Memory 64 Technology (EM64T).

No execute page protection (also referred to as data execution prevention, or DEP) means an attempt to transfer control to an instruction in a page marked as "no execute" will generate an access fault. This can prevent certain types of viruses from exploiting bugs in the system that permit the execution of code placed in a data page. If an attempt is made in kernel mode to execute code in a page marked as no execute, the system will crash with the ATTEMPTED_EXECUTE_OF_NOEXECUTE_MEMORY bugcheck code. If this occurs in user mode, a STATUS_ACCESS_VIOLATION (0xc0000005) exception is delivered to the thread attempting the illegal reference. If a process allocates memory that needs to be executable, it must explicitly mark such pages by specifying the PAGE_EXECUTE, PAGE_EXECUTE_READ, PAGE_EXECUTE_READWRITE, or PAGE_EXECUTE_WRITECOPY flags on the page granularity memory allocation functions.

On 64-bit versions of Windows, execution protection is always applied to all 64-bit programs and device drivers and cannot be disabled. Execution protection for 32-bit programs depends on system configuration settings, described shortly. On 64-bit Windows, execution protection is applied to thread stacks (both user and kernel mode), user mode pages not specifically marked as executable, kernel paged pool, and kernel session pool (for a description of kernel memory pools, see the section "System Memory Pools"). However, on 32-bit Windows, execution protection is only applied to thread stacks and user mode pages, not to paged pool and session pool. Also, when execution protection is enabled on 32-bit Windows, the system automatically boots in PAE mode (automatically selecting the PAE kernel, \Windows\System32\Ntkrnlpa.exe). For a description of PAE, see the section "Physical Address Extension (PAE)."

The application of execution protection for 32-bit programs depends on the Boot.ini /NOEXECUTE= switch. The settings can be changed by going to the Data Execution Prevention tab under My Computer, Properties, Advanced, Performance Settings. (See Figure 7-2.) When you configure no execute protection with the DEP settings dialog box, Boot.ini is modified to add the appropriate /NOEXECUTE Boot.ini switch. For a list of the variations of the switch and how they correspond to the DEP settings tab, see Table 7-4. 32-bit applications that are excluded from execution protection are listed as registry values under the key HKLM\Software\Microsoft\Windows NT\CurrentVersion\AppCompatFlags\Layers with the value name being the full path of the executable and the data set to "DisableNXShowUI".



Data Execution Prevention (DEP) helps protect against damage from viruses and other security threats. How does it work?

○ Turn on DEP for essential Windows programs and services only

● Turn on DEP for all programs and services except those I select:

Figure 7-2 Data Execution Protection settings

On Windows XP (both 64-bit and 32-bit versions), execution protection for 32-bit applications is configured by default to apply only to core Windows operating system executables (/NOEXECUTE=OPTIN) so as not to break 32-bit applications that might rely on being able to execute code in pages not specifically marked as executable. On Windows Server 2003 systems, execution protection for 32-bit applications is configured by default to apply to all 32-bit programs (/NOEXECUTE=OPTOUT).

Note To obtain a complete list of which programs are protected, install the Windows Application Compatibility Toolkit (downloadable from *www.microsoft.com*) and run the Compatibility Administrator Tool. Click on System Database, Applications, and Windows Components and, on the right-hand pane, the list of protected executables will be shown.

Table 7-4 **Boot.ini/NOEXECUTE Switch**

Boot.ini Switch	Option in DEP Settings Dialog Box	Meaning
/NOEXECUTE=OPTIN	Turn on DEP for necessary Windows programs and services only.	Enables DEP for core Windows system images.
/NOEXECUTE=OPTOUT	Turn on DEP for all programs and services except those that I select.	Enables DEP for all executables except those specified.
/NOEXECUTE=ALWAYSON	(No GUI interface to select this option)	Enables DEP for all components with no ability to exclude certain applications.
/NOEXECUTE=ALWAYSOFF	(No GUI interface to select this option)	Disables DEP (not recommended).

Software Data Execution Prevention

Because most processors running Windows these days do not support hardware "no execute" protection, Windows XP Service Pack 2 and Windows Server 2003 Service Pack 1 and later support limited software data execution prevention (DEP). One aspect of software DEP reduces exploits of the exception handling mechanism in Windows. (See Chapter 3 for a description of structured exception handling.) If the program's image files are built with safe structured exception handling (a new feature in the Microsoft Visual C++ 2003 compiler), before an exception is dispatched, the system verifies that the exception handler is registered in the function table located within the image file. If the program's image files are not built with safe structured exception handling, software DEP ensures that before an exception is dispatched, the exception handler is located within a memory region marked as executable.

Copy-on-Write

Copy-on-write page protection is an optimization the memory manager uses to conserve physical memory. When a process maps a copy-on-write view of a section object that contains read/write pages, instead of making a process private copy at the time the view is mapped (as the Hewlett Packard OpenVMS operating system does), the memory manager defers making a copy of the pages until the page is written to. All modern UNIX systems use this technique as well. For example, as shown in Figure 7-3, two processes are sharing three pages, each marked copy-on-write, but neither of the two processes has attempted to modify any data on the pages.

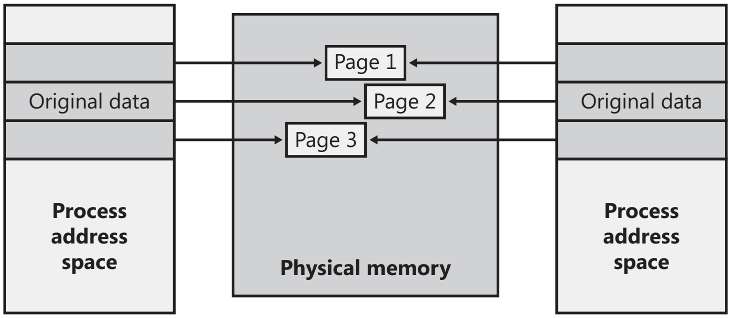


Figure 7-3 The "before" of copy-on-write

If a thread in either process writes to a page, a memory management fault is generated. The memory manager sees that the write is to a copy-on-write page, so instead of reporting the fault as an access violation, it allocates a new read/write page in physical memory, copies the contents of the original page to the new page, updates the corresponding page-mapping information (explained later in this chapter) in this process to point to the new location, and dismisses the exception, thus causing the instruction that generated the fault to be reexecuted. This time, the write operation succeeds, but as shown in Figure 7-4, the newly copied page is now private to the process that did the writing and isn't visible to the other processes still sharing the copy-on-write page. Each new process that writes to that same shared page will also get its own private copy.

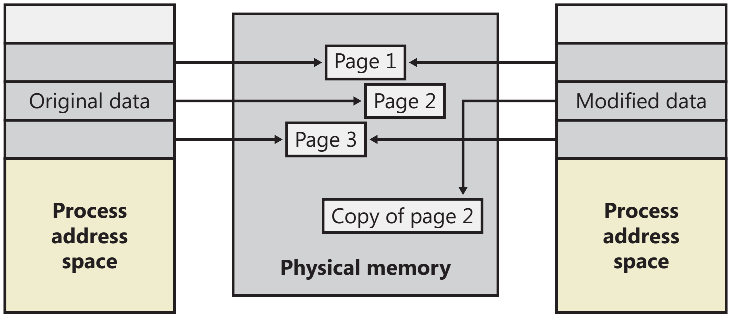


Figure 7-4 The "after" of copy-on-write

One application of copy-on-write is to implement breakpoint support in debuggers. For example, by default, code pages start out as execute-only. If a programmer sets a breakpoint while debugging a program, however, the debugger must add a breakpoint instruction to the code. It does this by first changing the protection on the page to PAGE_EXECUTE_READWRITE and then changing the instruction stream. Because the code page is part of a mapped section, the memory manager creates a private copy for the process with the breakpoint set, while other processes continue using the unmodified code page.

Copy-on-write is one example of an evaluation technique known as *lazy evaluation* that the memory manager uses as often as possible. Lazy-evaluation algorithms avoid performing an expensive operation until absolutely required—if the operation is never required, no time is wasted on it.

The POSIX subsystem takes advantage of copy-on-write to implement the *fork* function. Typically, when a UNIX application calls the *fork* function to create another process, the first thing that the new process does is call the *exec* function to reinitialize the address space with an executable program. Instead of copying the entire address space on *fork*, the new process shares the pages in the parent process by marking them copy-on-write. If the child writes to the data, a process private copy is made. If not, the two processes continue sharing and no copying takes place. One way or the other, the memory manager copies only the pages the process tries to write to rather than the entire address space.

To examine the rate of copy-on-write faults, see the performance counter Memory: Write Copies/Sec.

Heap Manager

Many applications allocate smaller blocks than the 64-KB minimum allocation granularity possible using page granularity functions such as *VirtualAlloc*. Allocating such a large area for relatively small allocations is not optimal from the memory usage and performance standpoint. To address this need, Windows provides a component called the *heap manager*, which manages allocations inside larger memory areas reserved using the page granularity memory allocation functions. The allocation granularity in the heap manager is relatively small: 8 bytes on 32-bit systems and 16 bytes on 64-bit systems. The heap manager has been designed to optimize memory usage and performance in the case of these smaller allocations.

The heap manager exists in two places: Ntdll.dll and Ntoskrnl.exe. The subsystem APIs (such as the Windows heap APIs) call the functions in Ntdll, and various executive components and device drivers call the functions in Ntoskrnl. Its native interfaces (prefixed with *Rtl*) are available only for use in internal Windows components or kernel mode device drivers. The documented Windows API interface to the heap (prefixed with *Heap*) are thin functions that call the native functions in Ntdll.dll. In addition, legacy APIs (prefixed with either *Local* or *Global*) are provided to support older Windows applications. The most common Windows heap functions are:

- *HeapCreate* or *HeapDestroy*—Creates or deletes, respectively, a heap. The initial reserved and committed size can be specified at creation.
- *HeapAlloc*—Allocates a heap block.
- *HeapFree*—Frees a block previously allocated with *HeapAlloc*.
- *HeapReAlloc*—Changes the size of an existing allocation (grows or shrinks an existing block).
- *HeapLock* or *HeapUnlock*—Controls mutual exclusion to the heap operations.
- *HeapWalk*—Enumerates the entries and regions in a heap.

Types of Heaps

Each process has at least one heap: the default process heap. The default heap is created at process startup and is never deleted during the process's lifetime. It defaults to 1 MB in size, but it can be bigger by specifying a starting size in the image file by using the /HEAP linker flag. This size is just the initial reserve, however–it will expand automatically as needed. (You can also specify the initial committed size in the image file.)

The default heap can be explicitly used by a program or implicitly used by some Windows internal functions. An application can query the default process heap by making a call to the Windows function *GetProcessHeap*. Processes can also create additional private heaps with the *HeapCreate* function. When a process no longer needs a private heap, it can recover the virtual address space by calling *HeapDestroy*. An array with all heaps is maintained in each process, and a thread can query them with the Windows function *GetProcessHeaps*.

A heap can manage allocations either in large memory regions reserved from the memory manager via *VirtualAlloc* or from memory mapped file objects mapped in the process address space. The latter approach is rarely used in practice, but it's suitable for scenarios where the content of the blocks need to be shared between two processes or between a kernel mode and a user mode component. If a heap is built on top of a memory mapped file region, certain constraints apply with respect to the component that can call heap functions. First, the internal heap structures use pointers, and therefore do not allow relocation to different addresses. Second, the synchronization across multiple processes or between a kernel component and a user process is not supported by the heap functions. Also, in the case of a shared heap between user and kernel mode, the user mode mapping should be read-only to prevent user mode code from corrupting the heap internal structures, which would result in a system crash.

Heap Manager Structure

As shown in Figure 7-5, the heap manager is structured in two layers: an optional front-end layer and the core heap. The core heap handles the basic functionality and is mostly common across the user and kernel mode heap implementations. The core functionality includes the management of blocks inside segments, the management of the segments, policies for extending the heap, committing and decommitting memory, and management of the large blocks.

For user mode heaps only, an optional front-end heap layer can exist on top of the existing core functionality. There are two types of front-end layers: look-aside lists and the Low Fragmentation Heap (or LFH, which is available in Windows XP and later), both of which are described later in this section. Only one front-end layer can be used for one heap at one time.

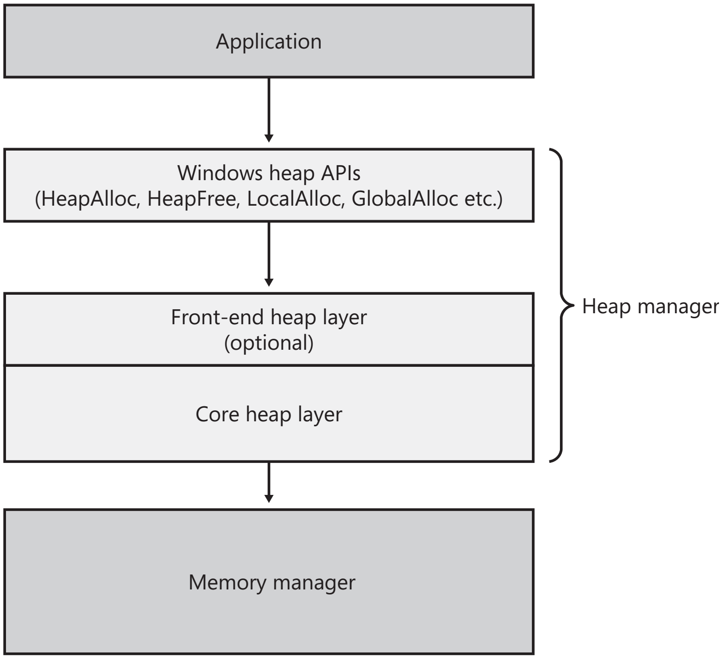


Figure 7-5 Heap manager layers

Heap Synchronization

The heap manager supports concurrent access from multiple threads by default. However, if a process is single threaded or uses an external mechanism for synchronization, it can tell the heap manager to avoid the overhead of synchronization by specifying HEAP_NO_SERIALIZE either at heap creation or on a per-allocation basis.

A process can also lock the entire heap and prevent other threads from performing heap operations for operations that would require consistent states across multiple heap calls. For instance, enumerating the heap blocks in a heap with the Windows function *HeapWalk* requires locking the heap if multiple threads can perform heap operations simultaneously.

If heap synchronization is enabled, there is one lock per heap that protects all internal heap structures. In heavily multithreaded applications (especially when running on multiprocessor systems), the heap lock might become a significant contention point. In that case, performance might be improved by enabling the front-end heap, described in an upcoming section.

Look-Aside Lists

Look-aside lists are single linked lists that allow elementary operations such as “push to the list” or “pop from the list” in a last in, first out (LIFO) order with nonblocking algorithms. A simplified version of these data structures is also available to Windows applications through the functions *InterlockedPopEntrySList* or *InterlockedPushEntrySList*. There are 128 look-aside lists per heap, which handle allocations up to 1 KB on 32-bit platforms and up to 2 KB on 64-bit platforms.

Look-aside lists provide a significant performance improvement over normal heap allocations because multiple threads can concurrently perform allocation and deallocation operations without acquiring the heap global lock. Also, cache locality is optimized by using a LIFO ordering model and by accessing fewer internal data files in each heap operation.

The heap manager maintains the number of blocks in each look-aside list and some counters that help tune the usage of each list independently. If a thread allocates a block of a size that does not exist in the corresponding look-aside list, the heap manager will forward the call to the core heap manager to complete the operation. The heap manager will also update in this case an internal counter of misses at allocations, which will later be used in tuning decisions.

The heap manager creates look-aside lists automatically when a heap is created, as long as no debugging options are enabled and the heap is expandable. Some applications might have compatibility issues as a result of the heap manager's use of look-aside lists. In this case, these legacy applications can be made to run properly by specifying the DisableHeapLookaside flag in the image file execution options for that application. (Image file execution options can be specified using the Imagecfg.exe tool in the Windows 2000 Server Resource Kit, supplement 1.)

The Low Fragmentation Heap

Many applications running in Windows have relatively small heap memory usage (usually less than one megabyte). For this class of applications, the heap manager's best-fit policy helps keep a low memory footprint for each process. However, this strategy does not scale for large processes and multiprocessor machines. In these cases, memory available for heap usage might be reduced as a result of heap fragmentation. Performance can suffer in scenarios where only certain sizes are often used concurrently from different threads scheduled to run on different processors. This is because several processors need to modify the same memory location (for example, the head of the look-aside list for that particular size) at the same time, thus invalidating the corresponding cache line for the other processors.

The Low Fragmentation Heap (LFH) addresses these issues using the core heap manager and look-aside lists. Unlike the look-aside lists that are used as front-end heaps by default if other heap settings are allowing it, LFH is turned on only if an application calls the *HeapSetInformation* function. For large heaps, a significant percentage of allocations is generally grouped in a relatively small number of buckets of certain sizes. The allocation strategy used by LFH is to optimize the usage for these patterns by efficiently handling same-size blocks.

To address scalability, LFH expands the frequently accessed internal structures to a number of slots that is two times larger than the current number of processors on the machine. The assignment of threads to these slots is done by an LFH component called the *affinity manager*. Initially, LFH starts using the first slot for heap allocations; however, if a contention is detected at accessing some internal data, LFH switches the current thread to use a different slot. Further contentions will spread threads on more slots. These slots are controlled for each size bucket, also to improve locality and minimize the overall memory consumption.

Heap Debugging Features

The heap manager includes several features to help detect bugs by using the following heap functions:

- **Enable tail checking** The end of each block carries a signature, which is checked when the block is released. If a buffer overrun destroyed the signature entirely or partially, the heap will report this error.
- **Enable free checking** A free block is filled with a pattern, which is checked at various points when the heap manager needs to access the block (such as at removal from the free list to allocate the block). If the process continued to write to the block after freeing it, the heap manager will detect changes in the pattern and the error will be reported.
- **Parameter checking** This function consists of extensive checking of the parameters passed to the heap functions.
- **Heap validation** The entire heap is validated at each heap call.
- **Heap tagging and stack traces support** This function supports specifying tags for allocation and/or captures user mode stack traces for the heap calls to help narrow the possible causes of a heap error.

The first three options are enabled by default if the loader detects that a process is started under the control of a debugger. (A debugger can override this behavior and turn off these features.) The heap debugging features can be specified for an executable image by setting various debugging flags in the image header using the gflags tool. (See the section "Windows Global Flags" in Chapter 3.) Or, heap debugging options can be enabled using the *!heap* command in the standard Windows debuggers. (See the debugger help for more information.)

Enabling heap debugger options affects all heaps in the process. Also, if any of the heap debug options are enabled, the front-end heap will be disabled automatically and the core heap will be used (with the required debugging options enabled). The front-end heaps are also not used for heaps that are not expandable (because of the extra overhead added to the existing heap structures) or for heaps that do not allow serialization.

Pageheap

Because the tail and free checking options described in the preceding sections might be discovering corruptions that occurred well before the problem was detected, an additional heap debugging tool, called *pageheap*, is provided that directs all or part of the heap calls to a different heap manager. Pageheap is part of the Windows Application Compatibility Toolkit, which can be downloaded from *www.microsoft.com*. The pageheap places allocations at the end of pages so that if a buffer overrun occurs, it will cause an access violation, making it easier to detect the offending code. Optionally, pageheap allows placing the blocks at the beginning of the pages to detect buffer underrun problems. (This is a rare occurrence.) The pageheap also can protect freed pages against any access to detect references to heap blocks after they have been freed.

Note that using the pageheap can result in running out of address space because of the significant overhead added for small allocations. Also, performance can suffer as of result of the increase of references to demand zero pages, loss of locality, and additional overhead caused by frequent calls to validate heap structures. A process can reduce the impact by specifying that the pageheap be used only for blocks of certain sizes, address ranges, and/or originating DLLs.

> **Note** For more information on pageheap, see article 286470 in the Microsoft Knowledge Base (*http://support.microsoft.com*).

Address Windowing Extensions

Although the 32-bit version of Windows can support up to 128 GB of physical memory (as shown in Table 2-4), each 32-bit user process has by default only a 2-GB virtual address space. (This can be configured up to 3 GB when using the /3GB and /USERVA Boot.ini switches, described in the upcoming section “x86 User Address Space Layouts.”) To allow a 32-bit process to allocate and access more physical memory than can be represented in its limited address space, Windows provides a set of functions called *Address Windowing Extensions* (AWE). For example, on a Windows 2000 Advanced Server system with 8 GB of physical memory, a database server application could use AWE to allocate and use perhaps 6 GB of memory as a database cache.

Allocating and using memory via the AWE functions is done in three steps:

1. Allocating the physical memory to be used
2. Creating a region of virtual address space to act as a window to map views of the physical memory
3. Mapping views of the physical memory into the window

To allocate physical memory, an application calls the Windows function *AllocateUserPhysicalPages*. (This function requires the Lock Pages In Memory user right.) The application then uses the Windows *VirtualAlloc* function with the MEM_PHYSICAL flag to create a window in the private portion of the process’s address space that is mapped to some or all of the physical memory previously allocated. The AWE-allocated memory can then be used with nearly all the Windows APIs. (For example, the Microsoft DirectX functions can’t use AWE memory.)

If an application creates a 256-MB window in its address space and allocates 4 GB of physical memory (on a system with more than 4 GB of physical memory), the application can use the *MapUserPhysicalPages* or *MapUserPhysicalPagesScatter* Windows functions to access any portion of the physical memory by mapping the memory into the 256-MB window. The size of the application's virtual address space window determines the amount of physical memory that the application can access with a given mapping. Figure 7-6 shows an AWE window in a server application address space mapped to a portion of physical memory previously allocated by *AllocateUserPhysicalPages*.

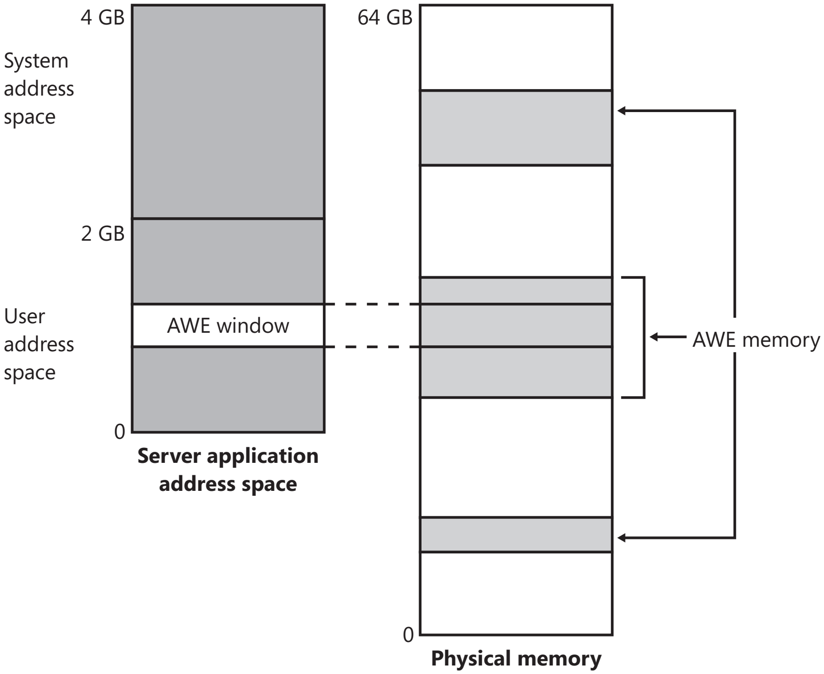


Figure 7-6 Using AWE to map physical memory

The AWE functions exist on all editions of Windows and are usable regardless of how much physical memory a system has. However, AWE is most useful on systems with more than 2 GB of physical memory because it's the only way for a 32-bit process to directly use more than 2 GB of memory. Another use is for security purposes: because AWE memory is never paged out, the data in AWE memory could never have a copy in the paging file that someone could examine by rebooting into an alternate operating system.

Finally, there are some restrictions on memory allocated and mapped by the AWE functions:

- Pages can't be shared between processes.
- The same physical page can't be mapped to more than one virtual address in the same process.
- On older versions of Windows, page protection is limited to read/write. In Windows Server 2003 Service Pack 1 and later, no access and read-only are supported.

For a description of the page table data structures used to map memory on systems with more than 4 GB of physical memory, see the section "Physical Address Extension (PAE)."

System Memory Pools

At system initialization, the memory manager creates two types of dynamically sized memory pools that the kernel-mode components use to allocate system memory:

- **Nonpaged pool** Consists of ranges of system virtual addresses that are guaranteed to reside in physical memory at all times and thus can be accessed at any time (from any IRQL level and from any process context) without incurring a page fault. One of the reasons nonpaged pool is required is because of the rule described in Chapter 2: page faults can't be satisfied at DPC/dispatch level or above.
- **Paged pool** A region of virtual memory in system space that can be paged in and out of the system. Device drivers that don't need to access the memory from DPC/dispatch level or above can use paged pool. It is accessible from any process context.

Both memory pools are located in the system part of the address space and are mapped in the virtual address space of every process. (In Table 7-8, you'll find out where in the system memory they start.) The executive provides routines to allocate and deallocate from these pools; for information on these routines, see the functions that start with *ExAllocatePool* in the Windows DDK documentation.

Uniprocessor systems have three paged pools; multiprocessor systems have five. Having more than one paged pool reduces the frequency of system code blocking on simultaneous calls to pool routines. Both nonpaged and paged pool start at an initial size based on the amount of physical memory on the system and then grow, if necessary, up to a maximum size computed at system boot time.

Note Future releases of Windows might support dynamic pool sizes, which means the maximum size will no longer apply. Therefore, applications and device drivers should not assume that the pool size maximum is a fixed value for each system.

Configuring Pool Sizes

You can override the initial size of these pools by changing the values NonPagedPoolSize and PagedPoolSize in the registry key HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management from 0 (which causes the system to compute the size) to the size desired in bytes. You can't, however, go beyond the maximum pool sizes listed in Table 7-5. A value of 0xFFFFFFFF for PagedPoolSize indicates that the largest possible size is selected, which means allowing a larger paged pool at the expense of system page table entries (PTEs).

Table 7-5 Maximum Pool Sizes

Pool Type	Maximum on 32-Bit Systems	Maximum on 64-Bit Systems
Nonpaged	256 MB (128 MB if booted /3GB)	128 GB
Paged	491.875 MB (Windows 2000 and Windows XP); 650 MB (Windows Server 2003)	128 GB

The computed sizes are stored in four kernel variables, three of which are exposed as performance counters. These variables and counters, as well as the two registry keys that can alter the sizes, are listed in Table 7-6.

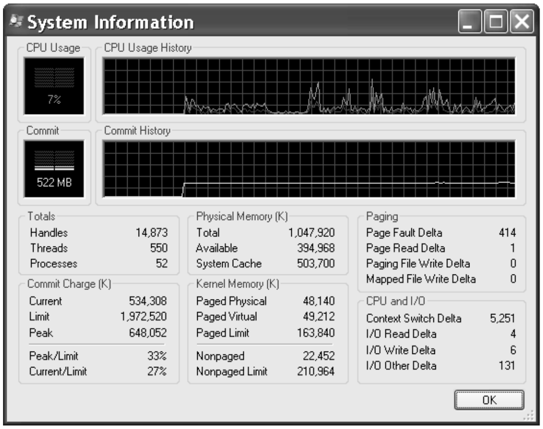
Table 7-6 System Pool Size Variables and Performance Counters

Kernel Variable	Performance Counter	Registry Key to Override	Description
MmSizeOfNonPagedPoolInBytes	Memory: Pool Nonpaged Bytes	Not applicable	Current size of nonpaged pool
MmMaximumNonPagedPoolInBytes	Not available	HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management\NonPagedPoolSize	Maximum size of nonpaged pool
Not available	Memory: Pool Paged Bytes	Not applicable	Current virtual size of paged pool
MmPagedPoolPage (number of pages)	Memory: Pool Paged Resident Bytes	Not applicable	Current physical (resident) size of paged pool
MmSizeOfPagedPoolInBytes	Not available	HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management\PagedPoolSize	Maximum (virtual) size of paged pool

EXPERIMENT: Determining the Maximum Pool Sizes

Because paged and nonpaged pool represent a critical system resource, it is important to know when you're nearing the maximum size computed for your system so that you can determine whether you need to override the default maximum with the appropriate registry values. The pool-size performance counters report only the current size, however, not the maximum size. So you don't know when you're reaching the limit until you've exhausted pool. (As noted earlier, future versions of Windows might support dynamically sized pools. Therefore, the need to check the pool maximums might no longer be necessary in the future.)

You can obtain the pool maximums by using either Process Explorer or live kernel debugging (explained in Chapter 1). To view pool maximums with Process Explorer, click on View, System Information. The pool maximums are displayed in the Kernel Memory section as shown below:



Totals
Handles 14,873
Threads 550
Processes 52

Commit Charge (K)
Current 534,308
Limit 1,972,520
Peak 648,052
Peak/Limit 33%
Current/Limit 27%

Physical Memory (K)
Total 1,047,920
Available 394,968
System Cache 503,700

Kernel Memory (K)
Paged Physical 48,140
Paged Virtual 49,212
Paged Limit 163,840
Nonpaged 22,452
Nonpaged Limit 210,964

Paging
Page Fault Delta 414
Page Read Delta 1
Paging File Write Delta 0
Mapped File Write Delta 0

CPU and I/O
Context Switch Delta 5,251
I/O Read Delta 4
I/O Write Delta 6
I/O Other Delta 131

Note that for Process Explorer to retrieve this information, it must have access to the symbols for the kernel running on your system. (For a description of how to configure Process Explorer to use symbols, see the experiment "Viewing Process Details with Process Explorer" in Chapter 1.)

To view the same information by using the kernel debugger, you can use the *!vm* command as shown below:

```
lkd> !vm

*** Virtual Memory Usage ***
Physical Memory:    261980   ( 1047920 Kb)
Page File: \??\C:\pagefile.sys
   Current:   1024000Kb Free Space:    862236Kb
   Minimum:   1024000Kb Maximum:      1024000Kb
Available Pages:     96077   (  384308 Kb)
ResAvail Pages:     194722   (  778888 Kb)
Locked IO Pages:        89   (     356 Kb)
Free System PTEs:   175933   (  703732 Kb)
Free NP PTEs:        31832   (  127328 Kb)
Free Special NP:         0   (       0 Kb)
Modified Pages:        529   (    2116 Kb)
Modified PF Pages:     529   (    2116 Kb)
NonPagedPool Usage:   5617   (   22468 Kb)
NonPagedPool Max:    52741   (  210964 Kb)
PagedPool 0 Usage:    6582   (   26328 Kb)
PagedPool 1 Usage:    3009   (   12036 Kb)
PagedPool 2 Usage:    3044   (   12176 Kb)
PagedPool Usage:     12635   (   50540 Kb)
PagedPool Maximum:   40960   (  163840 Kb)
```

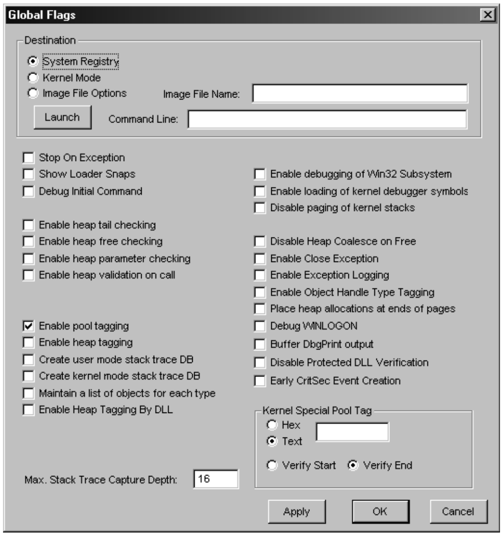
On this system, nonpaged and paged pool were far from their maximums. You can also examine the values of the kernel variables listed in Table 7-6:

```
kd> dd mmmaximumnonpagedpoolinbytes l1
8047f620  0328c000
kd> ? 328c000
Evaluate expression: 53002240 = 0328c000
kd> dd mmsizeofpagedpoolinbytes l1
80470a98  06800000
kd> ? 6800000
Evaluate expression: 109051904 = 06800000
```

From this example, you can see that the maximum size of nonpaged pool is 53,002,240 bytes (approximately 50 MB) and the maximum size of paged pool is 109,051,904 bytes (104 MB). On the test system we used for this example, current nonpaged pool usage was 5.5 MB and paged pool usage was 34 MB, so both pools were far from full.

Monitoring Pool Usage

The Memory performance counter object has separate counters for the size of nonpaged pool and paged pool (both virtual and physical). In addition, the Poolmon utility (in the Windows Support Tools) allows you to monitor the detailed usage of nonpaged and paged pool. To do so, you must have the internal Enable Pool Tagging option enabled. Pool tagging is turned on by default as of Windows Server 2003 (in fact, it cannot be turned off) and in all checked builds of Windows. To enable pool tagging, run the Gflags utility in the Windows Support Tools, Platform SDK, or DDK, and select Enable Pool Tagging, as shown here:



Global Flags

Destination
● System Registry
○ Kernel Mode
○ Image File Options   Image File Name:
Launch   Command Line:

☐ Stop On Exception
☐ Show Loader Snaps
☐ Debug Initial Command

☐ Enable heap tail checking
☐ Enable heap free checking
☐ Enable heap parameter checking
☐ Enable heap validation on call

☑ Enable pool tagging
☐ Enable heap tagging
☐ Create user mode stack trace DB
☐ Create kernel mode stack trace DB
☐ Maintain a list of objects for each type
☐ Enable Heap Tagging By DLL

Max. Stack Trace Capture Depth: 16

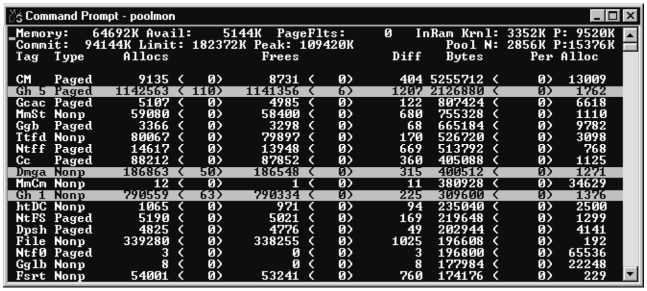
☐ Enable debugging of Win32 Subsystem
☐ Enable loading of kernel debugger symbols
☐ Disable paging of kernel stacks

☐ Disable Heap Coalesce on Free
☐ Enable Close Exception
☐ Enable Exception Logging
☐ Enable Object Handle Type Tagging
☐ Place heap allocations at ends of pages
☐ Debug WINLOGON
☐ Buffer DbgPrint output
☐ Disable Protected DLL Verification
☐ Early CritSec Event Creation

Kernel Special Pool Tag
○ Hex
● Text
○ Verify Start ● Verify End

Apply OK Cancel

Then click Apply, and reboot the system. After the system reboots, run Poolmon; you should see a display like this one:



```
 Memory:   64692K Avail:     5144K  PageFlts:      0   InRam Krnl: 3352K P: 9520K
 Commit:  94144K Limit: 182372K Peak: 109420K                Pool N: 2856K P:15376K
 Tag  Type     Allocs            Frees            Diff   Bytes          Per Alloc

 CM   Paged      9135 (    0)       8731 (    0)    404 5255712 (       0)   13009
 Gh 5 Paged   1142563 (  110)    1141356 (    6)   1207 2126880 (       0)    1762
 Gcac Paged      5107 (    0)       4985 (    0)    122  807424 (       0)    6618
 MmSt Nonp      59080 (    0)      58400 (    0)    680  755328 (       0)    1110
 Ggb  Paged      3366 (    0)       3298 (    0)     68  665184 (       0)    9782
 Ttfd Nonp      80067 (    0)      79897 (    0)    170  526720 (       0)    3098
 Ntff Paged     14617 (    0)      13948 (    0)    669  513792 (       0)     768
 Cc   Paged     88212 (    0)      87852 (    0)    360  405088 (       0)    1125
 Dmga Nonp     186863 (   50)     186548 (    0)    315  400512 (       0)    1271
 MmCm Nonp         12 (    0)          1 (    0)     11  380928 (       0)   34629
 Gh 1 Nonp     790559 (   63)     790334 (    0)    225  309600 (       0)    1376
 htDC Nonp       1065 (    0)        971 (    0)     94  235040 (       0)    2500
 NtFS Paged      5190 (    0)       5021 (    0)    169  219648 (       0)    1299
 Dpsh Paged      4825 (    0)       4776 (    0)     49  202944 (       0)    4141
 File Nonp     339280 (    0)     338255 (    0)   1025  196608 (       0)     192
 Ntf0 Paged         3 (    0)          0 (    0)      3  196800 (       0)   65536
 Gglb Nonp          8 (    0)          0 (    0)      8  177984 (       0)   22248
 Fsrt Nonp      54001 (    0)      53241 (    0)    760  174176 (       0)     229
```

The highlighted lines represent changes to the display. (You can disable the highlighting feature by typing *l* while running Poolmon. Type *l* again to reenable highlighting.) Type *?* while Poolmon is running to bring up its help screen. You can configure which pools you want to monitor (paged, nonpaged, or both) and the sort order. Also, the command-line options are shown, which allow you to monitor specific structures (or everything but one structure type). For example, the command *poolmon -iCM* will monitor only structures of type CM (the configuration manager, which manages the registry). The columns have the meanings shown in Table 7-7.

Table 7-7 **Poolmon Columns**

Column	Explanation
Tag	Four-byte tag given to the pool allocation
Type	Pool type (paged or nonpaged pool)
Allocs	Count of all allocations (The number in parentheses shows the difference in the Allocs column since the last update.)
Frees	Count of all Frees (The number in parentheses shows the difference in the Frees column since the last update.)
Diff	Count of Allocs minus Frees
Bytes	Total bytes consumed by this structure type (The number in parentheses shows the difference in the Bytes column since the last update.)
Per Alloc	Size in bytes of a single instance of this structure type

In this example, CM structures (the configuration manager, or registry) are taking up the most space in paged pool, and MmSt structures (a memory management–related structure used for mapped files) are taking up the most space in nonpaged pool.

For a description of the meaning of the pool tags used by Windows, see the file \Program Files\Debugging Tools for Windows\Triage\Pooltag.txt. (This file is installed as part of the Windows Debugging Tools described in Chapter 1.) Because third-party device driver pool tags are not listed in this file, you can use the "-c" switch on the version of Poolmon that comes with the Windows Server 2003 Device Driver Kit (DDK) to generate a local pool tag file

(Localtag.txt). This file will contain pool tags used by drivers found on your system, including third-party drivers. (Note that if a device driver binary has been deleted after it was loaded, its pool tags will not be recognized.)

Alternatively, you can search the device drivers on your system for a pool tag by using the Strings.exe tool from *www.sysinternals.com*. For example, the command

```
strings \windows\system32\drivers\*.sys > findstr /i "abcd"
```

will display drivers that contain the string “abcd”. Note that device drivers do not necessarily have to be located in \Windows\System32\Drivers—they can be in any folder. To list the full path of all loaded drivers, click Start, click Run, and type **Msinfo32**. Then click on Software Environment and System Drivers. As already noted, if a device driver has been loaded and then deleted from the system, it will not be listed here.

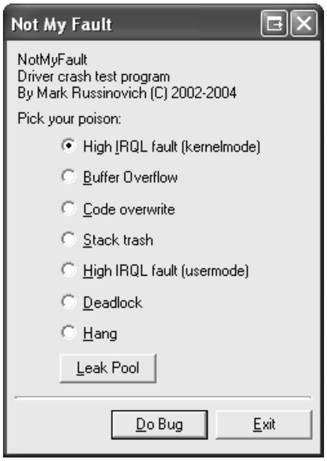
An alternative to view pool usage by device driver is to enable the pool tracking feature of Driver Verifier, explained later in this chapter. While this makes the mapping from pool tag to device driver unnecessary, it does require a reboot (to enable Driver Verifier on the desired drivers). After rebooting with pool tracking enabled, you can either run the graphical Driver Verifier Manager (\Windows\System32\Verifier.exe) or use the Verifier /Log command to send the pool usage information to a file.

Finally, if you are looking at a crash dump, you can view pool usage with the kernel debugger *!poolused* command. The command *!poolused 2* shows nonpaged pool usage sorted by structure tag using the most amount of pool. The command *!poolused 4* lists paged pool usage, again sorted by structure tag using the most amount of pool. The following example shows the partial output from these two commands:

```
 lkd> !poolused 2
   Sorting by  NonPaged Pool Consumed
  Pool Used:
            NonPaged            Paged
 Tag    Allocs     Used    Allocs     Used
 MmCm      386  2237440         0        0    Calls made to MmAllocateContiguousMemory , Bin
ary: nt!mm
 File    12544  2121232         0        0    File objects
 Devi      385  1487184         0        0    Device objects
 Ntfr    19163  1226888         0        0    ERESOURCE , Binary: ntfs.sys
kd> !poolused 4
   Sorting by Paged Pool Consumed

  Pool Used:
            NonPaged            Paged
 Tag    Allocs     Used    Allocs     Used
 CM         11      704      5146 22311904
 Gh 5        0        0       727  2523648
 MmSt        0        0       968  1975872
 Ntff       10     2240       790   682560
 Gla1        1      128       383   600544
 Ttfd        0        0       265   545440
```

EXPERIMENT: Troubleshooting a Pool Leak

In this experiment, you will fix a real paged pool leak on your system so that you can put to use the techniques described in the previous section to track down the leak. The leak will be generated by the NotMyFault tool, which you can download from *www.sysinternals.com/windowsinternals.shtml*. (Note that this tool is not listed on the main Sysinternals tools list.) When you run NotMyFault.exe, it loads a device driver Myfault.sys and presents the following dialog box:



NotMyFault
Driver crash test program
By Mark Russinovich (C) 2002-2004

Pick your poison:

● High IRQL fault (kernelmode)
○ Buffer Overflow
○ Code overwrite
○ Stack trash
○ High IRQL fault (usermode)
○ Deadlock
○ Hang

1. Click the Leak Pool button. This causes NotMyFault to begin sending requests to the Myfault device driver to allocate paged pool. (Do not click the Do Bug button or you will experience a system crash; this button is used in Chapter 14 to demonstrate various types of crashes.) NotMyFault will continue to do this until you click the Stop Leaking button. Note that the paged pool is not released even when you close the program; the pool is permanently leaked until you reboot the system. However, because you will leak pool only for a short period, this should not cause any problems on your system.

2. While the pool is leaking, first open Task Manager and click on the Performance tab. You should notice Paged Pool climbing. You can also check this with Process Explorer's System Information display. (Click on Show and then System Information.)

3. To determine the pool tag that is leaking, run Poolmon and press the "b" key to sort by the number of bytes. Press "p" twice so that Poolmon is showing only paged pool. You should notice the pool tag "Leak" climbing to the top of the list. (Poolmon shows changes to pool allocations by highlighting the lines that change.)

4. Now press the “Stop Leaking” button so that you don’t exhaust paged pool on your system.
5. Using the technique described in the previous section, run Strings (from *www.sysinternals.com*) to look for driver binaries that contain the pool tag “Leak”:

```
Strings \windows\system32\drivers\*.sys  |  findstr Leak
```

This should display a match on the file Myfault.sys, thus confirming it as the driver using the “Leak” pool tag.

Look-Aside Lists

Windows also provides a fast memory allocation mechanism called *look-aside lists*. The basic difference between pools and look-aside lists is that while general pool allocations can vary in size, a look-aside list contains only fixed-sized blocks. Although the general pools are more flexible in terms of what they can supply, look-aside lists are faster because they don’t use any spinlocks and also because the system doesn’t have to search for free memory that fits a varying size allocation.

Executive components and device drivers can create look-aside lists that match the size of frequently allocated data structures by using the *ExInitializeNPagedLookasideList* and *ExInitializePagedLookasideList* functions (documented in the DDK). To minimize the overhead of multiprocessor synchronization, several executive subsystems (such as the I/O manager, cache manager, and object manager) create separate look-aside lists for each processor for their frequently accessed data structures. The executive also creates a general per-processor paged and nonpaged look-aside list for small allocations (256 bytes or less).

If a look-aside list is empty (as it is when it’s first created), the system must allocate from paged or nonpaged pool. But if it contains a freed structure, the allocation can be satisfied very quickly. (The list grows as structures are returned to it.) The pool allocation routines automatically tune the number of freed buffers that look-aside lists store according to how often a device driver or executive subsystem allocates from the list—the more frequent the allocations, the more buffers are stored on a list. Look-aside lists are automatically reduced in size if they aren’t being allocated from. (This check happens once per second when the balance set manager system thread wakes up and calls the function *KiAdjustLookasideDepth*.)

EXPERIMENT: Viewing the System Look-Aside Lists

You can display the contents and sizes of the various system look-aside lists with the kernel debugger *!lookaside* command. The following excerpt is from the output of this command:

```
kd> !lookaside

Lookaside "nt!IopSmallIrpLookasideList" @ 804758a0 "Irps"
    Type     =     0000 NonPagedPool
    Current Depth  =        3   Max Depth  =        4
    Size           =      148   Max Alloc  =      592
    AllocateMisses =       32   FreeMisses =        9
    TotalAllocates =       52   TotalFrees =       32
    Hit Rate       =       38%  Hit Rate   =       71%
Lookaside "nt!IopLargeIrpLookasideList" @ 804756a0 "Irpl"
    Type     =     0000 NonPagedPool
    Current Depth  =        4   Max Depth  =        4
    Size           =      436   Max Alloc  =     1744
    AllocateMisses =     2623   FreeMisses =     2443
    TotalAllocates =     7039   TotalFrees =     6863
    Hit Rate       =       62%  Hit Rate   =       64%
"
Lookaside "nt!IopMdlLookasideList" @ 80475740 "Mdl "
    Type     =     0000 NonPagedPool
    Current Depth  =        3   Max Depth  =        4
    Size           =      120   Max Alloc  =      480
    AllocateMisses =     7017   FreeMisses =     1824
    TotalAllocates =    10901   TotalFrees =     5711
    Hit Rate       =       35%  Hit Rate   =       68%
§
Total NonPaged currently allocated for above lists = 4200
Total NonPaged potential for above lists           = 6144
Total Paged currently allocated for above lists    = 5136
Total Paged potential for above lists              = 12032
```

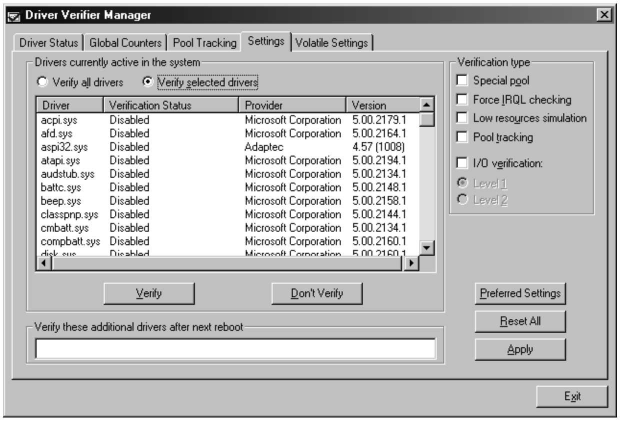
Driver Verifier

Driver Verifier is a mechanism that can be used to help find and isolate commonly found bugs in device driver or other kernel-mode system code. Microsoft uses Driver Verifier to check its own device drivers as well as all device drivers that vendors submit for Hardware Compatibility List (HCL) testing. Doing so ensures that the drivers on the HCL are compatible with Windows and free from common driver errors. (Although not described in this book, there is also a corresponding Application Verifier tool that has resulted in quality improvements for user-mode code in Windows.)

Driver Verifier consists of support in several system components: the memory manager, I/O manager, and the HAL all have driver verification options that can be enabled. This section describes the memory management–related verification options Driver Verifier provides. The options related to device drivers are described in Chapter 9.

Driver Verifier Configuration and Initialization

To configure Driver Verifier and view statistics about its operation, run the Driver Verifier Manager (\Windows\System32\Verifier.exe). (The Windows 2000 version is shown in Figure 7-7.) It displays several tabbed pages. You use the Settings tab to specify which device drivers you want to verify (including an option to Verify All Drivers) and what types of verification you want performed.



Drivers currently active in the system

○ Verify all drivers ● Verify selected drivers

Driver	Verification Status	Provider	Version
acpi.sys	Disabled	Microsoft Corporation	5.00.2179.1
afd.sys	Disabled	Microsoft Corporation	5.00.2164.1
aspi32.sys	Disabled	Adaptec	4.57 (1008)
atapi.sys	Disabled	Microsoft Corporation	5.00.2194.1
audstub.sys	Disabled	Microsoft Corporation	5.00.2134.1
battc.sys	Disabled	Microsoft Corporation	5.00.2148.1
beep.sys	Disabled	Microsoft Corporation	5.00.2158.1
classpnp.sys	Disabled	Microsoft Corporation	5.00.2144.1
cmbatt.sys	Disabled	Microsoft Corporation	5.00.2134.1
compbatt.sys	Disabled	Microsoft Corporation	5.00.2160.1
disk.sys	Disabled	Microsoft Corporation	5.00.2160.1

Verify these additional drivers after next reboot

Verification type

□ Special pool

□ Force IRQL checking

□ Low resources simulation

□ Pool tracking

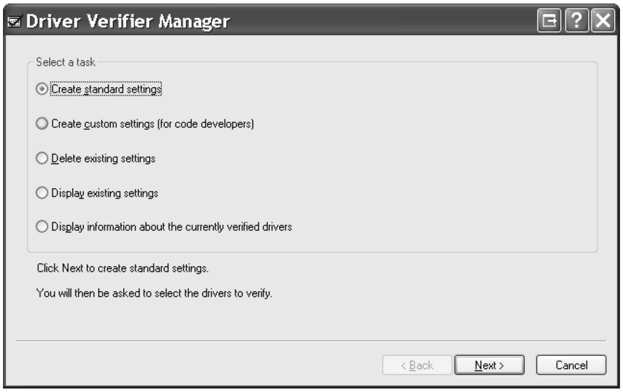
□ I/O verification:

● Level 1

○ Level 2

Figure 7-7 The Driver Verifier Manager in Windows 2000

In Windows XP and Windows Server 2003, Verifier.exe has a wizard-style interface, as shown in Figure 7-8.



Select a task

● Create standard settings

○ Create custom settings (for code developers)

○ Delete existing settings

○ Display existing settings

○ Display information about the currently verified drivers

Click Next to create standard settings.

You will then be asked to select the drivers to verify.

Figure 7-8 The Driver Verifier Manager in Windows XP and Windows Server 2003

You can also enable and disable Driver Verifier as well as display current settings by using its command line interface. Type "**verifier /?**" to see the switches.

The settings are stored in the registry under HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management. The value VerifyDriverLevel contains a bitmask that represents the verification types enabled. The VerifyDrivers value contains the names of the drivers to validate. (These values won't exist in the registry until you select drivers to verify in the Driver Verifier Manager.) If you choose to verify all drivers, VerifyDrivers is set to an asterisk (*) character. Depending on the settings you have made, you might need to reboot the system for the selected verification to occur.

Early in the boot process, the memory manager reads the Driver Verifier registry values to determine which drivers to verify and which Driver Verifier options you enabled. (Note that if you boot in safe mode, any Driver Verifier settings are ignored.) Subsequently, if you've selected at least one driver for verification, the kernel checks the name of every device driver it loads into memory against the list of drivers you've selected for verification. For every device driver that appears in both places, the kernel invokes the *MiApplyDriverVerifier* function, which replaces the driver's references to a number of kernel functions with references to Driver Verifier–equivalent versions of those functions. For example, *ExAllocatePool* is replaced with a call to *VerifierAllocatePool*. The windowing system driver also makes similar changes to use Driver Verifier–equivalent functions.

Now that we've reviewed how to set up Driver Verifier, let's examine the four memory-related verification options that can be applied to device drivers: Special Pool, Pool Tracking, Force IRQL Checking, and Low Resources Simulation.

Special Pool The Special Pool option causes the pool allocation routines to bracket pool allocations with an invalid page so that references before or after the allocation will result in a kernel-mode access violation, thus crashing the system with the finger pointed at the buggy driver. Special pool also causes some additional validation checks to be performed when a driver allocates or frees memory.

When special pool is enabled, the pool allocation routines allocate a region of kernel memory for Driver Verifier to use. Driver Verifier redirects memory allocation requests that drivers under verification make to the special pool area rather than to the standard kernel-mode memory pools. When a device driver allocates memory from special pool, Driver Verifier rounds up the allocation to an even-page boundary. Because Driver Verifier brackets the allocated page with invalid pages, if a device driver attempts to read or write past the end of the buffer, the driver will access an invalid page and the memory manager will raise a kernel-mode access violation.

Figure 7-9 shows an example of the special pool buffer that Driver Verifier allocates to a device driver when Driver Verifier checks for overrun errors.

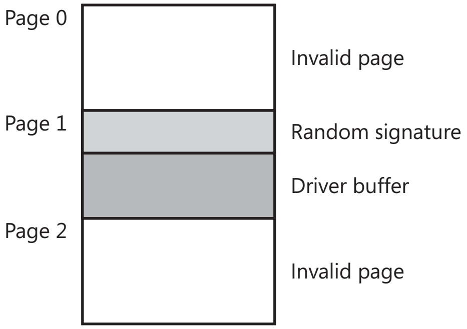


Figure 7-9 Layout of special pool allocations

By default, Driver Verifier performs *overrun* detection. It does this by placing the buffer that the device driver uses at the end of the allocated page and fills the beginning of the page with a random pattern. Although the Driver Verifier Manager doesn't let you specify underrun detection, you can set this type of detection manually by adding the DWORD registry value HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\MemoryManagement\PoolTagOverruns and setting it to 0 (or by running the Gflags utility and selecting the Verify Start option instead of the default option Verify End). When Windows enforces underrun detection, Driver Verifier allocates the driver's buffer at the beginning of the page rather than at the end.

The overrun-detection configuration includes some measure of underrun detection as well. When the driver frees its buffer to return the memory to Driver Verifier, Driver Verifier ensures that the pattern preceding the buffer hasn't changed. If the pattern is modified, the device driver has underrun the buffer and written to memory outside the buffer.

Special pool allocations also check to ensure that the processor IRQL at the time of an allocation and deallocation is legal. This check catches an error that some device drivers make: allocating pageable memory from an IRQL at DPC/dispatch level or above.

You can also configure special pool manually by adding the DWORD registry value HKLM\SYSTEMCurrentControlSet\Control\Session Manager\Memory Management\PoolTag, which represents the allocation tags the system uses for special pool. Thus, even if Driver Verifier isn't configured to verify a particular device driver, if the tag the driver associates with the memory it allocates matches what is specified in the PoolTag registry value, the pool allocation routines will allocate the memory from special pool. If you set the value of PoolTag to 0x0000002a or the wildcard (*), all memory that drivers allocate is from special pool, provided there's enough virtual and physical memory. (The drivers will revert to allocating from regular pool if there aren't enough free pages—bounding exists, but each allocation uses two pages.)

Pool Tracking If pool tracking is enabled, the memory manager checks at driver unload time whether the driver freed all the memory allocations it made. If it didn't, it crashes the system, indicating the buggy driver. Driver Verifier also shows general pool statistics on the Driver Verifier Manager's Pool Tracking tab. You can also use the *!verifier* kernel debugger command. This command shows more information than Driver Verifier and is useful to driver writers.

Force IRQL Checking One of the most common device driver bugs occurs when a driver accesses pageable data or code when the processor on which the device driver is executing is at an elevated IRQL. As explained in Chapter 3, the memory manager can't service a page fault when the IRQL is DPC/dispatch level or above. The system often doesn't detect instances of a device driver accessing pageable data when the processor is executing at a high IRQL level because the pageable data being accessed happens to be physically resident at the time. At other times, however, the data might be paged out, which results in a system crash with the stop code IRQL_NOT_LESS_OR_EQUAL (that is, the IRQL wasn't less than or equal to the level required for the operation attempted—in this case, accessing pageable memory).

Although testing device drivers for this kind of bug is usually difficult, Driver Verifier makes it easy. If you select the Force IRQL Checking option, Driver Verifier forces all kernel-mode pageable code and data out of the system working set whenever a device driver under verification raises the IRQL. The internal function that does this is *MmTrimAllSystemPagableMemory*. With this setting enabled, whenever a device driver under verification accesses pageable memory when the IRQL is elevated, the system instantly detects the violation and the resulting system crash identifies the faulty driver.

Low Resources Simulation Enabling Low Resources Simulation causes Driver Verifier to randomly fail memory allocations that verified device drivers perform. In the past, developers wrote many device drivers under the assumption that kernel memory would always be available and that if memory ran out, the device driver didn't have to worry about it because the system would crash anyway. However, because low-memory conditions can occur temporarily, it's important that device drivers properly handle allocation failures that indicate kernel memory is exhausted.

Beginning 7 minutes after the system boots—which is enough time to get past the critical initialization period in which a low-memory condition might prevent a device driver from loading—Driver Verifier starts randomly failing allocation calls for device drivers it is verifying. If a driver doesn't correctly handle allocation failures, this will likely show up as a system crash.

Driver Verifier is a valuable addition to the arsenal of verification and debugging tools available to device driver writers. Many device drivers that first ran with Driver Verifier had bugs that Driver Verifier was able to expose. Thus, Driver Verifier has resulted in an overall improvement in the quality of all kernel-mode code running in Windows.

Virtual Address Space Layouts

This section describes the components in the user and system address space, followed by the specific layouts on 32-bit and 64-bit systems. This information helps you to understand the limits on process and system virtual memory on both platforms.

Three main types of data are mapped into the virtual address space in Windows: per-process private code and data, sessionwide code and data, and systemwide code and data.

As explained in Chapter 1, each process has a private address space that cannot be accessed by other processes (unless they have permission to open the process for read or write access). Threads within the process can never access virtual addresses outside this private address space, unless they map to shared memory sections and/or use the cross-process memory functions that allow access to another process's address space. The information that describes the process virtual address space, called *page tables*, is described in the section on address translation. Page tables are marked as kernel-mode-only accessible pages so that user threads in a process cannot modify their own address space layout.

On systems with multiple sessions (Windows 2000 Server with Terminal Services installed, Windows XP, and Windows Server 2003), *session space* contains information global to each session. (For a description of sessions, see Chapter 2.) A *session* consists of the processes and other system objects (such as the window station, desktops, and windows) that represent a single user's workstation logon session. Each session has a session-specific paged pool area used by the kernel-mode portion of the Windows subsystem (Win32k.sys) to allocate session-private GUI data structures. In addition, each session has its own copy of the Windows subsystem process (Csrss.exe) and logon process (Winlogon.exe). The session manager process (Smss.exe) is responsible for creating new sessions, which includes loading a session-private copy of Win32k.sys, creating the session-private object manager namespace, and creating the session-specific instances of the Csrss and Winlogon processes.

To virtualize sessions, all sessionwide data structures are mapped into a region of system space called *session space*. When a process is created, this range of addresses is mapped to the pages appropriate to the session that the process belongs to. The size of the space for mapped views in session space can be configured by modifying registry values under HKLM\System\CurrentControlSet\Control\Session Manager\Memory Management. (For 32-bit systems, these registry values are ignored if the system is booted /3GB.)

Finally, *system space* contains global operating system code and data structures visible to each process. System space consists of the following components:

- **System code** Contains the operating system image, HAL, and device drivers used to boot the system.
- **System mapped views** Used to map Win32k.sys, the loadable kernel-mode part of the Windows subsystem, as well as kernel-mode graphics drivers it uses. (See Chapter 2 for more information on Win32k.sys.)
- **Hyperspace** A special region used to map the process working set list and to temporarily map other physical pages for operations such as zeroing a page on the free list (when the zero list is empty and a zero page is needed), invalidating page table entries in other page tables (such as when a page is removed from the standby list), and process creation to set up a new process's address space.
- **System working set list** The working set list data structures that describe the system working set.*
- **System cache** Virtual address space used to map files open in the system cache. (See Chapter 11 for detailed information about the cache manager.)
- **Paged pool** Pageable system memory heap.

- **System page table entries (PTEs)** Pool of system PTEs used to map system pages such as I/O space, kernel stacks, and memory descriptor lists. You can see how many system PTEs are available by examining the value of the Memory: Free System Page Table Entries counter in the Performance tool.
- **Nonpaged pool** Nonpageable system memory heap, usually existing in two parts–one in the lower end of system space and one in the upper end.
- **Crash dump information** Reserved to record information about the state of a system crash.
- **HAL usage** System memory reserved for HAL-specific structures.

> **Note** *Internally, the system working set is called the *system cache working set*. This term is misleading, however, because it includes not only the system cache but also the paged pool, pageable system code and data, and pageable driver code and data.

Now that we've described the basic components of the virtual address space in Windows, let's examine the specific layout on the x86, IA-64, and x64 platforms.

x86 User Address Space Layouts

By default, each user process on the 32-bit version of Windows can have up to a 2-GB private address space; the operating system takes the remaining 2 GB. Windows 2000 Advanced Server, Windows 2000 Datacenter Server, Windows XP Service Pack 2 and later, and Windows Server 2003 (all versions) support a boot-time option (the /3GB switch in Boot.ini) that allows 3-GB user address spaces. Windows XP and Windows Server 2003 support an additional switch (/USERVA), which allows specifying a user address space size between 2 GB and 3 GB. (The value is in number of megabytes.) These two possible address space layouts are shown in Figure 7-10.

The ability for a 32-bit process to grow beyond 2 GB was added as a short-term solution to accommodate the need for applications such as database servers to keep more data in memory than could be done with a 2-GB address space. The AWE functions described earlier in this chapter provide a much better solution to the need for accessing more data than can fit in the limited 2-GB (or 3-GB) process address space.

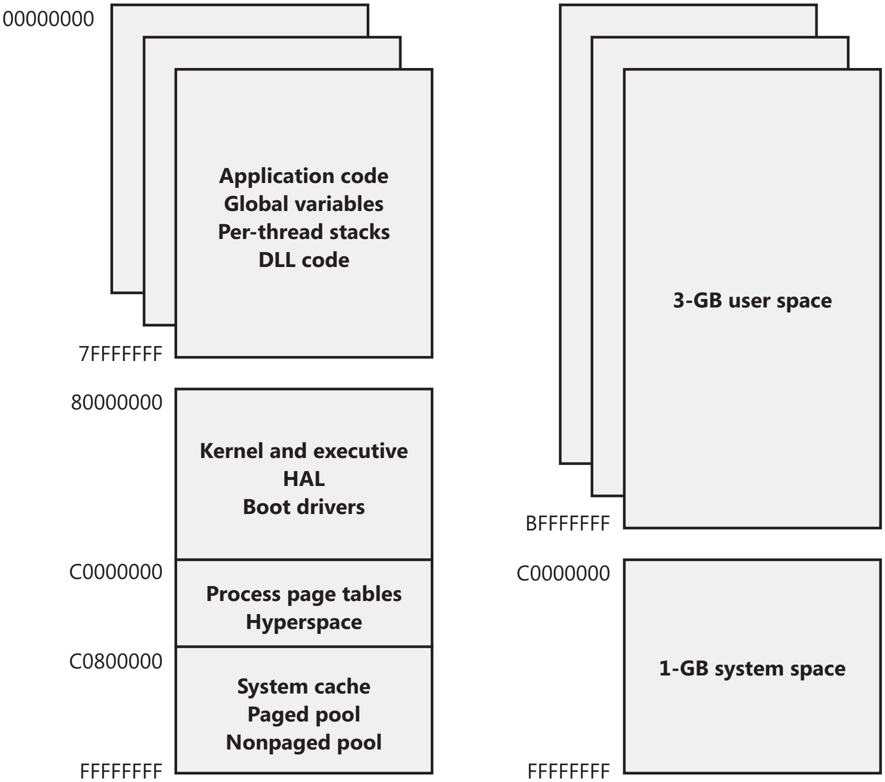


Figure 7-10 x86 virtual address space layouts

For a process to grow beyond 2 GB of address space, the image file must have the IMAGE_FILE_LARGE_ADDRESS_AWARE flag set in the image header. Otherwise, Windows reserves the additional address space for that process so that the application won't see virtual addresses greater than 0x7FFFFFFF. (This is to avoid breaking applications that might break with such addresses.) You set this flag by specifying the linker flag /LARGEADDRESSAWARE when building the executable. This flag has no effect when running the application on a system with a 2-GB user address space. (If you boot with /3GB in versions of Windows Server that do not support the /3GB switch, system space is reduced to 1 GB, but user processes are still limited to 2 GB, even if the large-address-space aware flag is set on an image that is run.)

Several system images are marked as large address space aware so that they can take advantage of systems running with large process address spaces. These include:

- Lsass.exe—the Local Security Authority Subsystem
- Inetinfo.exe—Internet Information Server
- Chkdsk.exe—the Check Disk utility
- Dllhst3g.exe—a special version of Dllhost.exe (for COM+ applications)

Finally, because by default, memory allocations using VirtualAlloc start with low virtual addresses and grow higher, unless a process allocates a lot of virtual memory or it has a very fragmented virtual address space, it will never get back very high virtual addresses. Therefore, for testing purposes, you can force memory allocations to start from high addresses by adding a DWORD registry value, HKLM\System\CurrentControlSet\Control\Session Manager\Memory Management\AllocationPreference, and setting it to 0x100000.

x86 System Address Space Layout

This section describes the detailed layout and contents of system space in 32-bit Windows. Figure 7-11 shows the overall structure on x86 systems with a 2-GB system space.

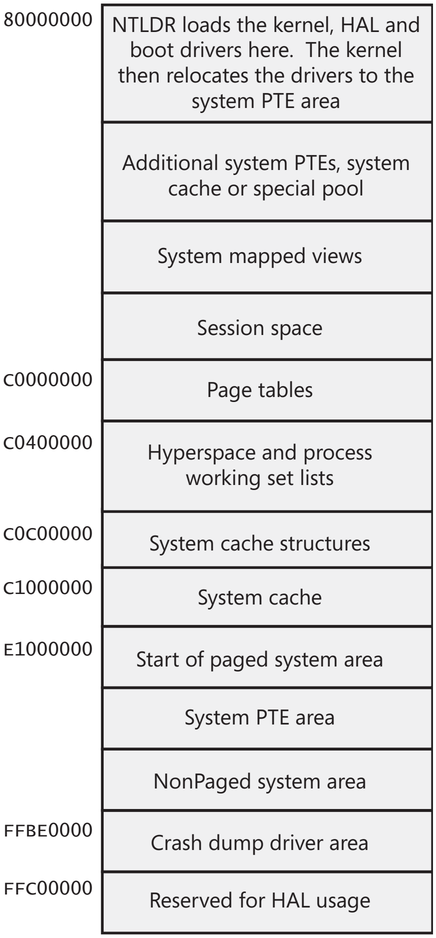

80000000
NTLDR loads the kernel, HAL and boot drivers here. The kernel then relocates the drivers to the system PTE area
Additional system PTEs, system cache or special pool
System mapped views
Session space
C0000000
Page tables
C0400000
Hyperspace and process working set lists
C0C00000
System cache structures
C1000000
System cache
E1000000
Start of paged system area
System PTE area
NonPaged system area
FFBE0000
Crash dump driver area
FFC00000
Reserved for HAL usage

Figure 7-11 x86 system space layout (not proportional)

Table 7-8 lists the kernel variables that contain start and end addresses of various system space regions. Some of these regions are fixed; some are computed at system boot time on the basis of memory size and whether the system is running a client or server edition of Windows.

Table 7-8 System Variables That Describe System Space Regions

System Variables	Description	x86 2-GB System Space (non-PAE)	x86 1-GB System Space (non-PAE)
MmSystemRangeStart	Start address of system space	0x80000000	0xC0000000
MmSystemCacheWorkingSetList	System working set list	0xC0C00000	Calculated
MmSystemCacheStart	Start of system cache	Calculated	Calculated
MmSystemCacheEnd	End of system cache	Calculated	Calculated
MiSystemCacheStartExtra	Start of system cache or system PTE extension	Calculated	0
MiSystemCacheEndExtra	End of system cache or PTE extension	0xC0000000	0
MmPagedPoolStart	Start of paged pool	Calculated	Calculated
MmPagedPoolEnd	End of paged pool	Calculated (maximum size is 650 MB)	Calculated (minimum size is 160 MB)
MmNonPagedSystemStart	Start of system PTEs	Calculated (lowest value is 0xEB000000)	Calculated
MmNonPagedPoolStart	Start of nonpaged pool	Calculated	Calculated
MmNonPagedPoolExpansionStart	Start of nonpaged pool expansion	Calculated	Calculated
MmNonPagedPoolEnd	End of nonpaged pool	0xFFBE0000	0xFFBE0000

x86 Session Space

For systems with multiple sessions, the code and data unique to each session is mapped into system address space but shared by the processes in that session. Figure 7-12 shows the general layout of session space.

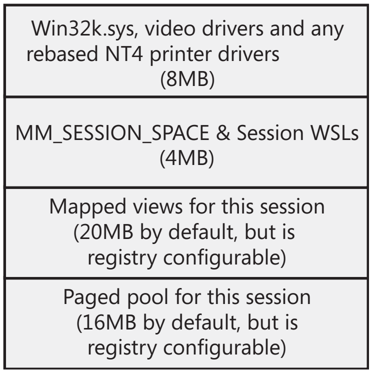

Win32k.sys, video drivers and any rebased NT4 printer drivers (8MB)
MM_SESSION_SPACE & Session WSLs (4MB)
Mapped views for this session (20MB by default, but is registry configurable)
Paged pool for this session (16MB by default, but is registry configurable)

Figure 7-12 x86 session space layout (not proportional)

The sizes of the components of session space can be configured by adding registry values under the key HKLM\System\CurrentControlSet\Session Manager\Memory Management. These registry values along with the kernel variables that contain the actual sizes are listed in Table 7-9.

Table 7-9 Session Space Configuration Values

Registry Value Under	Kernel Variable
SessionImageSize	MmSessionImageSize
SessionViewSize	MmSessionViewSize
SessionPoolSize	MmSessionPoolSize

EXPERIMENT: Viewing Sessions

You can display which processes are members of which sessions by examining the session ID process performance counter. This can be viewed with Task Manager, Process Explorer, or the Performance Tool. Using the kernel debugger, you can list the active sessions with the *!session* command as follows:

```
lkd> !session
Sessions on machine: 3
Valid Sessions: 0 1 2
Current Session 0
```

Then you can set the active session using the *!session –s* command and display the address of the session data structures and the processes in that session with the *!sprocess* command:

```
lkd> !session -s 2
Sessions on machine: 3
Implicit process is now 8144f3a0
Using session 2
lkd> !sprocess
Dumping Session 2
```

```
_MM_SESSION_SPACE f9d5a000
_MMSESSION        f9d5ad80
PROCESS 8144f3a0  SessionId: 2  Cid: 0d18    Peb: 7ffdf000  ParentCid: 0134
    DirBase: 07ef7000  ObjectTable: e1855bb0  HandleCount:  71.
    Image: csrss.exe

PROCESS 813f6500  SessionId: 2  Cid: 0d04    Peb: 7ffdf000  ParentCid: 0134
    DirBase: 0f3fd000  ObjectTable: e18f3008  HandleCount: 131.
    Image: winlogon.exe
```

To view the details of the session, dump the MM_SESSION_SPACE structure using the *dt* command as follows:

```
lkd> dt nt!_mm_session_space f9d4a000
nt!_MM_SESSION_SPACE
   +0x000 GlobalVirtualAddress : 0xf9d4a000
   +0x004 ReferenceCount   : 0x19
   +0x008 u                : __unnamed
   +0x00c SessionId        : 0
   +0x010 ProcessList      : _LIST_ENTRY [ 0x817d92cc - 0x812de88c ]
   +0x018 LastProcessSwappedOutTime : _LARGE_INTEGER 0x0
   +0x020 SessionPageDirectoryIndex : 0x6a29
   +0x024 NonPagablePages  : 0xf
   +0x028 CommittedPages   : 0x3f
   +0x02c PagedPoolStart   : 0xbc000000
   +0x030 PagedPoolEnd     : 0xbc3fffff
   +0x034 PagedPoolBasePde : 0xc0300bc0
   +0x038 Color            : 0xd6
   +0x03c ProcessOutSwapCount : 5
   +0x040 SessionPoolAllocationFailures : [4] 0
   +0x050 ImageList        : _LIST_ENTRY [ 0x81801fa8 - 0x816933a0 ]
   +0x058 LocaleId         : 0x409
   +0x05c AttachCount      : 0
   +0x060 AttachEvent      : _KEVENT
   +0x070 LastProcess      : (null)
   +0x074 ProcessReferenceToSession : 25
   +0x078 WsListEntry      : _LIST_ENTRY [ 0x8056d058 - 0x8056d058 ]
   +0x080 Lookaside        : [26] _GENERAL_LOOKASIDE
   +0xd80 Session          : _MMSESSION
   +0xdc0 PagedPoolMutex   : _KGUARDED_MUTEX
   +0xde0 PagedPoolInfo    : _MM_PAGED_POOL_INFO
   +0xe00 Vm               : _MMSUPPORT
   +0xe60 Wsle             : 0xbf400038
   +0xe64 Win32KDriverUnload : 0xbf9169ab     +ffffffffbf9169ab
   +0xe68 PagedPool        : _POOL_DESCRIPTOR
   +0x1e98 PageTables       : 0x816e7008
   +0x1e9c ImageLoadingCount : 0
```

EXPERIMENT: Viewing Session Space Utilization

You can view session space memory utilization with the *!vm 4* command in the kernel debugger. For example, the following output was taken from a 32-bit Windows Server 2003 Enterprise Edition system with two active sessions:

```
lkd> !vm 4
.
.
Terminal Server Memory Usage By Session:

Session Paged Pool Maximum is 4096K
Session View Space Maximum is 49152K

Session ID 0 @ f9d4a000:
Paged Pool Usage:          0K
Commit Usage:            252K

Session ID 1 @ f9d4c000:
Paged Pool Usage:          0K
Commit Usage:            172K
```

Executing the same command on an x64 Windows Server 2003 Enterprise Edition system with two active sessions shows the much larger session space regions:

```
Terminal Server Memory Usage By Session:
Session Paged Pool Maximum is 65536K
Session View Space Maximum is 106496K

Session ID 0 @ fffffadfe4bb9000:
Paged Pool Usage:          0K
Commit Usage:           4608K
Session ID 1 @ fffffadfe4c5f000:
Paged Pool Usage:          0K
Commit Usage:            584K
```

System Page Table Entries

System page table entries are used to dynamically map system pages such as I/O space, kernel stacks, and memory descriptor lists. System PTEs aren't an infinite resource. For example, Windows 2000 can describe only 660 MB of variable system virtual address space (440 MB of which could be contiguous). With the 32-bit edition of Windows XP and Windows Server 2003, the number of available system page table entries (PTEs) has increased such that the system can describe 1.3 GB of system virtual address space, 960 MB of which can be contiguous. On 64-bit Windows, system PTEs can describe up to 128 GB of contiguous virtual address space.

You can see how many system PTEs are available by examining the value of the Memory: Free System Page Table Entries counter in the Performance tool. By default, Windows calculates at boot time how many system PTEs to allocate based on the memory size. You can override this calculation by setting the registry value HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management\SystemPages to the number of PTEs you want. (This might be needed to support, for example, a device that requires a large number of system PTEs, such as a video card with 512 MB of video memory that has to be mapped all at one time.) If the value is 0xFFFFFFFF, the maximum number of system PTEs will be reserved.

64-Bit Address Space Layouts

The theoretical 64-bit virtual address space is 16 exabytes (18,446,744,073,709,551,616 bytes, or approximately 17.2 billion gigabytes). Unlike on the x86 where the default address space is divided in two parts (half for a process and half for the system), the 64-bit address is divided into a number of different sized regions whose components match conceptually the portions of user, system, and session space. The various sizes of these regions, listed in Table 7-10, represent current implementation limits that could easily be extended in future releases. (The sizes on the 32-bit x86 platform are included for comparison purposes.) Clearly, 64-bits provides a tremendous leap in terms of address space sizes.

Table 7-10 The 64-Bit Address Space Sizes

Region	IA-64	x64	x86
Process Address Space	7152 GB	8192 GB	2 to 3 GB
System PTE Space	128 GB	128 GB	1.2 GB
System Cache	128 GB	128 GB	960 MB
Paged Pool	128 GB	128 GB	470 to 650 MB
Nonpaged Pool	128GB	128 GB	256 MB

The detailed IA-64 and x64 address space layouts vary slightly. The IA-64 address space layout is shown in Figure 7-13, and the x64 address space layout is shown in Figure 7-14.

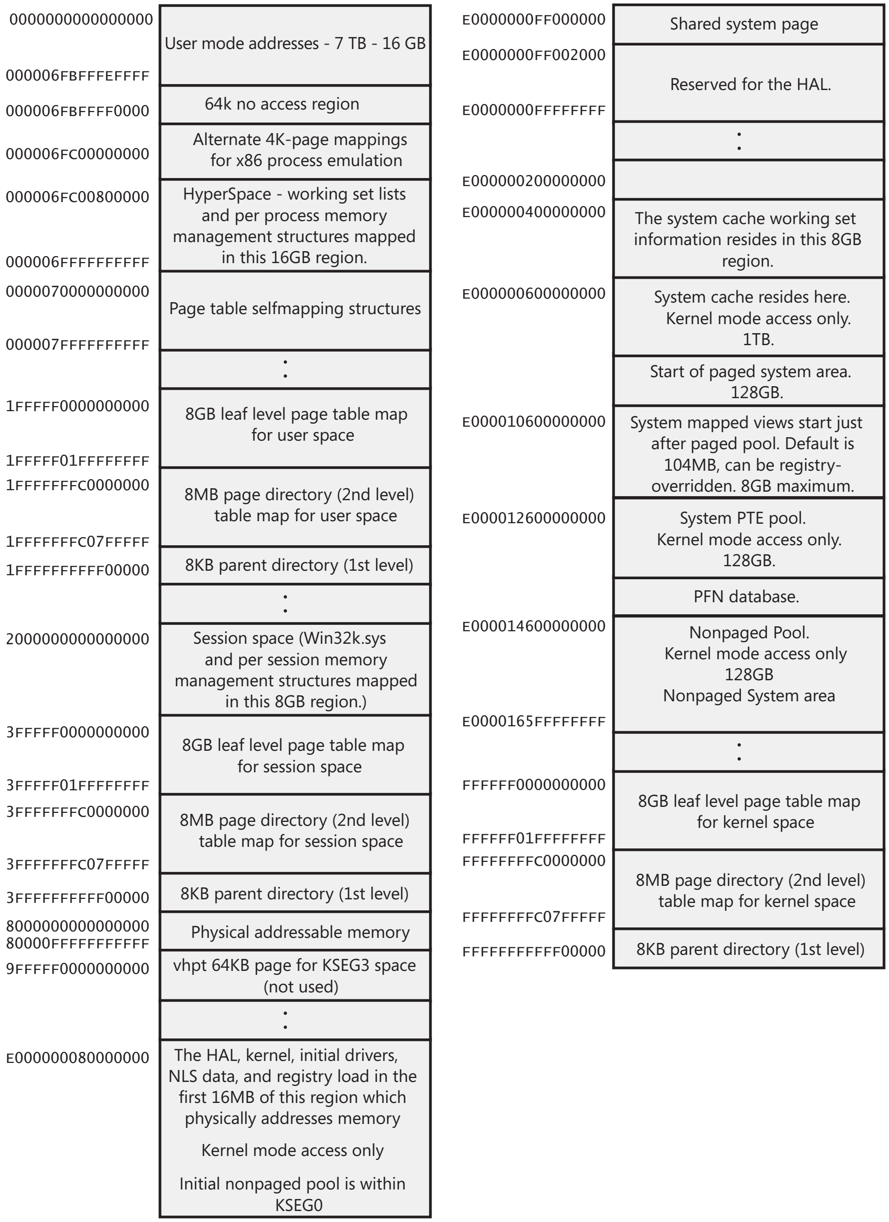

0000000000000000
User mode addresses - 7 TB - 16 GB
000006FBFFFEFFFF
000006FBFFFF0000
64k no access region
000006FC00000000
Alternate 4K-page mappings for x86 process emulation
000006FC00800000
HyperSpace - working set lists and per process memory management structures mapped in this 16GB region.
000006FFFFFFFFFF
0000070000000000
Page table selfmapping structures
000007FFFFFFFFFF
1FFFFF0000000000
8GB leaf level page table map for user space
1FFFFF01FFFFFFFF
1FFFFFFFC0000000
8MB page directory (2nd level) table map for user space
1FFFFFFFC07FFFFF
1FFFFFFFFFF00000
8KB parent directory (1st level)
2000000000000000
Session space (Win32k.sys and per session memory management structures mapped in this 8GB region.)
3FFFFF0000000000
8GB leaf level page table map for session space
3FFFFF01FFFFFFFF
3FFFFFFFC0000000
8MB page directory (2nd level) table map for session space
3FFFFFFFC07FFFFF
3FFFFFFFFFF00000
8KB parent directory (1st level)
8000000000000000
80000FFFFFFFFFFF
Physical addressable memory
9FFFFF0000000000
vhpt 64KB page for KSEG3 space (not used)
E000000080000000
The HAL, kernel, initial drivers, NLS data, and registry load in the first 16MB of this region which physically addresses memory
Kernel mode access only
Initial nonpaged pool is within KSEG0
E0000000FF000000
Shared system page
E0000000FF002000
Reserved for the HAL.
E0000000FFFFFFFF
E000000200000000
E000000400000000
The system cache working set information resides in this 8GB region.
E000000600000000
System cache resides here. Kernel mode access only. 1TB.
Start of paged system area. 128GB.
E000010600000000
System mapped views start just after paged pool. Default is 104MB, can be registry-overridden. 8GB maximum.
E000012600000000
System PTE pool. Kernel mode access only. 128GB.
PFN database.
E000014600000000
Nonpaged Pool. Kernel mode access only 128GB
Nonpaged System area
E0000165FFFFFFFF
FFFFFF0000000000
8GB leaf level page table map for kernel space
FFFFFF01FFFFFFFF
FFFFFFFFC0000000
8MB page directory (2nd level) table map for kernel space
FFFFFFFFC07FFFFF
FFFFFFFFFFF00000
8KB parent directory (1st level)

Figure 7-13 IA-64 address space layout

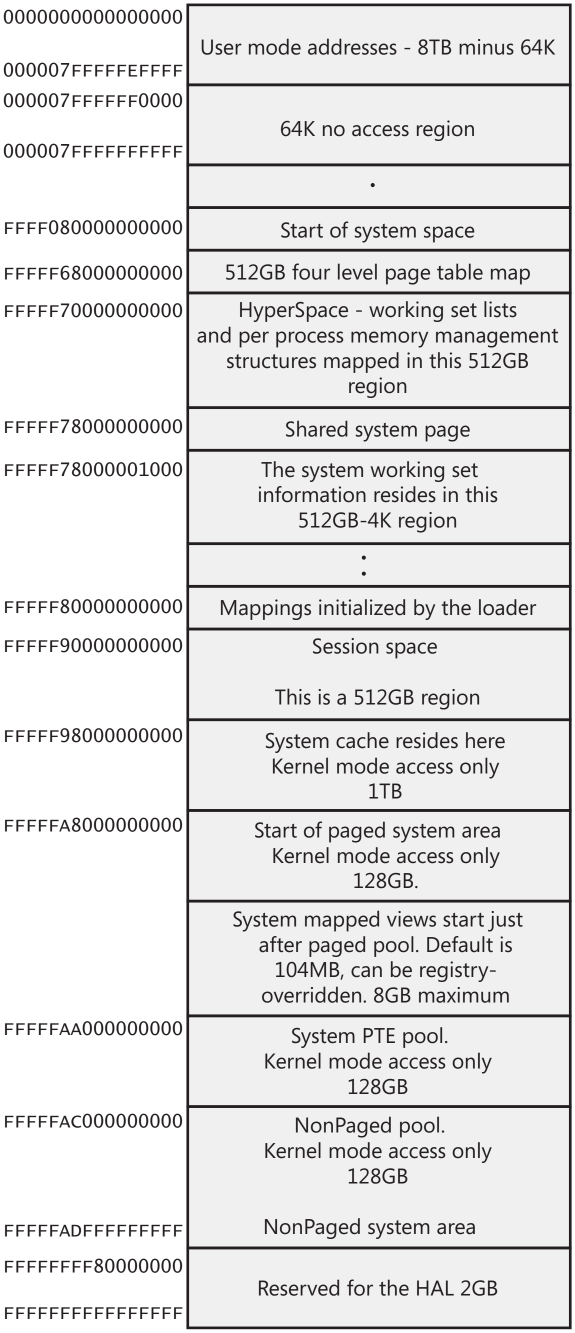


Figure 7-14 x64 address space layout

Address Translation

Now that you've seen how Windows structures the virtual address space, let's look at how it maps these address spaces to real physical pages. User applications and system code reference virtual addresses. This section starts with a detailed description of 32-bit x86 address translation, followed by a brief description of the differences on the 64-bit IA-64 and x64 platform. In the next section, we'll describe what happens when such a translation doesn't resolve to a physical memory address (paging) and explain how Windows manages physical memory via working sets and the page frame database.

x86 Virtual Address Translation

Using data structures the memory manager creates and maintains called *page tables*, the CPU translates virtual addresses into physical addresses. Each virtual address is associated with a system-space structure called a *page table entry* (PTE), which contains the physical address to which the virtual one is mapped. For example, Figure 7-15 shows how three consecutive virtual pages are mapped to three physically discontiguous pages on an x86.

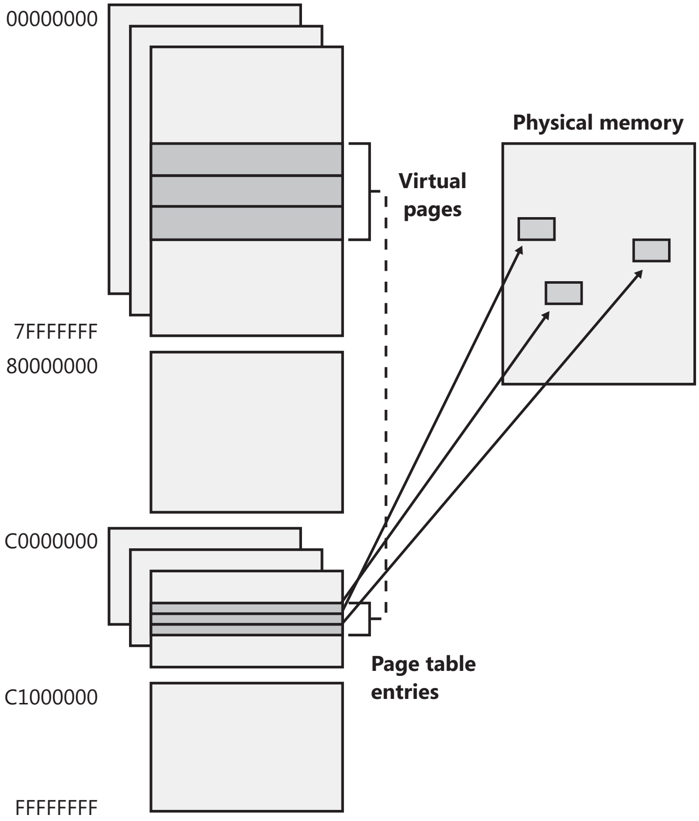


Figure 7-15 Mapping virtual addresses to physical memory (x86)

The dashed line connecting the virtual pages to the PTEs in Figure 7-15 represents the indirect relationship between virtual pages and physical memory.

Note Kernel-mode code (such as device drivers) can reference physical memory addresses by mapping them to virtual addresses. For more information, see the memory descriptor list (MDL) support routines described in the DDK documentation.

By default, Windows on an x86 system uses a two-level page table structure to translate virtual to physical addresses. (x86 systems running the PAE kernel use a three-level page table—this section assumes non-PAE systems.) A 32-bit virtual address is interpreted as three separate components—the page directory index, the page table index, and the byte index—that are used as indexes into the structures that describe page mappings, as illustrated in Figure 7-16. The page size and the PTE width dictate the width of the page directory and page table index fields. For example, on x86 systems, the byte index is 12 bits because pages are 4096 bytes (212 = 4096).

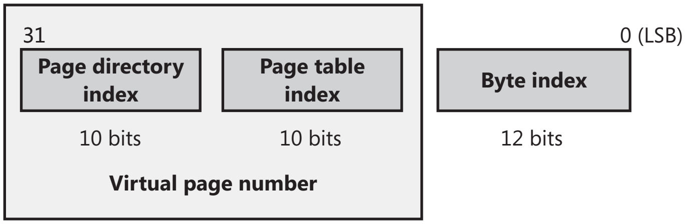


Figure 7-16 Components of a 32-bit virtual address on x86 systems

The *page directory index* is used to locate the page table in which the virtual address's PTE is located. The *page table index* is used to locate the PTE, which, as mentioned earlier, contains the physical address to which a virtual page maps. The *byte index* finds the proper address within that physical page. Figure 7-17 shows the relationship of these three values and how they are used to map a virtual address into a physical address.

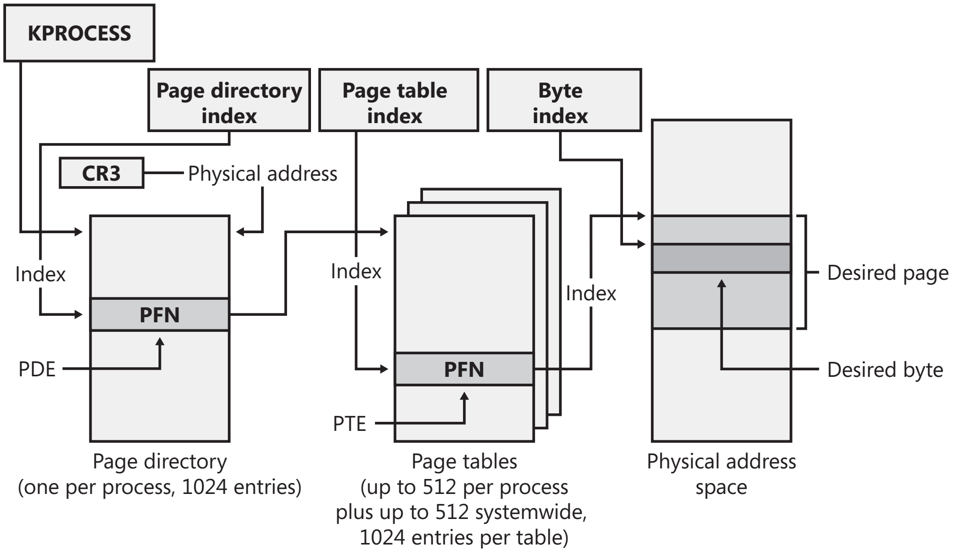


Figure 7-17 Translating a valid virtual address (x86-specific)

The following basic steps are involved in translating a virtual address:

1. The memory management hardware locates the page directory for the current process. On each process context switch, the hardware is told the address of a new process page directory, typically by the operating system setting a special CPU register.
2. The page directory index is used as an index into the page directory to locate the page directory entry (PDE) that describes the location of the page table needed to map the virtual address. The PDE contains the page frame number (PFN) of the page table (if it is resident—page tables can be paged out).
3. The page table index is used as an index into the page table to locate the PTE that describes the physical location of the virtual page in question.
4. The PTE is used to locate the page. If the page is valid, it contains the PFN of the page in physical memory that contains the virtual page. If the PTE indicates that the page isn't valid, the memory management fault handler locates the page and tries to make it valid. (See the section on page fault handling.) If the page can't be made valid (for example, because of a protection fault), the fault handler generates an access violation or a bug check.
5. When the PTE is pointed to a valid page, the byte index is used to locate the address of the desired data within the physical page.

Now that you have the overall picture, let's look at the detailed structure of page directories, page tables, and PTEs.

Page Directories

Each process has a single *page directory*, a page the memory manager creates to map the location of all page tables for that process. The physical address of the process page directory is stored in the kernel process (KPROCESS) block, but it is also mapped virtually at address 0xC0300000 on x86 systems (0xC0600000 on systems running the PAE kernel image). All code running in kernel mode references virtual addresses, not physical ones. (For more detailed information about KPROCESS and other process data structures, refer to Chapter 6.)

The CPU knows the location of the page directory page because a special register (CR3 on x86 systems) inside the CPU that is loaded by the operating system contains the physical address of the page directory. Each time a context switch occurs to a thread that is in a different process than that of the currently executing thread, this register is loaded from the KPROCESS block of the target process being switched to by the context-switch routine in the kernel. Context switches between threads in the same process don't result in reloading the physical address of the page directory because all threads within the same process share the same process address space.

The page directory is composed of *page directory entries* (PDEs), each of which is 4 bytes long (8 bytes on systems running the PAE kernel image) and describes the state and location of all the possible page tables for that process. (As described later in the chapter, page tables are created on demand, so the page directory for most processes points only to a small set of page tables.) The format of a PDE isn't repeated here because it's mostly the same as a hardware PTE.

On x86 systems running in non-PAE mode, 1024 page tables are required to describe the full 4-GB virtual address space. The process page directory that maps these page tables contains 1024 PDEs. Therefore, the page directory index needs to be 10 bits wide (2^{10} = 1024). On x86 systems running in PAE mode, there are 512 entries in a page table (because the page directory index is 9 bits wide). Because there are 4 page directories, the result is a maximum of 2048 page tables.

EXPERIMENT: Examining the Page Directory and PDEs

You can see the physical address of the currently running process's page directory by examining the DirBase field in the *!process* kernel debugger output:

Physical address of PDE

```
kd> !process
PROCESS 80145880   Cid: 0000    Peb: 00000000   ParentCid: 0000
    DirBase: 00030000  ObjectTable: 80695668  TableSize: 126.
    Image: Idle
    VadRoot 0 Clone 0 Private 0. Modified 0. Locked 0.
    80145A3C MutantState Locked OwningThread 0
    Process Lock Owned by Thread 0
        .
        .
        .
```

You can see the page directory's virtual address by examining the kernel debugger output for the PTE of a particular virtual address, as shown here:

Virtual address of PDE

Contents of PDE

```
kd> !pte 50001
00050001  -  PDE at C0300000         PTE at C0000140
          contains 00700067         contains 00E63047
          pfn 00700  --DA--UW     pfn 00E63  --D---UWV
```

PFN of page table page

State and protection of page table page

The PTE part of the kernel debugger output is defined in the section "Page Table Entries."

Because Windows provides a private address space for each process, each process has its own set of process page tables to map that process's private address space. However, the page tables that describe system space are shared among all processes (and session space is shared among processes in a session). To avoid having multiple page tables describing the same virtual memory, when a process is created, the page directory entries that describe system space are initialized to point to the existing system page tables. If the process is part of a session, session space page tables are also shared by pointing the session space page directory entries to the existing session page tables.

But as shown in Figure 7-18, not all processes have the same view of system space. For example, if paged pool expansion requires the allocation of a new system page table, the memory manager doesn't go back and update all the process page directories to point to the new system page table. Instead, it updates the process page directories when the processes reference the new virtual address.

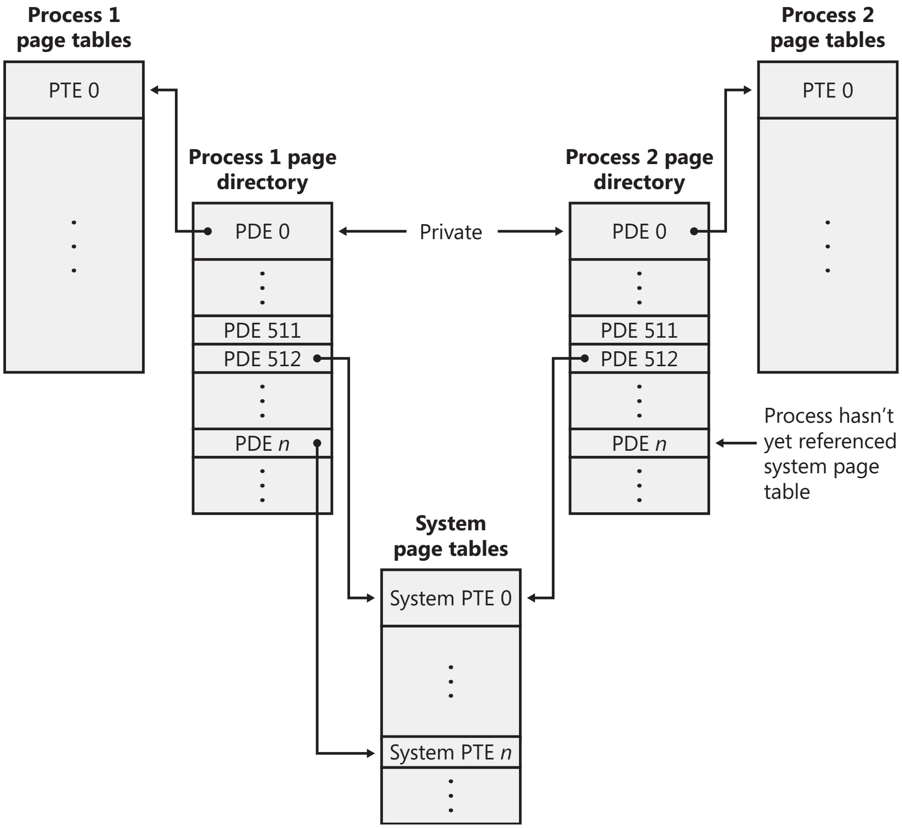


Figure 7-18 System and process-private page tables

Thus, a process can take a page fault when referencing paged pool that is in fact physically resident because its process page directory doesn't yet point to the new system page table that describes the new area of pool. Page faults don't occur when accessing nonpaged pool, even though it too can be expanded, because Windows builds enough system page tables to describe the maximum size during system initialization.

Page Tables and Page Table Entries

The process page directory entries point to individual page tables. Page tables are composed of an array of PTEs. The virtual address's page table index field (as shown in Figure 7-17) indicates which PTE within the page table maps the data page in question. On x86 systems, the page table index is 10 bits wide (9 on PAE), allowing you to reference up to 1024 4-byte PTEs

(512 8-byte PTEs on PAE systems). However, because 32-bit Windows provides a 4-GB private virtual address space, more than one page table is needed to map the entire address space. To calculate the number of page tables required to map the entire 4-GB process virtual address space, divide 4 GB by the virtual memory mapped by a single page table. Recall that each page table on an x86 system maps 4 MB (2 MB on PAE) of data pages. Thus, 1024 page tables (4 GB/4 MB)—or 2048 page tables (4 GB/2 MB) for PAE—are required to map the full 4-GB address space.

You can use the *!pte* command in the kernel debugger to examine PTEs. (See the experiment "Translating Addresses.") We'll discuss valid PTEs here and invalid PTEs in a later section. Valid PTEs have two main fields: the page frame number (PFN) of the physical page containing the data or of the physical address of a page in memory, and some flags that describe the state and protection of the page, as shown in Figure 7-19.

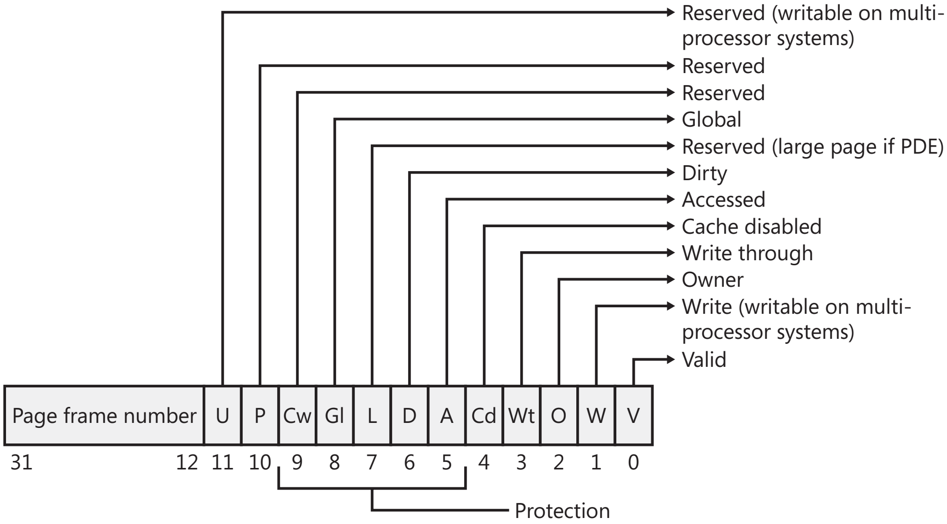


Figure 7-19 Valid x86 hardware PTEs

As you'll see later, the bits labeled Reserved in Figure 7-19 are used only when the PTE isn't valid. (The bits are interpreted by software.) Table 7-11 briefly describes the hardware-defined bits in a valid PTE.

Table 7-11 PTE Status and Protection Bits

Name of Bit	Meaning
Accessed	Page has been read.
Cache disabled	Disables caching for that page.
Dirty	Page has been written to.
Global	Translation applies to all processes. (For example, a translation buffer flush won't affect this PTE.)

Table 7-11 PTE Status and Protection Bits

Name of Bit	Meaning
Large page	Indicates that the PDE maps a 4-MB page (or 2 MB on PAE systems). See the section "Large and Small Pages" earlier in the chapter.
Owner	Indicates whether user-mode code can access the page or whether the page is limited to kernel-mode access.
Valid	Indicates whether the translation maps to a page in physical memory.
Write through	Disables caching of writes to this page so that changes are immediately flushed to disk.
Write	On uniprocessor systems, indicates whether the page is read/write or read-only; on multiprocessor systems, indicates whether the page is writable. (The Write bit is stored in a reserved bit in the PTE.)

On x86 systems, a hardware PTE contains a Dirty bit and an Accessed bit. The Accessed bit is clear if a physical page represented by the PTE hasn't been read or written; the processor sets this bit when the page is first read or written. The processor sets the Dirty bit only when a page is first written. In addition to those two bits, the x86 architecture has a Write bit that provides page protection. When this bit is clear, the page is read-only; when it is set, the page is read/write. If a thread attempts to write to a page with the Write bit clear, a memory management exception occurs and the memory manager's access fault handler (described in the next section) must determine whether the thread can write to the page (for example, if the page was really marked copy-on-write) or whether an access violation should be generated.

Hardware PTEs on multiprocessor x86 systems have an additional Write bit implemented in software that is intended to avoid stalls when flushing the PTE cache (called the *translation look-aside buffer*, described in the next section) across processors. This bit indicates that a page has been modified by another processor.

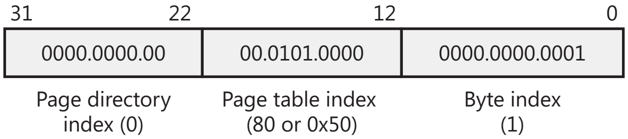
Byte Within Page

Once the memory manager has found the physical page in question, it must find the requested data within that page. This is where the byte index field comes in. The byte index field tells the CPU which byte of data in the page you want to reference. On x86 systems, the byte index is 12 bits wide, allowing you to reference up to 4096 bytes of data (the size of a page). So, adding the byte offset to the physical page number retrieved from the PTE completes the translation of a virtual address to a physical address.

EXPERIMENT: Translating Addresses

To clarify how address translation works, let's go through a real example of translating a virtual address on an x86 non-PAE system, using the available tools in the kernel debugger to examine page directories, page tables, and PTEs. In this example, we'll use a process that has virtual address 0x50001 currently mapped to a valid physical address. In later examples, you'll see how to follow address translation for invalid addresses with the kernel debugger.

First let's convert 0x50001 to binary and break it into the three fields that are used to translate an address. In binary, 0x50001 is 101.0000.0000.0000.0001. Breaking it into the component fields yields the following:



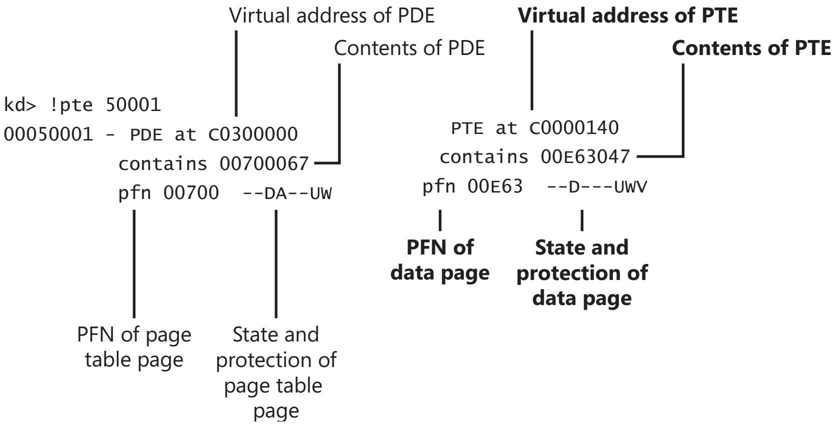
To start the translation process, the CPU needs the physical address of the process page directory, stored in the CR3 register while a thread in that process is running. You can display this address either by examining the CR3 register itself or by dumping the KPROCESS block for the process in question using the *!process* command, as shown here:

```
kd> !process
PROCESS 81555020 Cid: 0099 Peb: 7ffdf000  ParentCid: 0094
  DirBase: 012f0000  ObjectTable: 80695ba8  TableSize: 46.
  Image: IEXPLORE.EXE
        .
        .
        .
kd> r cr3
Cr3: 012F0000
```

Physical address of page directory

In this case, the page directory is stored at physical address 0x12F0000. As shown in the preceding illustration, the page directory index field in this example is 0. Therefore, the PDE is at physical address 0x12F0000.

The kernel debugger *!pte* command displays the PDE and PTE that describe a virtual address, as shown here:



```
kd> !pte 50001
00050001 - PDE at C0300000          PTE at C0000140
          contains 00700067       contains 00E63047
          pfn 00700  --DA--UW     pfn 00E63  --D---UWV
```

Virtual address of PDE
Contents of PDE
Virtual address of PTE
Contents of PTE
PFN of page table page
State and protection of page table page
PFN of data page
State and protection of data page

In the first column the kernel debugger displays the PDE, and in the second column it displays the PTE. Notice that the PDE address is shown as a virtual address, not a physical address—as noted earlier, the process page directory starts at virtual address 0xC0300000 on x86 systems. Because we're looking at the first PDE in the page directory, the PDE address is the same as the page directory address.

The PTE is at virtual address 0xC0000140. You can compute this address by multiplying the page table index (0x50 in this example) by the size of a PTE: 0x50 multiplied by 4 equals 0x140. Because the memory manager maps page tables starting at 0xC0000000, adding 140 yields the virtual address shown in the kernel debugger output: 0xC0000140. The page table page is at PFN 0x700, and the data page is at PFN 0xe63.

The PTE flags are displayed to the right of the PFN number. For example, the PTE that describes the page being referenced has flags of D---UWV. *D* here stands for dirty (the page has been modified), *U* for user-mode page (as opposed to a kernel-mode page), *W* for writable page (rather than read-only), and *V* for valid. (The PTE represents a valid page in physical memory.)

Translation Look-Aside Buffer

As we've learned so far, each address translation requires two lookups: one to find the right page table in the page directory and one to find the right entry *in* the page table. Because doing two additional memory lookups for every reference to a virtual address would result in unacceptable system performance, most CPUs cache address translations so that repeated accesses to the same addresses don't have to be retranslated. The processor provides such a cache in the form of an array of associative memory called the *translation look-aside buffer*, or TLB. Associative memory, such as the TLB, is a vector whose cells can be read simultaneously and compared to a target value. In the case of the TLB, the vector contains the virtual-to-physical page mappings of the most recently used pages, as shown in Figure 7-20, and the type of page protection applied to each page. Each entry in the TLB is like a cache entry, whose tag holds portions of the virtual address and whose data portion holds a physical page number, protection field, valid bit, and usually a dirty bit indicating the condition of the page to which the cached PTE corresponds. If a PTE's global bit is set (used for system space pages that are globally visible to all processes), the TLB entry isn't invalidated on process context switches.

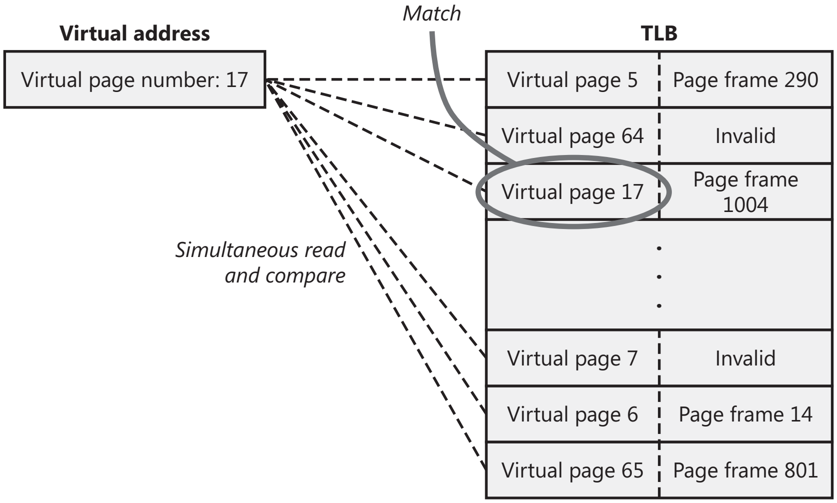


Figure 7-20 Accessing the translation look-aside buffer

Virtual addresses that are used frequently are likely to have entries in the TLB, which provides extremely fast virtual-to-physical address translation and, therefore, fast memory access. If a virtual address isn't in the TLB, it might still be in memory, but multiple memory accesses are needed to find it, which makes the access time slightly slower. If a virtual page has been paged out of memory or if the memory manager changes the PTE, the memory manager explicitly invalidates the TLB entry. If a process accesses it again, a page fault occurs and the memory manager brings the page back into memory and re-creates an entry for it in the TLB.

To maximize the amount of common code, the memory manager treats all PTEs the same whenever possible, whether they are maintained by hardware or by software. For example, the memory manager calls a kernel routine when a PTE changes from invalid to valid. The job of this routine is to load this new PTE into the TLB in whatever hardware-specific manner the architecture requires. On x86 systems, the code is a NOP because the processor loads the TLB without any intervention from the software.

Physical Address Extension (PAE)

The Intel x86 Pentium Pro processor introduced a memory-mapping mode called *Physical Address Extension* (PAE). With the proper chipset, the PAE mode allows access to up to 64 GB of physical memory on current Intel x86 processors and 1024 GB of physical memory on x64 processors (though Windows currently limits this to 128 GB due to the size of the PFN database required to map so much memory). When the processor executes in PAE mode, the memory management unit (MMU) divides virtual addresses into four fields, as shown in Figure 7-21.

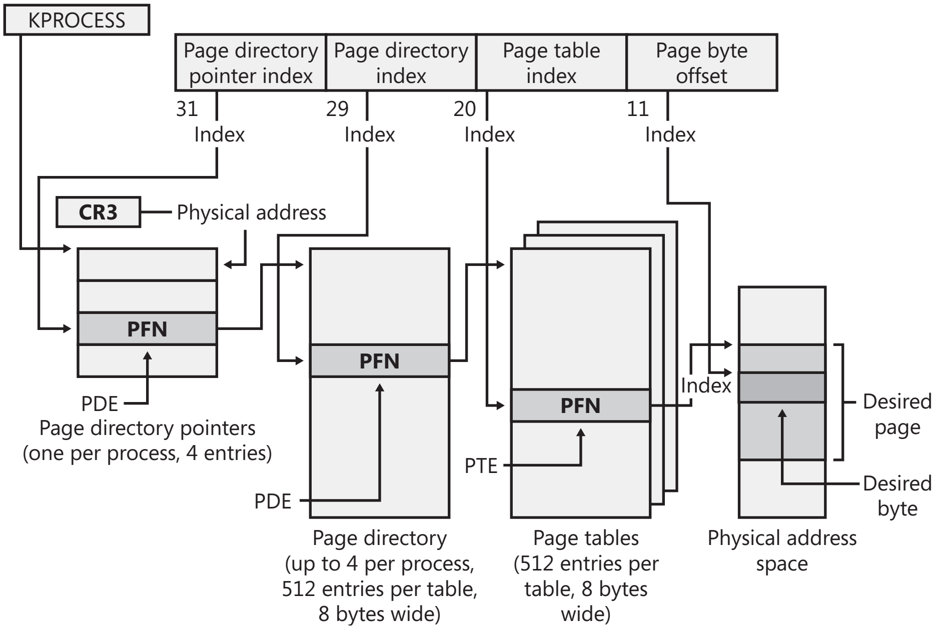

KPROCESS
Page directory pointer index
Page directory index
Page table index
Page byte offset
31
29
20
11
Index
Index
Index
Index
CR3
Physical address
PFN
PFN
PFN
Index
PDE
Page directory pointers (one per process, 4 entries)
PDE
Page directory (up to 4 per process, 512 entries per table, 8 bytes wide)
PTE
Page tables (512 entries per table, 8 bytes wide)
Physical address space
Desired page
Desired byte

Figure 7-21 Page mappings with PAE

The MMU still implements page directories and page tables, but a third level, the page directory pointer table, exists above them. PAE mode can address more memory than the standard translation mode not because of the extra level of translation but because PDEs and PTEs are 64 bits wide rather than 32 bits. The system represents physical addresses internally with 25 bits, which gives the ability to support a maximum of 2^{25+12} bytes, or 128 GB, of memory. One way in which 32-bit applications can take advantage of such large memory configurations is described in the earlier section “Address Windowing Extensions.” However, even if applications are not using such functions, the memory manager will use all available physical memory for file cache data through the use of the system cache, standby, and modified lists (described in the section “Page Frame Number Database”).

As explained in Chapter 2, there is a special version of the 32-bit Windows kernel with support for PAE called Ntkrnlpa.exe. To select this PAE-enabled kernel, you must boot with the /PAE switch in Boot.ini. Note that this special version of the kernel image is installed on all 32-bit Windows systems, even Windows 2000 Professional or Windows XP systems with small memory. The reason for this is to facilitate device driver testing. Because the PAE kernel presents 64-bit addresses to device drivers and other system code, booting /PAE even on a small memory system allows device driver developers to test parts of their drivers with large addresses. The other relevant Boot.ini switch is /NOLOWMEM, which discards memory below 4 GB (assuming you have at least 5 GB of physical memory) and relocates device drivers above this range. This guarantees that drivers will be presented with physical addresses greater than 32 bits, which makes any possible driver sign extension bugs easier to find.

IA-64 Virtual Address Translation

The virtual address space for IA-64 is divided into eight regions by the hardware. Each region can have its own set of page tables. Windows uses five of the regions, three of which have page tables. Table 7-12 lists the regions and how they are used.

Table 7-12 The IA-64 Regions

Region	Use
0	User code and data
1	Session space code and data
2	Unused
3	Unused
4	Kseg3, which is a cached 1-to-1 mapping of physical memory. No page tables are needed for this region because the necessary TLB inserts are done directly by the memory manager.
5	Kseg4, which is a noncached 1-to-1 mapping for physical memory. This is used only in a few places for accessing I/O locations such as the I/O port range. There are no page tables needed for this region.
6	Unused
7	Kernel code and data

Address translation by 64-bit Windows on the IA-64 platform uses a three-level page table scheme. Each process has a page directory pointer structure that contains 1024 pointers to page directories. Each page directory contains 1024 pointers to page tables, which in turn point to physical pages. Figure 7-22 shows the format of an IA-64 hardware PTE.

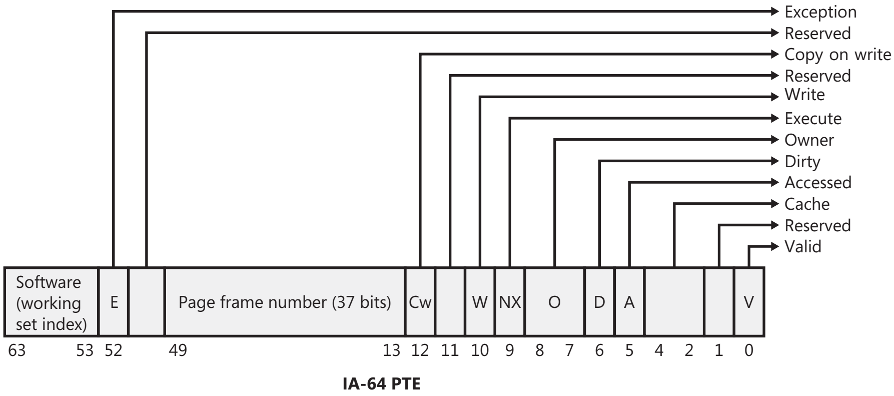


Figure 7-22 IA-64 page table entry

x64 Virtual Address Translation

64-bit Windows on the x64 architecture uses a four-level page table scheme. Each process has a top level extended page directory (called the page map level 4) that contains 512 pointers to a third-level structure called a page parent directory. Each page parent directory contains 512 pointers to second-level page directories, each of which contain 512 pointers to the individual page tables. Finally, the page tables (each of which contain 512 page table entries) point to pages in memory. Current implementations of the x64 architecture limit virtual addresses to 48 bits. The components that make up this 48-bit virtual address are shown in Figure 7-23. The connections between these structures are shown in Figure 7-24. Finally, the format of an x64 hardware page table entry is shown in Figure 7-25.

x64 64-bit (48-bit in today's processors)

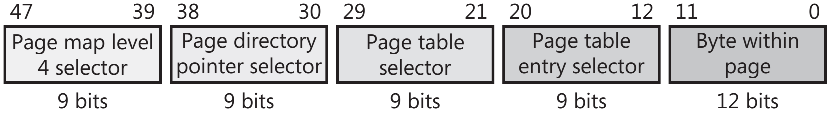


Figure 7-23 x64 virtual address

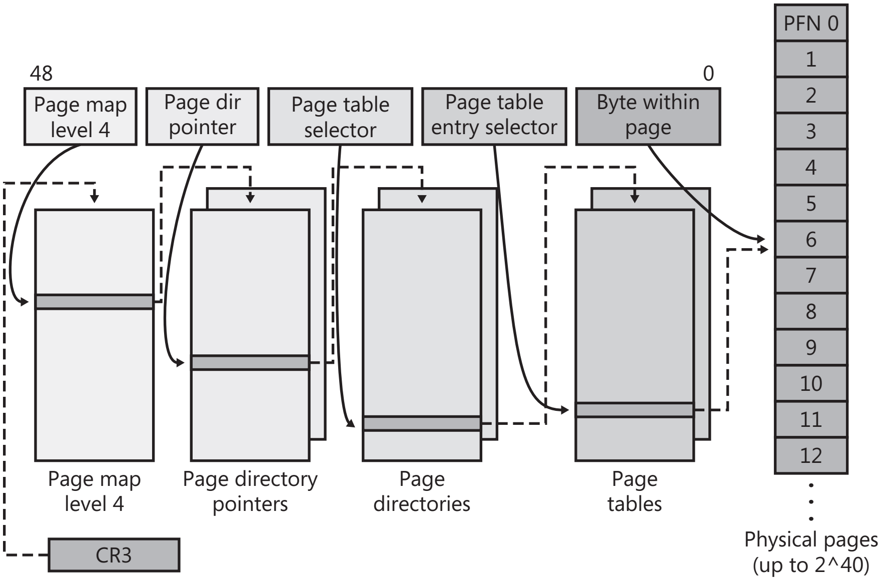


Figure 7-24 x64 address translation structures

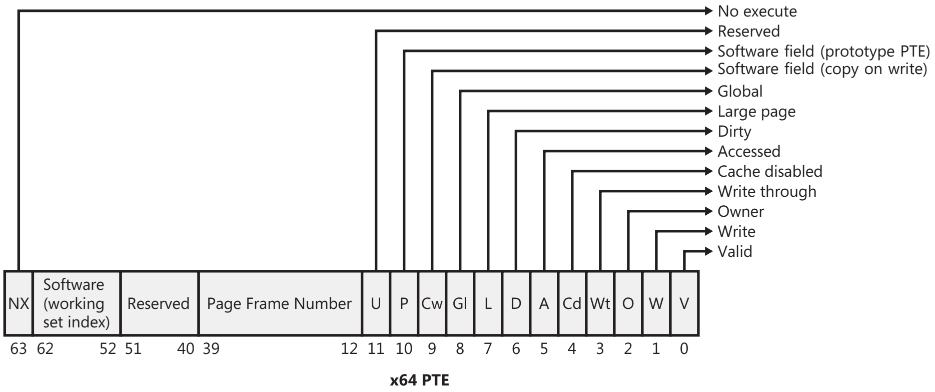


Figure 7-25 x64 hardware page table entry

Page Fault Handling

Earlier, you saw how address translations are resolved when the PTE is valid. When the PTE valid bit is clear, this indicates that the desired page is for some reason not (currently) accessible to the process. This section describes the types of invalid PTEs and how references to them are resolved.

Note Only the 32-bit x86 PTE formats are detailed in this book. PTEs for 64-bit systems contain similar information, but their detailed layout is not presented.

A reference to an invalid page is called a *page fault*. The kernel trap handler (introduced in the section "Trap Dispatching" in Chapter 3) dispatches this kind of fault to the memory manager fault handler (*MmAccessFault*) to resolve. This routine runs in the context of the thread that incurred the fault and is responsible for attempting to resolve the fault (if possible) or raise an appropriate exception. These faults can be caused by a variety of conditions, as listed in Table 7-13.

Table 7-13 Reasons for Access Faults

Reason for Fault	Result
Accessing a page that isn't resident in memory but is on disk in a page file or a mapped file	Allocate a physical page, and read the desired page from disk and into the working set
Accessing a page that is on the standby or modified list	Transition the page to the process or system working set
Accessing a page that isn't committed (for example, reserved address space or address space that isn't allocated)	Access violation
Accessing a page from user mode that can be accessed only in kernel mode	Access violation

Table 7-13 Reasons for Access Faults

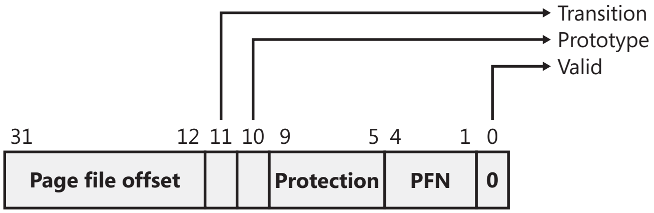
Reason for Fault	Result
Writing to a page that is read-only	Access violation
Accessing a demand-zero page	Add a zero-filled page to the process working set
Writing to a guard page	Guard-page violation (if a reference to a user-mode stack, perform automatic stack expansion)
Writing to a copy-on-write page	Make process-private (or session-private) copy of page, and replace original in process, session, or system working set
Referencing a page in system space that is valid but not in the process page directory (for example, if paged pool expanded after the process page directory was created)	Copy page directory entry from master system page directory structure, and dismiss exception
On a multiprocessor system, writing to a page that is valid but hasn't yet been written to	Set dirty bit in PTE
Executing code in a page that is marked as no execute	Access violation (supported only on hardware platforms that support no execute protection running Windows XP Service Pack 2 or Windows Server 2003 Service Pack 1 and later)

The following section describes the four basic kinds of invalid PTEs that are processed by the access fault handler. Following that is an explanation of a special case of invalid PTEs, prototype PTEs, which are used to implement shareable pages.

Invalid PTEs

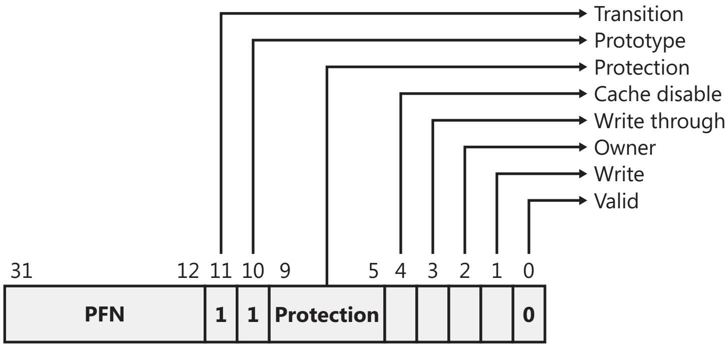
The following list details the four kinds of invalid PTEs and their structure. Some of the flags are the same as those for a hardware PTE as described in Table 7-11.

- **Page file** The desired page resides within a paging file. An in-page operation is initiated, as illustrated here:



- **Demand zero** The desired page must be satisfied with a page of zeros. The pager looks at the zero page list. If the list is empty, the pager takes a page from the free list and zeroes it. If that list is empty, it takes a page from the standby list and zeroes it. The PTE format is the same as the page file PTE shown in the previous entry, but the page file number and offset are zeros.

- **Transition** The desired page is in memory on either the standby, modified, or modified-no-write list. The page will be removed from the list and added to the working set if referenced, as shown here:



- **Unknown** The PTE is zero, or the page table doesn't yet exist. In both cases, this flag means that you should examine the virtual address descriptors (VADs) to determine whether this virtual address has been committed. If so, page tables are built to represent the newly committed address space. (See the discussion of VADs later in the chapter.)

Prototype PTEs

If a page can be shared between two processes, the memory manager relies on a software structure called *prototype page table entries* (prototype PTEs) to map these potentially shared pages. For page file backed sections, an array of prototype PTEs is created when a section object is first created; for mapped files, portions of the array are created on demand as each view is mapped. (See the following note.) These prototype PTEs are part of the *segment* structure, described at the end of this chapter.

Note In Windows 2000 and Windows 2000 Service Pack 1, the memory manager allocates all the prototype page table entries needed to map the entire file, even though applications might only map views to small parts of the file(s) at any one time. Because these structures are allocated from a finite resource (paged pool), attempting to map large files can exhaust this resource. As a result, this limits the total amount of mapped file(s) in use at any time to approximately 200 GB.

This limit has been removed in Windows 2000 Service Pack 2 and beyond by having the memory manager allocate these structures only when mapped views into the file are created. A result of this change is that it is now possible for file backup of any size file on any size system. Previously, you couldn't back up a 500-GB file on a 32-MB system because there wasn't enough paged pool to create the prototype page table entries for the entire section. Now that these structures are allocated for active views only, large file backups are supported on systems with small physical memory configurations.

When a process first references a page mapped to a view of a section object (recall that the VADs are created only when the view is mapped), the memory manager uses the information in the prototype PTE to fill in the real PTE used for address translation in the process page table. When a shared page is made valid, both the process PTE and the prototype PTE point to the physical page containing the data. To track the number of process PTEs that reference a valid shared page, a counter in the PFN database entry is incremented. Thus, the memory manager can determine when a shared page is no longer referenced by any page table and thus can be made invalid and moved to a transition list or written out to disk.

When a shareable page is invalidated, the PTE in the process page table is filled in with a special PTE that points to the prototype PTE entry that describes the page, as shown in Figure 7-26.

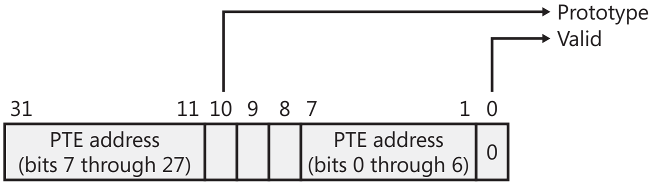

Prototype
Valid
31
11
10
9
8
7
1
0
PTE address (bits 7 through 27)
PTE address (bits 0 through 6)
0

Figure 7-26 Structure of an invalid PTE that points to the prototype PTE

Thus, when the page is later accessed, the memory manager can locate the prototype PTE using the information encoded in this PTE, which in turn describes the page being referenced. A shared page can be in one of six different states as described by the prototype PTE entry:

- **Active/valid** The page is in physical memory as a result of another process that accessed it.
- **Transition** The desired page is in memory on the standby or modified list.
- **Modified-no-write** The desired page is in memory and on the modified-no-write list. (See Table 7-20.)
- **Demand zero** The desired page should be satisfied with a page of zeros.
- **Page file** The desired page resides within a page file.
- **Mapped file** The desired page resides within a mapped file.

Although the format of these prototype PTE entries is the same as that of the real PTE entries described earlier, these prototype PTEs aren't used for address translation—they are a layer between the page table and the page frame number database and never appear directly in page tables.

By having all the accessors of a potentially shared page point to a prototype PTE to resolve faults, the memory manager can manage shared pages without needing to update the page tables of each process sharing the page. For example, a shared code or data page might be paged out to disk at some point. When the memory manager retrieves the page from disk, it needs only to update the prototype PTE to point to the page's new physical location—the PTEs in each of the processes sharing the page remain the same (with the valid bit clear and still pointing to the prototype PTE). Later, as processes reference the page, the real PTE will get updated.

Figure 7-27 illustrates two virtual pages in a mapped view. One is valid, and the other is invalid. As shown, the first page is valid and is pointed to by the process PTE and the prototype PTE. The second page is in the paging file—the prototype PTE contains its exact location. The process PTE (and any other processes with that page mapped) points to this prototype PTE.

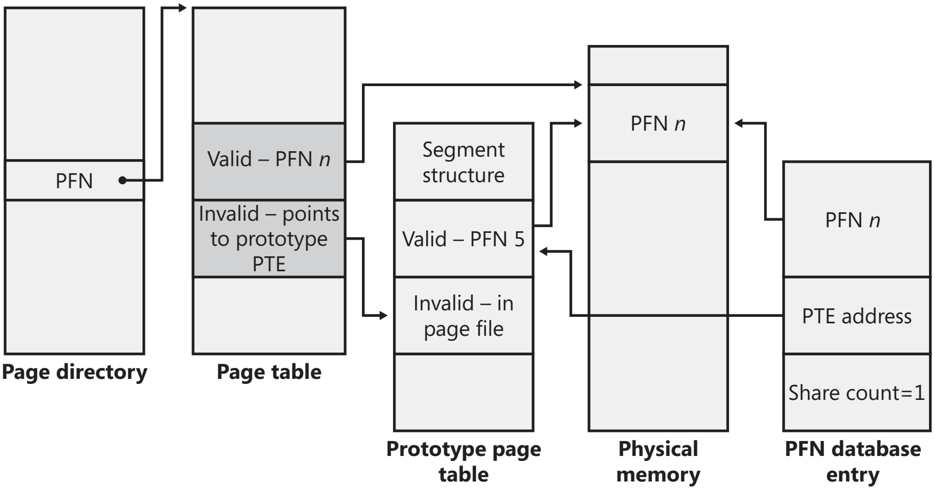


Figure 7-27 Prototype page table entries

In-Paging I/O

In-paging I/O occurs when a read operation must be issued to a file (paging or mapped) to satisfy a page fault. Also, because page tables are pageable, the processing of a page fault can incur additional page faults when the system is loading the page table page that contains the PTE or the prototype PTE that describes the original page being referenced.

The in-page I/O operation is synchronous—that is, the thread waits on an event until the I/O completes—and isn't interruptible by asynchronous procedure call (APC) delivery. The pager uses a special modifier in the I/O request function to indicate paging I/O. Upon completion of paging I/O, the I/O system triggers an event, which wakes up the pager and allows it to continue in-page processing.

While the paging I/O operation is in progress, the faulting thread doesn't own any critical memory management synchronization objects. Other threads within the process are allowed to issue virtual memory functions and handle page faults while the paging I/O takes place.

But a number of interesting conditions that the pager must recognize when the I/O completes are exposed:

- Another thread in the same process or a different process could have faulted the same page (called a *collided page fault* and described in the next section).
- The page could have been deleted (and remapped) from the virtual address space.
- The protection on the page could have changed.
- The fault could have been for a prototype PTE, and the page that maps the prototype PTE could be out of the working set.

The pager handles these conditions by saving enough state on the thread's kernel stack before the paging I/O request such that when the request is complete, it can detect these conditions and, if necessary, dismiss the page fault without making the page valid. When and if the faulting instruction is reissued, the pager is again invoked and the PTE is reevaluated in its new state.

Collided Page Faults

The case when another thread or process faults a page that is currently being in-paged is known as a *collided page fault*. The pager detects and handles collided page faults optimally because they are common occurrences in multithreaded systems. If another thread or process faults the same page, the pager detects the collided page fault, noticing that the page is in transition and that a read is in progress. (This information is in the PFN database entry.) In this case, the pager issues a wait operation on the event specified in the PFN database entry. This event was initialized by the thread that first issued the I/O needed to resolve the fault.

When the I/O operation completes, all threads waiting on the event have their wait satisfied. The first thread to acquire the PFN database lock is responsible for performing the in-page completion operations. These operations consist of checking I/O status to ensure the I/O operation completed successfully, clearing the read-in-progress bit in the PFN database, and updating the PTE.

When subsequent threads acquire the PFN database lock to complete the collided page fault, the pager recognizes that the initial updating has been performed as the read-in-progress bit is clear and checks the in-page error flag in the PFN database element to ensure that the in-page I/O completed successfully. If the in-page error flag is set, the PTE isn't updated and an in-page error exception is raised in the faulting thread.

Page Files

Page files are used to store modified pages that are still in use by some process but have had to be written to disk (because of modified page writing). Page file space is reserved when the pages are initially committed, but the actual page file locations are not chosen until pages are written out to disk. The important point is that the system commit limit is charged for private pages as they are created. Thus, the Process: Page File Bytes performance counter is actually the total process private committed memory, of which none, some, or all may be in the paging file. (In fact, it's the same as the Process: Private Bytes performance counter.)

The memory manager keeps track of private committed memory usage on a global basis, termed *commitment*, and on a per-process basis as *page file quota*. (Again, this memory usage doesn't represent page file usage—it represents private committed memory usage.) Commitment and page file quota are charged whenever virtual addresses that require new private physical pages are committed. Once the global commit limit has been reached (physical memory and the page files are full), allocating virtual memory will fail until processes free committed memory (for example, when a process exits).

When the system boots, the Session Manager process (described in Chapter 4) reads the list of page files to open by examining the registry value HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management\PagingFiles. This multistring registry value contains the name, minimum size, and maximum size of each paging file. Windows supports up to 16 paging files. On x86 systems running the normal kernel, each page file can be a maximum of 4095 MB. On x64 systems and x86 systems running the PAE kernel, each page file can be 16 terabytes (TB). On IA-64 systems, each page file can be 32 TB. Once open, the page files can't be deleted while the system is running because the System process (described in Chapter 2) maintains an open handle to each page file. The fact that the paging files are open explains why the built-in defragmentation tool cannot defragment the paging file while the system is up. To defragment your paging file, use the freeware Pagedefrag tool from *www.sysinternals.com*. It uses the same approach as other third-party defragmentation tools—it runs its defragmentation process early in the boot process before the page files are opened by the Session Manager.

Because the page file contains parts of process and kernel virtual memory, for security reasons the system can be configured to clear the page file at system shutdown. To enable this, set the registry value HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management\ClearPageFileAtShutdown to 1. Otherwise, after shutdown, the page file will contain whatever data happened to have been paged out while the system was up. This data could then be accessed by someone who gained physical access to the machine.

If no paging files are specified, Windows 2000 creates a default 20-MB page file on the boot partition. Windows XP and Windows Server 2003 do not create this temporary paging file, which means the system virtual memory commit limit is based on available memory. In Windows XP and Windows Server 2003, if the minimum and maximum paging file sizes are both zero, this indicates a system managed paging file, which causes the system to choose the page file size as shown in Table 7-14.

Table 7-14 Default Page File Sizes

System Memory Size	Minimum Page File Size	Maximum Page File Size
< 1 GB	1.5 * RAM	3 * RAM
>= 1 GB	1 * RAM	3 * RAM

EXPERIMENT: Viewing System Page Files

To view the list of page files, look in the registry at HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management\PagingFiles. This contains the paging file configuration settings modified through the System utility in Control Panel. In Windows 2000, click the Performance Options button on the Advanced tab, and then click the Change button. In Windows XP and Windows Server 2003, click the Advanced tab, click the Settings button in the Performance section, click the Advanced tab, and finally, click the Change button in the Virtual Memory section.

To add a new page file, Control Panel uses the (internal only) *NtCreatePagingFile* system service defined in Ntdll.dll. Page files are always created as noncompressed files, even if the directory they are in is compressed. To keep new page files from being deleted, a handle is duplicated into the System process so that when the creating process closes the handle to the new page file, another process can still open the page file.

The performance counters listed in Table 7-15 allow you to examine private committed memory usage on a systemwide or per-page-file basis. There's no way to determine how much of a process's private committed memory is resident and how much is paged out to paging files.

Table 7-15 Committed Memory and Page File Performance Counters

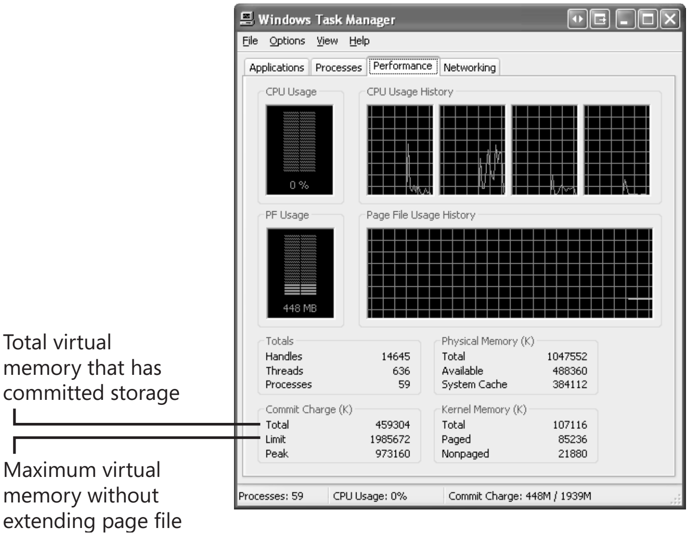
Performance Counter	Description
Memory: Committed Bytes	Number of bytes of virtual (not reserved) memory that has been committed. This number doesn't necessarily represent page file usage because it includes private committed pages in physical memory that have never been paged out. Rather, it represents the amount of page file space that would be used if the process was completely made nonresident.
Memory: Commit Limit	Number of bytes of virtual memory that can be committed without having to extend the paging files; if the paging files can be extended, this limit is soft.
Paging File: % Usage	Percentage of the paging file committed.
Paging File: % Usage Peak	Highest percentage of the paging file committed.

Note that these counters can assist you in choosing a page file size. Although most do it, basing page file size as a function of RAM makes no sense because the more memory you have, the less likely you are to need to page data out. To determine how much page file space your system really needs based on the mix of applications that have run since the system booted, examine the peak commit charge (displayed in the Commit Charge section of Task Manager's performance tab and also in Process Explorer's System Information display). This number represents the peak amount of page file space since the system booted that would have been needed if the system had to page out all private committed virtual memory (which rarely happens).

If the page file on your system is too big, the system will not use it any more or less—in other words, increasing the size of the page file does not change system performance, it simply means the system can have more nonshareable committed virtual memory. If the page file is too small for the mix of applications you are running, you might get the "system running low on virtual memory" error message. In this case, first check to see whether a process has a memory leak by examining the process private bytes count (found in the "VM Size" column in Task Manager's Processes tab). If no process appears to have a leak, check the system paged pool size—if a device driver is leaking paged pool, this might also explain the error. (See the "Troubleshooting a Pool Leak" experiment in the "System Memory Pools" section for how to troubleshoot a pool leak.)

EXPERIMENT: Viewing Page File Usage with Task Manager

You can also view committed memory usage with Task Manager by clicking its Performance tab. You'll see the following counters related to page files:



Totals

Handles	14645
Threads	636
Processes	59

Physical Memory (K)

Total	1047552
Available	488360
System Cache	384112

Commit Charge (K)

Total	459304
Limit	1985672
Peak	973160

Kernel Memory (K)

Total	107116
Paged	85236
Nonpaged	21880

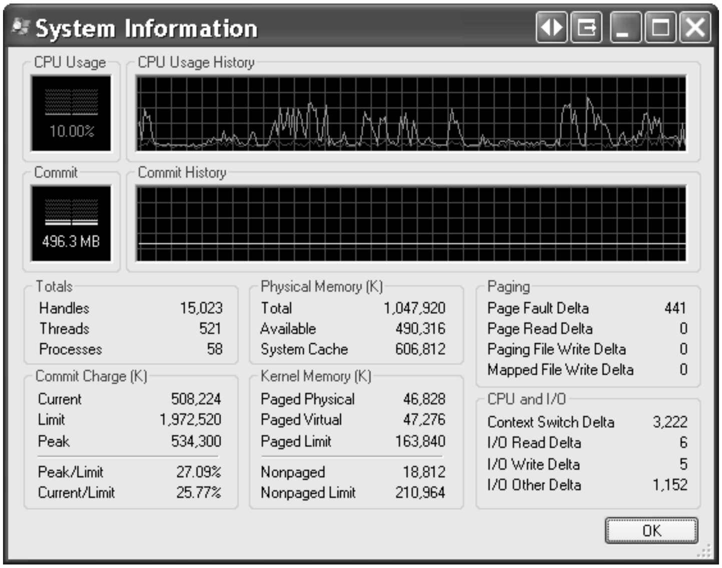
Processes: 59 CPU Usage: 0% Commit Charge: 448M / 1939M

Total virtual memory that has committed storage

Maximum virtual memory without extending page file

Note that the Mem Usage bar (called PF Usage in Windows XP and Windows Server 2003) is actually the system commit total. This number represents *potential* page file usage, not actual page file usage. It is how much page file space would be used if all the private committed virtual memory in the system had to be paged out all at once.

Process Explorer's System Information display shows an additional piece of information about system commit usage, namely the percentage of the peak as compared to the limit and the current usage as compared to the limit:



Totals	
Handles	15,023
Threads	521
Processes	58

Commit Charge (K)	
Current	508,224
Limit	1,972,520
Peak	534,300
Peak/Limit	27.09%
Current/Limit	25.77%

Physical Memory (K)	
Total	1,047,920
Available	490,316
System Cache	606,812

Kernel Memory (K)	
Paged Physical	46,828
Paged Virtual	47,276
Paged Limit	163,840
Nonpaged	18,812
Nonpaged Limit	210,964

Paging	
Page Fault Delta	441
Page Read Delta	0
Paging File Write Delta	0
Mapped File Write Delta	0

CPU and I/O	
Context Switch Delta	3,222
I/O Read Delta	6
I/O Write Delta	5
I/O Other Delta	1,152

Virtual Address Descriptors

The memory manager uses a demand-paging algorithm to know when to load pages into memory, waiting until a thread references an address and incurs a page fault before retrieving the page from disk. Like copy-on-write, demand paging is a form of *lazy evaluation*—waiting to perform a task until it is required.

The memory manager uses lazy evaluation not only to bring pages into memory but also to construct the page tables required to describe new pages. For example, when a thread commits a large region of virtual memory with *VirtualAlloc*, the memory manager could immediately construct the page tables required to access the entire range of allocated memory. But what if some of that range is never accessed? Creating page tables for the entire range would be a wasted effort. Instead, the memory manager waits to create a page table until a thread incurs a page fault, and then it creates a page table for that page. This method significantly improves performance for processes that reserve and/or commit a lot of memory but access it sparsely.

With the lazy-evaluation algorithm, allocating even large blocks of memory is a fast operation. When a thread allocates memory, the memory manager must respond with a range of addresses for the thread to use. To do this, the memory manager maintains another set of data structures to keep track of which virtual addresses have been reserved in the process's address space and which have not. These data structures are known as *virtual address descriptors* (VADs). For each process, the memory manager maintains a set of VADs that describes the status of the process's address space. VADs are structured as a self-balancing binary tree to make lookups efficient. Windows Server 2003 implements an AVL tree algorithm (named after their inventors, Adelson-Velskii and Landis) that better balances the VAD tree, resulting in, on average, fewer comparisons when searching for a VAD corresponding with a virtual address. A diagram of a VAD tree is shown in Figure 7-28.

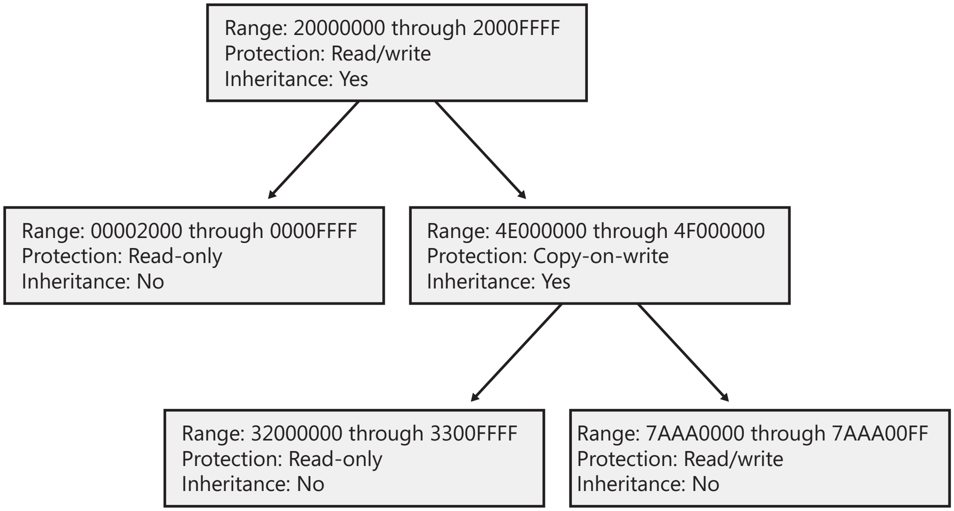

Range: 20000000 through 2000FFFF
Protection: Read/write
Inheritance: Yes
Range: 00002000 through 0000FFFF
Protection: Read-only
Inheritance: No
Range: 4E000000 through 4F000000
Protection: Copy-on-write
Inheritance: Yes
Range: 32000000 through 3300FFFF
Protection: Read-only
Inheritance: No
Range: 7AAA0000 through 7AAA00FF
Protection: Read/write
Inheritance: No

Figure 7-28 Virtual address descriptors

When a process reserves address space or maps a view of a section, the memory manager creates a VAD to store any information supplied by the allocation request, such as the range of addresses being reserved, whether the range will be shared or private, whether a child process can inherit the contents of the range, and the page protection applied to pages in the range.

When a thread first accesses an address, the memory manager must create a PTE for the page containing the address. To do so, it finds the VAD whose address range contains the accessed address and uses the information it finds to fill in the PTE. If the address falls outside the range covered by the VAD or in a range of addresses that are reserved but not committed, the memory manager knows that the thread didn't allocate the memory before attempting to use it and therefore generates an access violation.

EXPERIMENT: Viewing Virtual Address Descriptors

You can use the kernel debugger's *!vad* command to view the VADs for a given process. First find the address of the root of the VAD tree with the *!process* command. Then specify that address to the *!vad* command, as shown in the following example of the VAD tree for a process running Notepad.exe:

```
kd> !process 2a0 1
Searching for Process with Cid == 2a0
PROCESS 8614d030 SessionId: 0 Cid: 02a0 Peb: 7ffdf000 ParentCid: 0554
    DirBase: 00d93000  ObjectTable: 81bc47c8  TableSize:  41.
    Image: notepad.exe
    VadRoot 8118d868 Clone 0 Private 252. Modified 0. Locked 0.
```

```
kd> !vad 8118d868
VAD     level      start      end    commit
84df4148 ( 2)         10       10         1 Private      READWRITE
850cdbe8 ( 3)         20       20         1 Private      READWRITE
810b0ee8 ( 1)         30       6f         7 Private      READWRITE
8109d308 ( 3)         70      16f        32 Private      READWRITE
810e9a28 ( 2)        170      17f         0 Mapped       READWRITE
84aedfc8 ( 3)        180      195         0 Mapped       READONLY
8118d868 ( 0)        1a0      1ce         0 Mapped       READONLY
81190a08 ( 4)        1d0      210         0 Mapped       READONLY
85c7b928 ( 3)        220      223         0 Mapped       READONLY
86253a08 ( 4)        230      2f7         0 Mapped       EXECUTE_READ
810aab48 ( 2)        300      342         0 Mapped       READONLY
80db5448 ( 5)        350      64f         0 Mapped       EXECUTE_READ

 Total VADs:    49  average level:    6  maximum depth: 13
```

Section Objects

As you'll remember from the section on shared memory earlier in the chapter, the *section object*, which the Windows subsystem calls a *file mapping object*, represents a block of memory that two or more processes can share. A section object can be mapped to the paging file or to another file on disk.

The executive uses sections to load executable images into memory, and the cache manager uses them to access data in a cached file. (See Chapter 11 for more information on how the cache manager uses section objects.) You can also use section objects to map a file into a process address space. The file can then be accessed as a large array by mapping different views of the section object and reading or writing to memory rather than to the file (an activity called *mapped file I/O*). When the program accesses an invalid page (one not in physical memory), a page fault occurs and the memory manager automatically brings the page into memory from the mapped file. If the application modifies the page, the memory manager writes the changes back to the file during its normal paging operations (or the application can flush a view by using the Windows *FlushViewOfFile* function).

Section objects, like other objects, are allocated and deallocated by the object manager. The object manager creates and initializes an object header, which it uses to manage the objects; the memory manager defines the body of the section object. The memory manager also implements services that user-mode threads can call to retrieve and change the attributes stored in the body of section objects. The structure of a section object is shown in Figure 7-29.

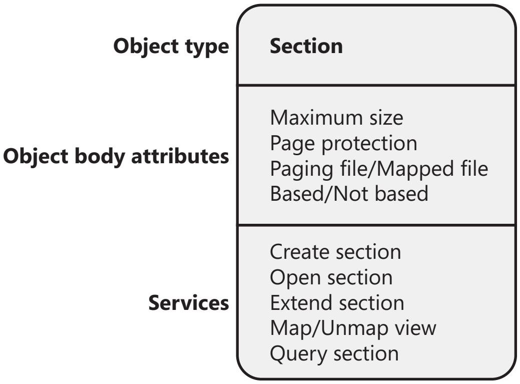


Figure 7-29 A section object

Table 7-16 summarizes the unique attributes stored in section objects.

Table 7-16 Section Object Body Attributes

Attribute	Purpose
Maximum size	The largest size to which the section can grow in bytes; if mapping a file, the maximum size is the size of the file.
Page protection	Page-based memory protection assigned to all pages in the section when it is created.
Paging file/Mapped file	Indicates whether the section is created empty (backed by the paging file—as explained earlier, page file backed sections use page-file resources only when the pages need to be written out to disk) or loaded with a file (backed by the mapped file).
Based/Not based	Indicates whether a section is a based section, which must appear at the same virtual address for all processes sharing it, or a nonbased section, which can appear at different virtual addresses for different processes.

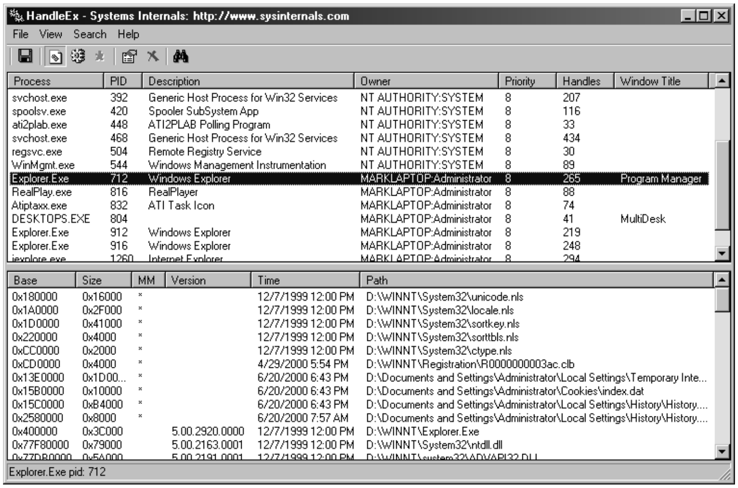
EXPERIMENT: Viewing Section Objects

With the Object Viewer (Winobj.exe from *www.sysinternals.com* from the book's companion CD or Winobj.exe in the Platform SDK), you can see the list of sections that have global names. You can list the open handles to section objects with any of the tools described in the Object Manager section in Chapter 3 that list the open handle table. (As explained in Chapter 3, these names are stored in the object manager directory \BaseNamedObjects.)

Using Process Explorer or Handles.exe (both from *www.sysinternals.com*) or the Windows resource kit Oh.exe (Open Handles) tool, you can list the open handles to section objects. For example, the following command displays all open handles to objects of type *section*, whether or not they have names. (A section must have a name only if other processes need to open it by name.)

```
c:\> oh -t section -a
00000008 System        Section  0070
0000008C smss.exe      Section  0004
000000A4 csrss.exe     Section  0004
000000A4 csrss.exe     Section  0024
000000A4 csrss.exe     Section  0038
000000A4 csrss.exe     Section  0040 \NLS\NlsSectionUnicode
000000A4 csrss.exe     Section  0044 \NLS\NlsSectionLocale
000000A4 csrss.exe     Section  0048 \NLS\NlsSectionCType
000000A4 csrss.exe     Section  004c \NLS\NlsSectionSortkey
000000A4 csrss.exe     Section  0050 \NLS\NlsSectionSortTbls
000000A0 winlogon.exe  Section  0004
000000A0 winlogon.exe  Section  0034
000000A0 winlogon.exe  Section  0168 \BaseNamedObjects\mmGlobalPnpInfo
§
```

You can also use Process Explorer from *www.sysinternals.com* to view mapped files. Select DLLs from the Lower Pane View entry of the View menu. Files marked with an asterisk in the MM column are mapped files (rather than DLLs and other files the image loader loads as modules). Here's an example:



Process	PID	Description	Owner	Priority	Handles	Window Title
svchost.exe	392	Generic Host Process for Win32 Services	NT AUTHORITY:SYSTEM	8	207	
spoolsv.exe	420	Spooler SubSystem App	NT AUTHORITY:SYSTEM	8	116	
ati2plab.exe	448	ATI2PLAB Polling Program	NT AUTHORITY:SYSTEM	8	33	
svchost.exe	468	Generic Host Process for Win32 Services	NT AUTHORITY:SYSTEM	8	434	
regsvc.exe	504	Remote Registry Service	NT AUTHORITY:SYSTEM	8	30	
WinMgmt.exe	544	Windows Management Instrumentation	NT AUTHORITY:SYSTEM	8	89	
Explorer.Exe	712	Windows Explorer	MARKLAPTOP:Administrator	8	265	Program Manager
RealPlay.exe	816	RealPlayer	MARKLAPTOP:Administrator	8	88	
Atiptaxx.exe	832	ATI Task Icon	MARKLAPTOP:Administrator	8	74	
DESKTOPS.EXE	804		MARKLAPTOP:Administrator	8	41	MultiDesk
Explorer.Exe	912	Windows Explorer	MARKLAPTOP:Administrator	8	219	
Explorer.Exe	916	Windows Explorer	MARKLAPTOP:Administrator	8	248	
iexplore.exe	1260	Internet Explorer	MARKLAPTOP:Administrator	8	294	

Base	Size	MM	Version	Time	Path
0x180000	0x16000	*		12/7/1999 12:00 PM	D:\WINNT\System32\unicode.nls
0x1A0000	0x2F000	*		12/7/1999 12:00 PM	D:\WINNT\System32\locale.nls
0x1D0000	0x41000	*		12/7/1999 12:00 PM	D:\WINNT\System32\sortkey.nls
0x220000	0x4000	*		12/7/1999 12:00 PM	D:\WINNT\System32\sorttbls.nls
0xCC0000	0x2000	*		12/7/1999 12:00 PM	D:\WINNT\System32\ctype.nls
0xCD0000	0x4000	*		4/29/2000 5:54 PM	D:\WINNT\Registration\R000000003ac.clb
0x13E0000	0x1D00...	*		6/20/2000 6:43 PM	D:\Documents and Settings\Administrator\Local Settings\Temporary Inte...
0x15B0000	0x10000	*		6/20/2000 6:43 PM	D:\Documents and Settings\Administrator\Cookies\index.dat
0x15C0000	0xB4000	*		6/20/2000 6:43 PM	D:\Documents and Settings\Administrator\Local Settings\History\History....
0x2580000	0x8000	*		6/20/2000 7:57 AM	D:\Documents and Settings\Administrator\Local Settings\History\History....
0x400000	0x3C000		5.00.2920.0000	12/7/1999 12:00 PM	D:\WINNT\Explorer.Exe
0x77F80000	0x79000		5.00.2163.0001	12/7/1999 12:00 PM	D:\WINNT\System32\ntdll.dll
0x77DB0000	0x5A000		5.00.2191.0001	12/7/1999 12:00 PM	D:\WINNT\system32\ADVAPI32.DLL

Explorer.Exe pid: 712

The data structures maintained by the memory manager that describe mapped sections are shown in Figure 7-30. These structures ensure that data read from mapped files is consistent, regardless of the type of access (open file, mapped file, and so on).

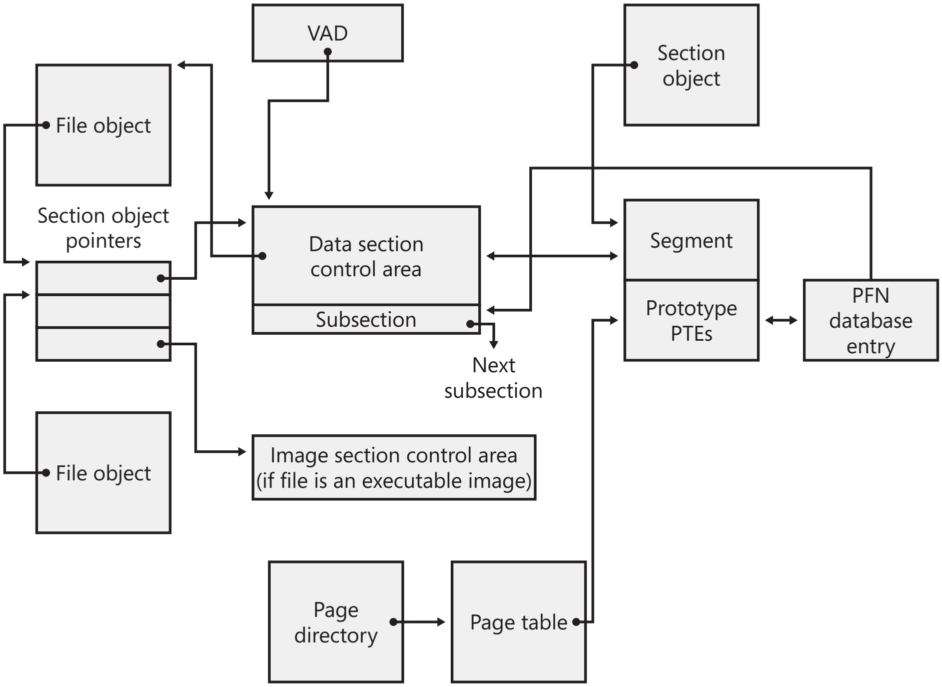


Figure 7-30 Internal section structures

For each open file (represented by a file object), there is a single *section object pointers* structure. This structure is the key to maintaining data consistency for all types of file access as well as to providing caching for files. The section object pointers structure points to one or two *control areas*. One control area is used to map the file when it is accessed as a data file, and one is used to map the file when it is run as an executable image.

A control area in turn points to *subsection* structures that describe the mapping information for each section of the file (read-only, read-write, copy-on-write, and so on). The control area also points to a *segment* structure allocated in paged pool, which in turn points to the prototype PTEs used to map to the actual pages mapped by the section object. As described earlier in the chapter, process page tables point to these prototype PTEs, which in turn map the pages being referenced.

Although Windows ensures that any process that accesses (reads or writes) a file will always see the same, consistent data, there is one case in which two copies of pages of a file can reside in physical memory (but even in this case, all accessors get the latest copy and data consistency is maintained). This duplication can happen when an image file has been accessed as a data file (having been read or written) and then run as an executable image (for example, when an image is linked and then run—the linker had the file open for data access, and then when the image was run, the image loader mapped it as an executable). Internally, the following actions occur:

1. If the executable file was created using the file mapping APIs (or the cache manager), a data control area is created to represent the data pages in the image file being read or written.
2. When the image is run and the section object is created to map the image as an executable, the memory manager finds that the section object pointers for the image file point to a data control area and flushes the section. This step is necessary to ensure that any modified pages have been written to disk before accessing the image through the image control area.
3. The memory manager then creates a control area for the image file.
4. As the image begins execution, its (read-only) pages are faulted in from the image file.

Because the pages mapped by the data control area might still be resident (on the standby list), this is the one case in which two copies of the same data are in two different pages in memory. However, this duplication doesn't result in a data consistency issue because, as mentioned, the data control area has already been flushed to disk, so the pages read from the image are up to date (and these pages are never written back to disk).

EXPERIMENT: Viewing Control Areas

To find the address of the control area structures for a file, you must first get the address of the file object in question. You can obtain this address through the kernel debugger by dumping the process handle table with the *!handle* command and noting the object address of a file object. Although the kernel debugger *!file* command displays the basic information in a file object, it doesn't display the pointer to the section object pointers structure. Then, using the *dt* command, format the file object to get the address of the section object pointers structure. This structure consists of three pointers: a pointer to the data control area, a pointer to the shared cache map (explained in Chapter 11), and a pointer to the image control area. From the section object pointers structure, you can obtain the address of a control area for the file (if one exists) and feed that address into the *!ca* command.

For example, if you open a PowerPoint file and display the handle table for that process using *!handle*, you will find an open handle to the PowerPoint file as shown below. (For information on using *!handle*, see the “Object Manager” section in Chapter 3.)

```
lkd> !handle 1 f 5c4 file
.
.
0284: Object: 8645c038  GrantedAccess: 00120089
Object: 8645c038  Type: (867ddca0) File
    ObjectHeader: 8645c020
        HandleCount: 1  PointerCount: 1
        Directory Object: 00000000  Name: \slides\ntint\new\
3-systemarchitecture.ppt {HarddiskVolume1}
```

Taking the file object address (8645c038) and formatting it with *dt* results in this:

```
lkd> dt nt!_file_object 8645c038
nt!_FILE_OBJECT
   +0x000 Type             : 5
   +0x002 Size             : 112
   +0x004 DeviceObject     : 0x86742e30
   +0x008 Vpb              : 0x867a3090
   +0x00c FsContext        : 0xe2786530
   +0x010 FsContext2       : 0xe309a378
   +0x014 SectionObjectPointer : 0x85512fec
```

Then taking the address of the section object pointers structure (0x85512fec) and formatting it with *dt* results in this:

```
lkd> dt nt!_section_object_pointers 0x85512fec
nt!_SECTION_OBJECT_POINTERS
   +0x000 DataSectionObject : 0x8564f8a0
   +0x004 SharedCacheMap   : (null)
   +0x008 ImageSectionObject : (null)
```

Finally, use *!ca* to display the control area using the address:

```
lkd> !ca 0x8564f8a0
ControlArea @8564f8a0
  Segment:    e3817528    Flink             0   Blink               0
  Section Ref        1    Pfn Ref         25c   Mapped Views        1
  User Ref           2    WaitForDel        0   Flush Count         0
  File Object 85774580    ModWriteCount     0   System Views        0
  Flags (9008080) File WasPurged HadUserReference Accessed
  File: \slides\ntint\new\3-systemarchitecture.ppt
```

Another technique is to display the list of all control areas with the *!memusage* command. The following excerpt is from the output of this command:

```
kd> !memusage
 loading PFN database
loading (99% complete)
             Zeroed:      9 (    36 kb)
               Free:      0 (     0 kb)
            Standby:   2103 (  8412 kb)
           Modified:    300 (  1200 kb)
    ModifiedNoWrite:      1 (     4 kb)
       Active/Valid:  30318 (121272 kb)
         Transition:      3 (    12 kb)
            Unknown:      0 (     0 kb)
              TOTAL:  32734 (130936 kb)
  Building kernel map
  Finished building kernel map
  Usage Summary (in Kb):
Control Valid Standby Dirty Shared Locked PageTables  name
8119b608  3000  528     0     0     0     0  mapped_file( WINWORD.EXE )
849e7c68    96    0     0     0     0     0    No Name for File
8109c388    24    0     0     0     0     0  mapped_file( DEVDTG.PKG )
81402488   236    0     0     0     0     0    No Name for File
80fba0a8   268    0     0     0     0     0  mapped_file( kernel32.pdb )
810ab168  1504  380     0     0     0     0  mapped_file( OUTLLIB.DLL )
81126a08     0  276     0     0     0     0  mapped_file( H )
81112d28   656    0     0     0     0     0    No Name for File
```

The Control column points to the control area structure that describes the mapped file. You can display control areas, segments, and subsections with the kernel debugger *!ca* command. For example, to dump the control area for the mapped file Winword.exe in this example, type the *!ca* command followed by the Control number, as shown here:

```
kd> !ca 8119b608

ControlArea @8119b608
  Segment:    e2c28000  Flink             0  Blink:            0
  Section Ref        1  Pfn Ref         372  Mapped Views:     1
  User Ref           2  Subsections       6  Flush Count:      0
  File Object 8213fb98  ModWriteCount     0  System Views:     0
  WaitForDel         0  Paged Usage    3000  NonPaged Usage  100
  Flags (90000a0) Image File HadUserReference Accessed

   File: \Program Files\Microsoft Office\Office\WINWORD.EXE

Segment @ e2c28000:
   Base address        0  Total Ptes       86a  NonExtendPtes:      86a
   Image commit       d7  ControlArea 8119b608  SizeOfSegment: 86a000
   Image Base          0  Committed          0  PTE Template:  919b6c38
   Based Addr   30000000  ProtoPtes   e2c28038  Image Info:    e2c2a1e4

Subsection 1. @ 8119b640
   ControlArea: 8119b608  Starting Sector 0 Number Of Sectors 8
```

```
     Base Pte     e2c28038  Ptes In subsect        1 Unused Ptes          0
     Flags              15  Sector Offset          0 Protection           1
      ReadOnly CopyOnWrite

  Subsection 2. @ 8119b660
     ControlArea: 8119b608  Starting Sector 10 Number Of Sectors 3c00
     Base Pte     e2c2803c  Ptes In subsect      780 Unused Ptes          0
     Flags              35  Sector Offset          0 Protection           3
      ReadOnly CopyOnWrite

  Subsection 3. @ 8119b680
     ControlArea: 8119b608  Starting Sector 3C10 Number Of Sectors 5d8
     Base Pte     e2c29e3c  Ptes In subsect       c1 Unused Ptes          0
     Flags              55  Sector Offset          0 Protection           5
      ReadOnly CopyOnWrite

  Subsection 4. @ 8119b6a0
     ControlArea: 8119b608  Starting Sector 41E8 Number Of Sectors a8
     Base Pte     e2c2a140  Ptes In subsect       15 Unused Ptes          0
     Flags              55  Sector Offset          0 Protection           5
      ReadOnly CopyOnWrite

  Subsection 5. @ 8119b6c0
     ControlArea: 8119b608  Starting Sector 0 Number Of Sectors 0
     Base Pte     e2c2a194  Ptes In subsect        1 Unused Ptes          0
     Flags              55  Sector Offset          0 Protection           5
      ReadOnly CopyOnWrite
   Subsection 6. @ 8119b6e0
     ControlArea: 8119b608  Starting Sector 4290 Number Of Sectors 90
     Base Pte     e2c2a198  Ptes In subsect       12 Unused Ptes          0
     Flags              15  Sector Offset          0 Protection           1
      ReadOnly CopyOnWrite
```

Working Sets

In the last several sections, we've concentrated on the virtual view of a Windows process—page tables, PTEs, and VADs. In the remainder of this chapter, we'll explain how Windows keeps a subset of virtual addresses in physical memory.

As you'll recall, the term used to describe a subset of virtual pages resident in physical memory is called a *working set.* There are three kinds of working sets:

- Process working sets contain the pages referenced by threads within a single process.
- The system working set contains the resident subset of the pageable system code (for example, Ntoskrnl.exe and drivers), paged pool, and the system cache.
- On systems with terminal services enabled, each session has a working set that contains the resident subset of the kernel-mode session-specific data structures allocated by the kernel-mode part of the Windows subsystem (Win32k.sys), session paged pool, session mapped views, and other session-space device drivers.

Before examining the details of each type of working set, let's look at the overall policy for deciding which pages are brought into physical memory and how long they remain. After that, we'll explore the various types of working sets.

Demand Paging

The Windows memory manager uses a demand-paging algorithm with clustering to load pages into memory. When a thread receives a page fault, the memory manager loads into memory the faulted page plus a small number of pages preceding and/or following it. This strategy attempts to minimize the number of paging I/Os a thread will incur. Because programs, especially large ones, tend to execute in small regions of their address space at any given time, loading clusters of virtual pages reduces the number of disk reads. For page faults that reference data pages in images, the cluster size is 3 pages. For all other page faults, the cluster size is 7 pages.

However, a demand paging policy can result in a process incurring many page faults when its threads first begin executing or when they resume execution at a later point. To optimize the startup of a process (and the system), Windows XP and Windows Server 2003 have an intelligent prefetch engine called the *logical prefetcher*, described in the next section.

Logical Prefetcher

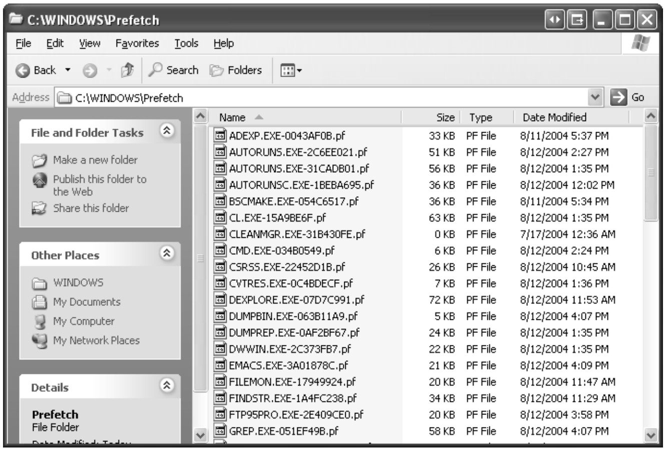
During a typical system boot or application startup, the order of faults is such that some pages are brought in from one part of a file, then perhaps from a distant part of the same file, then from a different file, perhaps from a directory, and then again from the first file. This jumping around slows down each access considerably and, thus, analysis shows that disk seek times are a dominant factor in slowing boot and application startup times. By prefetching batches of pages all at once, a more sensible ordering of access, without excessive backtracking, can be achieved, thus improving the overall time for system and application startup. The pages that are needed can be known in advance because of the high correlation in accesses across boots or application starts.

The prefetcher, introduced in Windows XP, tries to speed the boot process and application startup by monitoring the data and code accessed by boot and application startups and using that information at the beginning of a subsequent boot or application startup to read in the code and data. When the prefetcher is active, the memory manager notifies the prefetcher code in the kernel of page faults, both those that require that data be read from disk (hard faults) and those that simply require data already in memory be added to a process's working set (soft faults). The prefetcher monitors the first 10 seconds of application startup. For boot, the prefetcher by default traces from system start through the 30 seconds following the start of the user's shell (typically Explorer) or, failing that, up through 60 seconds following Windows service initialization or through 120 seconds, whichever comes first.

The trace assembled in the kernel notes faults taken on the NTFS Master File Table (MFT) metadata file (if the application accesses files or directories on NTFS volumes), on referenced files, and on referenced directories. With the trace assembled, the kernel prefetcher code notifies the prefetcher component of the Task Scheduler service (\Windows\System32\Schedsvc.dll), running in a copy of Svchost, by signaling an event object named *PrefetchTracesReady*.

Note You can enable or disable prefetching of the boot or application startups by editing the DWORD registry value HKLM\System\CurrentControlSet\Control\Session Manager\Memory Management\PrefetchParameters\EnablePrefetcher. Set it to 0 to disable prefetching altogether, 1 to enable prefetching of only applications, 2 for prefetching of boot only, and 3 for both boot and applications.

When the *PrefetchTracesReady* event is signaled, the Task Scheduler performs a call to the internal *NtQuerySystemInformation* system call requesting the trace data. The Task Scheduler post processes the trace data, combining it with previously collected data, and writes it out to a file in the \Windows\Prefetch folder, which is shown in Figure 7-31. The file's name is the name of the application to which the trace applies followed by a dash and the hexadecimal representation of a hash of the file's path. The file has a ".pf" extension, so an example would be NOTEPAD.EXE-AF43252301.PF.



Name	Size	Type	Date Modified
ADEXP.EXE-0043AF0B.pf	33 KB	PF File	8/11/2004 5:37 PM
AUTORUNS.EXE-2C6EE021.pf	51 KB	PF File	8/12/2004 2:27 PM
AUTORUNS.EXE-31CADB01.pf	56 KB	PF File	8/12/2004 1:35 PM
AUTORUNSC.EXE-1BEBA695.pf	36 KB	PF File	8/12/2004 12:02 PM
BSCMAKE.EXE-054C6517.pf	36 KB	PF File	8/11/2004 5:34 PM
CL.EXE-15A9BE6F.pf	63 KB	PF File	8/12/2004 1:35 PM
CLEANMGR.EXE-31B430FE.pf	0 KB	PF File	7/17/2004 12:36 AM
CMD.EXE-034B0549.pf	6 KB	PF File	8/12/2004 2:24 PM
CSRSS.EXE-22452D1B.pf	26 KB	PF File	8/12/2004 10:45 AM
CVTRES.EXE-0C4BDECF.pf	7 KB	PF File	8/12/2004 1:36 PM
DEXPLORE.EXE-07D7C991.pf	72 KB	PF File	8/12/2004 11:53 AM
DUMPBIN.EXE-063B11A9.pf	5 KB	PF File	8/12/2004 4:07 PM
DUMPREP.EXE-0AF2BF67.pf	24 KB	PF File	8/12/2004 1:35 PM
DWWIN.EXE-2C373FB7.pf	22 KB	PF File	8/12/2004 1:35 PM
EMACS.EXE-3A01878C.pf	21 KB	PF File	8/12/2004 4:09 PM
FILEMON.EXE-17949924.pf	20 KB	PF File	8/12/2004 11:47 AM
FINDSTR.EXE-1A4FC238.pf	34 KB	PF File	8/12/2004 11:29 AM
FTP95PRO.EXE-2E409CE0.pf	20 KB	PF File	8/12/2004 3:58 PM
GREP.EXE-051EF49B.pf	58 KB	PF File	8/12/2004 4:07 PM

Figure 7-31 Prefetch folder

There are two exceptions to the file name rule. The first is for images that host other components, including the Microsoft Management Console (\Windows\System32\Mmc.exe) and Dllhost (\Windows\System32\Dllhost.exe). Because add-on components are specified on the command line for these applications, the prefetcher includes the command line in the generated hash. Thus, invocations of these applications with different components on the command line will result in different traces. The prefetcher reads the list of executables that it should treat this way from the *HostingAppList* value in its parameters registry key, HKLM\System\CurrentControlSet\Control\Session Manager\Memory Management\PrefetchParameters.

The other exception to the file name rule is the file that stores the boot's trace, which is always named NTOSBOOT-B00DFAAD.PF. (If read as a word, "boodfaad" sounds similar to the English words *boot fast*.) Only after the prefetcher has finished the boot trace (the time of which was defined earlier) does it collect page fault information for specific applications.

EXPERIMENT: Looking Inside a Prefetch File

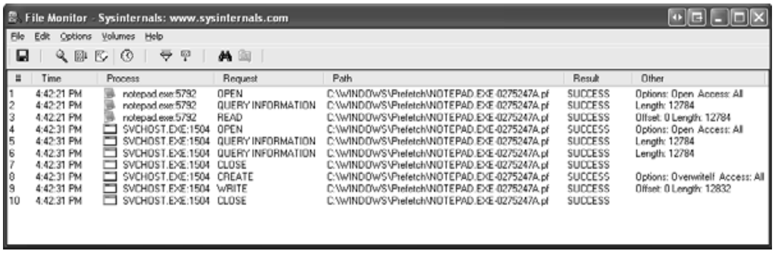
A prefetch file's contents serve as a record of files and directories accessed during the boot or an application startup, and you can use the Strings utility from *www.sysinternals.com* to see the record. The following command lists all the files and directories referenced during the last boot:

```
C:\Windows\Prefetch>Strings ntosboot-boodfaad.pf

Strings v2.1
Copyright (C) 1999-2003 Mark Russinovich
Systems Internals - www.sysinternals.com

NTOSBOOT
SCCA
\DEVICE\HARDDISKVOLUME2\$MFT
\DEVICE\HARDDISKVOLUME2\WINDOWS\PREFETCH\NTOSBOOT-B00DFAAD.PF
\DEVICE\HARDDISKVOLUME2\SYSTEM VOLUME INFORMATION\_RESTORE{987E0331-0F01-427C-A5
8A-7A2E4AABF84D}\RP24\CHANGE.LOG
\DEVICE\HARDDISKVOLUME2\WINDOWS\SYSTEM32\DRIVERS\PROCESSR.SYS
\DEVICE\HARDDISKVOLUME2\WINDOWS\SYSTEM32\DRIVERS\FGLRYM.SYS
\DEVICE\HARDDISKVOLUME2\WINDOWS\SYSTEM32\DRIVERS\VIDEOPRT.SYS
\DEVICE\HARDDISKVOLUME2\WINDOWS\SYSTEM32\DRIVERS\E1000325.SYS
\DEVICE\HARDDISKVOLUME2\WINDOWS\SYSTEM32\DRIVERS\USBUHCI.SYS
\DEVICE\HARDDISKVOLUME2\WINDOWS\SYSTEM32\DRIVERS\USBPORT.SYS
\DEVICE\HARDDISKVOLUME2\WINDOWS\SYSTEM32\DRIVERS\USBEHCI.SYS
\DEVICE\HARDDISKVOLUME2\WINDOWS\SYSTEM32\DRIVERS\NIC1394.SYS
…
```

When the system boots or an application starts, the prefetcher is called to give it an opportunity to perform prefetching. The prefetcher looks in the prefetch directory to see if a trace file exists for the prefetch scenario in question. If it does, the prefetcher calls NTFS to prefetch any MFT metadata file references, reads in the contents of each of the directories referenced, and finally opens each file referenced. It then calls the memory manager function *MmPrefetchPages* to read in any data and code specified in the trace that's not already in memory. The memory manager initiates all the reads asynchronously and then waits for them to complete before letting an application's startup continue.

EXPERIMENT: Watching Prefetch File Reads and Writes

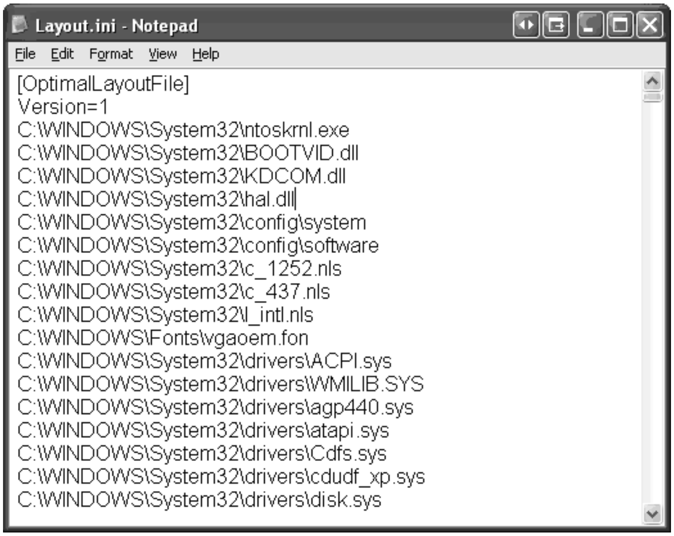
If you capture a trace of application startup with Filemon from *www.sysinternals.com* in Windows XP, you can see the prefetcher check for and read the application's prefetch file (if it exists), and roughly ten seconds after the application started, see the prefetcher write out a new copy of the file. Below is a capture of Notepad startup with an Include filter set to "prefetch" so that Filemon shows only accesses to the \Windows\Prefetch directory:



#	Time	Process	Request	Path	Result	Other
1	4:42:21 PM	notepad.exe:5792	OPEN	C:\WINDOWS\Prefetch\NOTEPAD.EXE-0275247A.pf	SUCCESS	Options: Open Access: All
2	4:42:21 PM	notepad.exe:5792	QUERY INFORMATION	C:\WINDOWS\Prefetch\NOTEPAD.EXE-0275247A.pf	SUCCESS	Length: 12784
3	4:42:21 PM	notepad.exe:5792	READ	C:\WINDOWS\Prefetch\NOTEPAD.EXE-0275247A.pf	SUCCESS	Offset: 0 Length: 12784
4	4:42:31 PM	SVCHOST.EXE:1504	OPEN	C:\WINDOWS\Prefetch\NOTEPAD.EXE-0275247A.pf	SUCCESS	Options: Open Access: All
5	4:42:31 PM	SVCHOST.EXE:1504	QUERY INFORMATION	C:\WINDOWS\Prefetch\NOTEPAD.EXE-0275247A.pf	SUCCESS	Length: 12784
6	4:42:31 PM	SVCHOST.EXE:1504	QUERY INFORMATION	C:\WINDOWS\Prefetch\NOTEPAD.EXE-0275247A.pf	SUCCESS	Length: 12784
7	4:42:31 PM	SVCHOST.EXE:1504	CLOSE	C:\WINDOWS\Prefetch\NOTEPAD.EXE-0275247A.pf	SUCCESS	
8	4:42:31 PM	SVCHOST.EXE:1504	CREATE	C:\WINDOWS\Prefetch\NOTEPAD.EXE-0275247A.pf	SUCCESS	Options: OverwriteIf Access: All
9	4:42:31 PM	SVCHOST.EXE:1504	WRITE	C:\WINDOWS\Prefetch\NOTEPAD.EXE-0275247A.pf	SUCCESS	Offset: 0 Length: 12832
10	4:42:31 PM	SVCHOST.EXE:1504	CLOSE	C:\WINDOWS\Prefetch\NOTEPAD.EXE-0275247A.pf	SUCCESS	

Lines 1 through 3 show the Notepad prefetch file being read in the context of the Notepad process during its startup. Lines 4 through 10, which have time stamps 10 seconds later than the first 3 lines, show the Task Scheduler, which is running in the context of a Svchost process, write out the updated prefetch file.

To minimize seeking even further, every three days or so, during system idle periods, the Task Scheduler organizes a list of files and directories in the order that they are referenced during a boot or application start and stores the list in a file named Windows\Prefech\Layout.ini, shown in Figure 7-32.



```
[OptimalLayoutFile]
Version=1
C:\WINDOWS\System32\ntoskrnl.exe
C:\WINDOWS\System32\BOOTVID.dll
C:\WINDOWS\System32\KDCOM.dll
C:\WINDOWS\System32\hal.dll
C:\WINDOWS\System32\config\system
C:\WINDOWS\System32\config\software
C:\WINDOWS\System32\c_1252.nls
C:\WINDOWS\System32\c_437.nls
C:\WINDOWS\System32\l_intl.nls
C:\WINDOWS\Fonts\vgaoem.fon
C:\WINDOWS\System32\drivers\ACPI.sys
C:\WINDOWS\System32\drivers\WMILIB.SYS
C:\WINDOWS\System32\drivers\agp440.sys
C:\WINDOWS\System32\drivers\atapi.sys
C:\WINDOWS\System32\drivers\Cdfs.sys
C:\WINDOWS\System32\drivers\cdudf_xp.sys
C:\WINDOWS\System32\drivers\disk.sys
```

Figure 7-32 Prefetch defragmentation layout file

Then it launches the system defragmenter with a command-line option that tells the defragmenter to defragment based on the contents of the file instead of performing a full defrag. The defragmenter finds a contiguous area on each volume large enough to hold all the listed files and directories that reside on that volume and then moves them in their entirety into the area so that they are stored one after the other. Thus, future prefetch operations will even be more efficient because all the data read in is now stored physically on the disk in the order it will be read. Because the files defragmented for prefetching usually number only in the hundreds, this defragmentation is much faster than full volume defragmentations. (See Chapter 12 for more information on defragmentation.)

Placement Policy

When a thread receives a page fault, the memory manager must also determine where in physical memory to put the virtual page. The set of rules it uses to determine the best position is called a *placement policy*. Windows considers the size of CPU memory caches when choosing page frames to minimize unnecessary thrashing of the cache.

If physical memory is full when a page fault occurs, a *replacement policy* is used to determine which virtual page must be removed from memory to make room for the new page. Common replacement policies include *least recently used* (LRU) and *first in, first out* (FIFO). The LRU algorithm (also known as the *clock algorithm*, as implemented in most versions of UNIX) requires the virtual memory system to track when a page in memory is used. When a new page frame is required, the page that hasn't been used for the greatest amount of time is removed from the working set. The FIFO algorithm is somewhat simpler; it removes the page that has been in physical memory for the greatest amount of time, regardless of how often it's been used.

Replacement policies can be further characterized as either global or local. A global replacement policy allows a page fault to be satisfied by any page frame, whether or not that frame is owned by another process. For example, a global replacement policy using the FIFO algorithm would locate the page that has been in memory the longest and would free it to satisfy a page fault; a local replacement policy would limit its search for the oldest page to the set of pages already owned by the process that incurred the page fault. Global replacement policies make processes vulnerable to the behavior of other processes—an ill-behaved application can undermine the entire operating system by inducing excessive paging activity in all processes.

Windows implements a combination of local and global replacement policy. When a working set reaches its limit and/or needs to be trimmed because of demands for physical memory, the memory manager removes pages from working sets until it has determined there are enough free pages.

Working Set Management

Every process starts with a default working set minimum of 50 pages and a working set maximum of 345 pages. Although it has little effect, you can change the process working set limits with the Windows *SetProcessWorkingSetSize* function, though you must have the "increase scheduling priority" user right to do this. However, unless you have configured the process to use hard working set limits (new in Windows Server 2003), these limits are ignored, in that the memory manager will permit a process to grow beyond its maximum if it is paging heavily and there is ample memory (and conversely, the memory manager will shrink a process below its working set minimum if it is not paging and there is a high demand for physical memory on the system). Although in Windows 2000, the extent to which a process was over its working set maximum was a factor in its priority to be trimmed, as of Windows XP, this decision is based solely on how many pages have been accessed.

In Windows Server 2003, hard working set limits can be set using the *SetProcessWorkingSetSizeEx* function along with the QUOTA_LIMITS_HARDWS_ENABLE flag. An sample consumer of this function is the Windows System Resource Manager (WSRM), described in Chapter 6.

The maximum working set size can't exceed the systemwide maximum calculated at system initialization time and stored in the kernel variable *MmMaximumWorkingSetSize*. This value is set to be the number of available pages (the size of the zero, free, and standby list) at the time the computation is made minus 512 pages. However, there are hard upper limits for working set sizes—these are listed in Table 7-17.

Table 7-17 Upper Limit for Working Set Maximums

Windows Version	Working Set Maximum
x86 versions of Windows 2000, Windows XP, Windows XP SP1, Windows Server 2003	1984 MB
x86 versions of Windows XP SP2, Windows Server 2003 SP1	2047.9 MB
x86 versions of Windows booted /3GB	3008 MB
IA-64	7152 GB
x64	8192 GB

When a page fault occurs, the process's working set limits and the amount of free memory on the system are examined. If conditions permit, the memory manager allows a process to grow to its working set maximum (or beyond if the process does not have a hard working set limit and there are enough free pages available). However, if memory is tight, Windows replaces rather than adds pages in a working set when a fault occurs.

Although Windows attempts to keep memory available by writing modified pages to disk, when modified pages are being generated at a very high rate, more memory is required in order to meet memory demands. Therefore, when physical memory runs low, the *working set manager*, a routine that runs in the context of the balance set manager system thread (described in the next section), initiates automatic working set trimming to increase the amount of free memory available in the system. (With the Windows *SetProcessWorkingSetSize* function mentioned earlier, you can also initiate working set trimming of your own process—for example, after process initialization.)

The working set manager examines available memory and decides which, if any, working sets need to be trimmed. If there is ample memory, the working set manager calculates how many pages could be removed from working sets if needed. If trimming is needed, it looks at working sets that are above their minimum setting. It also dynamically adjusts the rate at which it examines working sets as well as arranges the list of processes that are candidates to be trimmed into an optimal order. For example, processes with many pages that have not been accessed recently are examined first; larger processes that have been idle longer are considered before smaller processes that are running more often; the process running the foreground application is considered last; and so on.

When it finds processes using more than their minimums, the working set manager looks for pages to remove from their working sets, making the pages available for other uses. If the amount of free memory is still too low, the working set manager continues removing pages from processes' working sets until it achieves a minimum number of free pages on the system.

On a single-processor Windows 2000 system and all systems running Windows XP and Windows Server 2003, the working set manager tries to remove pages that haven't been accessed recently. It does this by checking the accessed bit in the hardware PTE to see whether the page has been accessed. If the bit is clear, the page is *aged*, that is, a count is incremented indicating that the page hasn't been referenced since the last working set trim scan. Later, the age of pages is used to locate candidate pages to remove from the working set.

Note On a Windows 2000 multiprocessor system, the working set manager mistakenly didn't check the access bit, resulting in pages being removed from the working set without regard to the state of the accessed bit.

If the hardware PTE accessed bit is set, the working set manager clears it and goes on to examine the next page in the working set. In this way, if the accessed bit is clear the next time the working set manager examines the page, it knows that the page hasn't been accessed since the last time it was examined. This scan for pages to remove continues through the working set list until either the number of desired pages has been removed or the scan has returned to the starting point. (The next time the working set is trimmed, the scan picks up where it left off last.)

EXPERIMENT: Viewing Process Working Set Sizes

You can use the Performance tool to examine process working set sizes by looking at the following performance counters:

Counter	Description
Process: Working Set	Current size of the selected process's working set in bytes
Process: Working Set Peak	Peak size of the selected process's working set in bytes
Process: Page Faults/Sec	Number of page faults for the process that occur each second

Several other process viewer utilities (such as Task Manager, Pview, and Pviewer) also display the process working set size.

You can also get the total of all the process working sets by selecting the _Total process in the instance box in the Performance tool. This process isn't real—it's simply a total of the process-specific counters for all processes currently running on the system. The total you see is misleading, however, because the size of each process working set includes pages being shared by other processes. Thus, if two or more processes share a page, the page is counted in each process's working set.

EXPERIMENT: Viewing the Working Set List

You can view the individual entries in the working set by using the kernel debugger *!wsle* command. The following example shows a partial output of the working set list of LiveKd. (This command was run on the LiveKd process.)

```
kd> !wsle 7

Working Set @ c0502000
    Quota:      9f  FirstFree:  40  FirstDynamic:      3
    LastEntry  1fe  NextSlot:    3  LastInitialized  257
    NonDirect   5c  HashTable:   0  HashTableSize:     0

Virtual Address  Age  Locked  ReferenceCount
c0300203          0       1        1
c0301203          0       1        1
c0502203          0       1        1
c01df201          0       0        1
c01ff201          0       0        1
c0005201          0       0        1
c0001201          0       0        1
c0002201          0       0        1
c0000201          0       0        1
c0006201          0       0        1
77e87119          0       0        1
```

```
00402319         0        0        1
77e01201         0        0        1
7ffdf201         0        0        1
00130201         0        0        1
77e9e119         0        0        1
78033201         0        0        1
00230221         0        0        1
00131201         0        0        1
77d50119         0        0        1
00132201         0        0        1
c01e0201         0        0        1
00411309         0        0        1
0040d201         0        0        1
77edf201         0        0        1
77ee0201         0        0        1
77fcd201         0        0        1
0040e201         0        0        1
7ffc1009         0        0        1
00401319         0        0        1
```

Notice that some entries in the working set list are page table pages (the ones with addresses greater than 0xC0000000), some are from system DLLs (the ones in the 0x7nnnnnnn range), and some are from the code of LiveKd.exe itself (those in the 0x004nnnnn range).

Balance Set Manager and Swapper

Working set expansion and trimming take place in the context of a system thread called the *balance set manager* (routine *KeBalanceSetManager*). The balance set manager is created during system initialization. Although the balance set manager is technically part of the kernel, it calls the memory manager's working set manager to perform working set analysis and adjustment.

The balance set manager waits for two different event objects: an event that is signaled when a periodic timer set to fire once per second expires and an internal working set manager event that the memory manager signals at various points when it determines that working sets need to be adjusted. For example, if the system is experiencing a high page fault rate or the free list is too small, the memory manager wakes up the balance set manager so that it will call the working set manager to begin trimming working sets. When memory is more plentiful, the working set manager will permit faulting processes to gradually increase the size of their working sets by faulting pages back into memory, but the working sets will grow only as needed.

When the balance set manager wakes up as the result of its 1-second timer expiring, it takes the following four steps:

1. Every fourth time the balance set manager wakes up because its 1-second timer has expired, it signals an event that wakes up another system thread called the swapper (routine *KeSwapProcessOrStack*).

2. The balance set manager then checks the look-aside lists and adjusts their depths if necessary (to improve access time and to reduce pool usage and pool fragmentation).
3. It looks for threads that might warrant having their priority boosted because they are CPU starved. (See the section "Priority Boosts for CPU Starvation" in Chapter 6.)
4. It calls the memory manager's working set manager. (The working set manager has its own internal counters that regulate when to perform working set trimming and how aggressively to trim.)

The swapper is also awakened by the scheduling code in the kernel if a thread that needs to run has its kernel stack swapped out or if the process has been swapped out. The swapper looks for threads that have been in a wait state for 7 seconds in Windows 2000 and 15 seconds in Windows XP and Windows Server 2003. If it finds one, it puts the thread's kernel stack in transition (moving the pages to the modified or standby lists) so as to reclaim its physical memory, operating on the principle that if a thread's been waiting that long, it's going to be waiting even longer. When the last thread in a process has its kernel stack removed from memory, the process is marked to be entirely outswapped. That's why, for example, processes that have been idle for a long time (such as Winlogon is after you log on) can have a zero working set size.

System Working Set

Just as processes have working sets, the pageable code and data in the operating system are managed by a single *system working set*. Five different kinds of pages can reside in the system working set:

- System cache pages
- Paged pool
- Pageable code and data in Ntoskrnl.exe
- Pageable code and data in device drivers
- System mapped views

You can examine the size of the system working set or the size of the five components that contribute to it with the performance counters or system variables shown in Table 7-18. Keep in mind that the performance counter values are in bytes whereas the system variables are measured in terms of pages.

You can also examine the paging activity in the system working set by examining the Memory: Cache Faults/Sec performance counter, which describes page faults that occur in the system working set (both hard and soft).

Table 7-18 **System Working Set Performance Counters**

Performance Counter (in Bytes)	System Variable (in Pages)	Description
Memory: Cache Bytes*	*MmSystemCacheWs.WorkingSetSize*	Total size of system working set (including the cache, paged pool, pageable Ntoskrnl and driver code, and system mapped views); this is *not* the size of the system cache alone, even though the name implies that it is.
Memory: Cache Bytes Peak	*MmSystemCacheWs.Peak*	Peak system working set size.
Memory: System Cache Resident Bytes	*MmSystemCachePage*	Physical memory consumed by the system cache.
Memory: System Code Resident Bytes	*MmSystemCodePage*	Physical memory consumed by pageable code in Ntoskrnl.exe.
Memory: System Driver Resident Bytes	*MmSystemDriverPage*	Physical memory consumed by pageable device driver code.
Memory: Pool Paged Resident Bytes	*MmPagedPoolPage*	Physical memory consumed by paged pool.

* Internally, this working set is called the system *cache* working set, even though the system cache is just one of five different components in it. Thus, several utilities think they are displaying the size of the file cache when they are displaying the total size of the system working set.

The system variable that contains the value for this counter is *MmSystemCacheWs.PageFaultCount*.

The minimum and maximum system working set size is computed at system initialization time based on the amount of physical memory on the machine and whether the system is running a client or server edition of Windows. The calculated working set minimum and maximum are stored in the kernel variables shown in Table 7-19. These variables aren't available through any performance counter, but you can examine them with the kernel debugger.

Table 7-19 **System Variables That Store Working Set Minimums or Maximums**

Variable	Type	Description
MmSystemCacheWs.MinimumWorkingSetSize	ULONG	Minimum working set size
MmSystemCacheWs.MaximumWorkingSetSize	ULONG	Maximum working set size

You can configure the system to give priority to the system working set (as opposed to process working sets) by changing the value of HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management\LargeSystemCache. In Windows 2000 Server systems, you can adjust this value indirectly by setting the properties of the file server service; in Windows XP and Windows Server 2003, you can adjust this by going to My Computer, choosing Properties, clicking the Advanced tab, pressing the Settings button in the Performance section, and finally clicking the Advanced tab. (See Chapter 11 for details.)

Page Frame Number Database

Whereas working sets describe the resident pages owned by a process or the system, the *page frame number (PFN) database* describes the state of each page in physical memory. The page states are listed in Table 7-20.

Table 7-20 Page States

Status	Description
Active (also called Valid)	The page is part of a working set (either a process working set or the system working set) or it's not in any working set (for example, nonpaged kernel page), and a valid PTE points to it.
Transition	A temporary state for a page that isn't owned by a working set and isn't on any paging list. A page is in this state when an I/O to the page is in progress. The PTE is encoded so that collided page faults can be recognized and handled properly. (Note that this use of the term "transition" differs from the use of the word in the section on Invalid PTEs; an invalid transition PTE refers to a page on the standby or modified list.)
Standby	The page previously belonged to a working set but was removed. The page wasn't modified since it was last written to disk. The PTE still refers to the physical page but is marked invalid and in transition.
Modified	The page previously belonged to a working set but was removed. However, the page was modified while it was in use and its current contents haven't yet been written to disk. The PTE still refers to the physical page but is marked invalid and in transition. It must be written to disk before the physical page can be reused.
Modified no-write	Same as a modified page, except that it has been marked so that the memory manager's modified page writer won't write it to disk. The cache manager marks pages as modified no-write at the request of file system drivers. For example, NTFS uses this state for pages containing file system metadata so that it can first ensure that transaction log entries are flushed to disk before the pages they are protecting are written to disk. (NTFS transaction logging is explained in Chapter 12.)
Free	The page is free but has unspecified dirty data in it. (These pages can't be given as a user page to a user process without being initialized with zeros, for security reasons.)
Zeroed	The page is free and has been initialized with zeros by the zero page thread.
Rom	The page has been faulted in from read-only memory (new as of Windows XP).
Bad	The page has generated parity or other hardware errors and can't be used.

The PFN database consists of an array of structures that represent each physical page of memory on the system. The PFN database and its relationship to page tables are shown in

Figure 7-33. As this figure shows, valid PTEs point to entries in the PFN database, and the PFN database entries (for nonprototype PFNs) point back to the page table that is using them. For prototype PFNs, they point back to the prototype PTE.

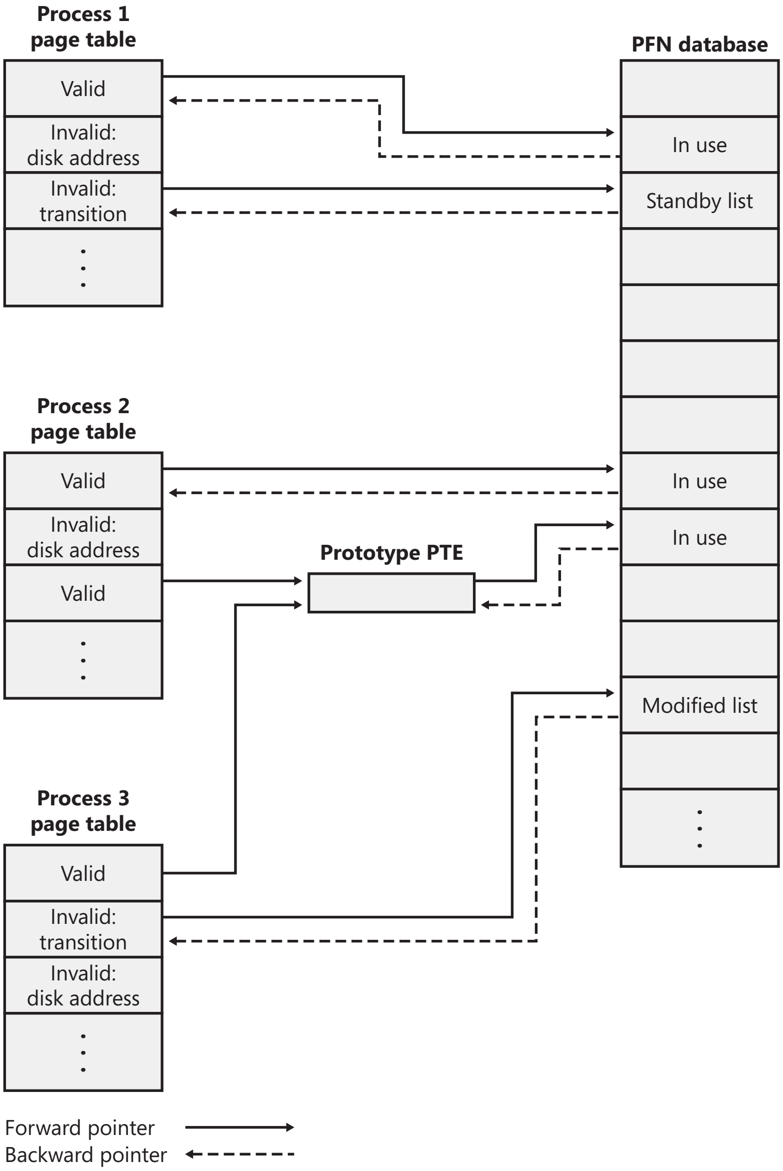


Figure 7-33 Page tables and the page frame number database

Of the page states listed in Table 7-20, six are organized into linked lists so that the memory manager can quickly locate pages of a specific type. (Active/valid pages and transition pages aren't in any systemwide page list.) Figure 7-34 shows an example of how these entries are linked together.

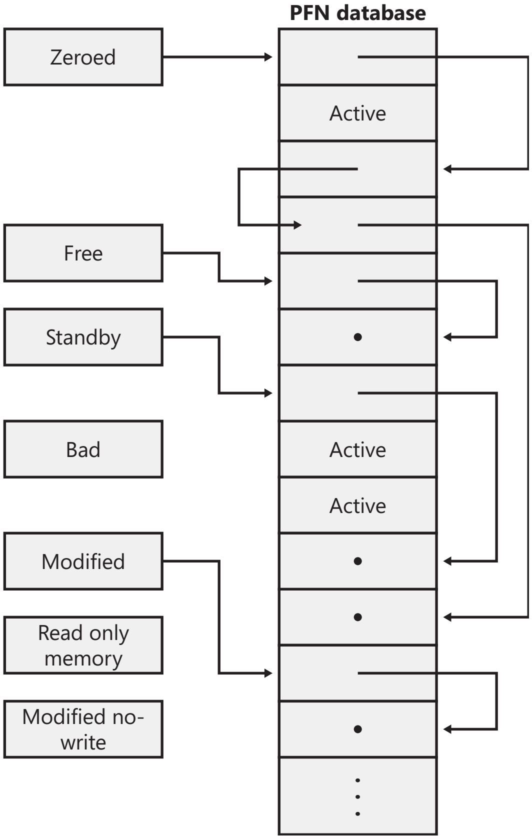


Figure 7-34 Page lists in the PFN database

In the next section, you'll find out how these linked lists are used to satisfy page faults and how pages move to and from the various lists.

EXPERIMENT: Viewing the PFN Database

Using the kernel debugger *!memusage* command, you can dump the size of the various paging lists. The following is the output from this command:

```
lkd> !memusage
 loading PFN database
loading (100% complete)
Compiling memory usage data (99% Complete).
             Zeroed:   8474 ( 33896 kb)
               Free:    256 (  1024 kb)
            Standby:  50790 (203160 kb)
           Modified:    496 (  1984 kb)
    ModifiedNoWrite:      0 (     0 kb)
       Active/Valid: 201980 (807920 kb)
         Transition:      1 (     4 kb)
            Unknown:      0 (     0 kb)
              TOTAL: 261997 (1047988 kb)
```

Page List Dynamics

Figure 7-35 shows a state diagram for page frame transitions. For simplicity, the modified-no-write list isn't shown.

Page frames move between the paging lists in the following ways:

- When the memory manager needs a zero-initialized page to service a demand-zero page fault (a reference to a page that is defined to be all zeros or to a user-mode committed private page that has never been accessed), it first attempts to get one from the zero page list. If the list is empty, it gets one from the free page list and zeroes the page. If the free list is empty, it goes to the standby list and zeroes that page.

  One reason zero-initialized pages are required is to meet C2 security requirements. C2 specifies that user-mode processes must be given initialized page frames to prevent them from reading a previous process's memory contents. Therefore, the memory manager gives user-mode processes zeroed page frames unless the page is being read in from a mapped file. If that's the case, the memory manager prefers to use nonzeroed page frames, initializing them with the data off the disk.

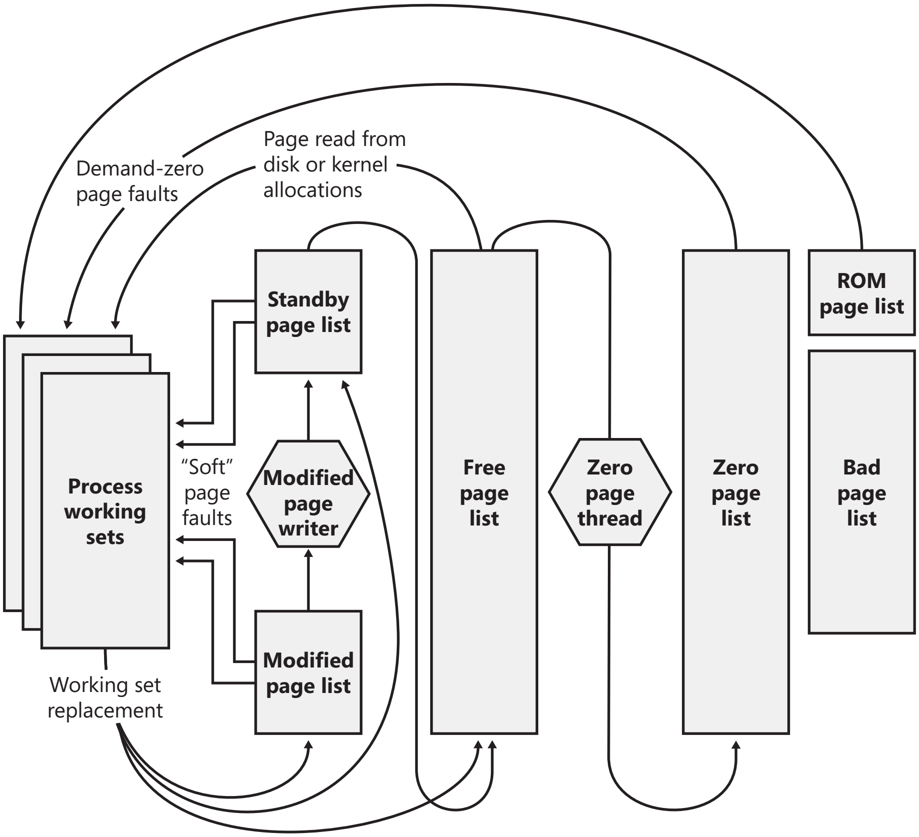


Figure 7-35 State diagram for page frames

The zero page list is populated from the free list by a system thread called the *zero page thread* (thread 0 in the System process). The zero page thread waits on an event object to signal it to go to work. When the free list has eight or more pages, this event is signaled. However, the zero page thread will run only if no other threads are running, because the zero page thread runs at priority 0 and the lowest priority that a user thread can be set to is 1.

Note In Windows Server 2003 and later, when memory needs to be zeroed as a result of a physical page allocation by a driver that calls *MmAllocatePagesForMdl* or a Windows application that calls *AllocateUserPhysicalPages*, or when an application allocates large pages, the memory manager zeroes the memory using a higher performing function called *MiZeroInParallel*, which maps larger regions than the zero page thread, which only zeroes a page at a time. In addition, on multiprocessor systems, it creates additional system threads to perform the zeroing in parallel (and in a NUMA-optimized fashion on NUMA platforms).

- When the memory manager doesn’t require a zero-initialized page, it goes first to the free list. If that’s empty, it goes to the zeroed list. If the zeroed list is empty, it goes to the standby list. Before the memory manager can use a page frame from the standby list, it must first backtrack and remove the reference from the invalid PTE (or prototype PTE) that still points to the page frame. Because entries in the PFN database contain pointers back to the previous user’s page table (or to a prototype PTE for shared pages), the memory manager can quickly find the PTE and make the appropriate change.
- When a process has to give up a page out of its working set (either because it referenced a new page and its working set was full or the memory manager trimmed its working set), the page goes to the standby list if the page was clean (not modified) or to the modified list if the page was modified while it was resident. When a process exits, all the private pages go to the free list. Also, when the last reference to a page file backed section is closed, these pages also go to the free list.

EXPERIMENT: Viewing Page Fault Behavior

With the Pfmon tool (in the Windows 2000 and 2003 resource kits, as well as in the Windows XP Support Tools), you can watch page fault behavior as it occurs. A *soft fault* refers to a page fault satisfied from one of the transition lists. *Hard faults* refer to a disk-read. The following example is a portion of output you’ll see if you start Notepad with Pfmon and then exit. Be sure to notice the summary of page fault activity at the end.

```
C:\> pfmon notepad
SOFT: KiUserApcDispatcher : KiUserApcDispatcher
SOFT: LdrInitializeThunk : LdrInitializeThunk
SOFT: 0x77f61016 :  : 0x77f61016
SOFT: 0x77f6105b :  : fltused+0xe00
HARD: 0x77f6105b :  : fltused+0xe00
SOFT: LdrQueryImageFileExecutionOptions :
                        LdrQueryImageFileExecutionOptions
SOFT: RtlAppendUnicodeToString : RtlAppendUnicodeToString
SOFT: RtlInitUnicodeString : RtlInitUnicodeString

  notepad Caused   8 faults had   9 Soft  5 Hard faulted VA’s
    ntdll Caused  94 faults had  42 Soft  8 Hard faulted VA’s
 comdlg32 Caused   3 faults had   0 Soft  3 Hard faulted VA’s
  shlwapi Caused   2 faults had   2 Soft  2 Hard faulted VA’s
    gdi32 Caused  18 faults had  10 Soft  2 Hard faulted VA’s
 kernel32 Caused  48 faults had  36 Soft  3 Hard faulted VA’s
   user32 Caused  38 faults had  26 Soft  6 Hard faulted VA’s
 advapi32 Caused   7 faults had   6 Soft  3 Hard faulted VA’s
   rpcrt4 Caused   6 faults had   4 Soft  2 Hard faulted VA’s
 comctl32 Caused   6 faults had   5 Soft  2 Hard faulted VA’s
  shell32 Caused   6 faults had   5 Soft  2 Hard faulted VA’s
          Caused  10 faults had   9 Soft  5 Hard faulted VA’s
 winspool Caused   4 faults had   2 Soft  2 Hard faulted VA’s

 PFMON: Total Faults 250
       (KM 74 UM 250 Soft 204, Hard 46, Code 121, Data 129)
```

Modified Page Writer

When the modified list gets too big, or if the size of the zeroed and standby lists falls below a minimum threshold (as indicated by the kernel variable *MmMinimumFreePages*, which is computed at system boot time), one of two system threads are awakened to write pages back to disk and move the pages to the standby list. One system thread writes out modified pages (*MiModifiedPageWriter*) to the paging file, and a second one writes modified pages to mapped files (*MiMappedPageWriter*). Two threads are required to avoid creating a deadlock, which would occur if the writing of mapped file pages caused a page fault that in turn required a free page when no free pages were available (thus requiring the modified page writer to create more free pages). By having the modified page writer perform mapped file paging I/Os from a second system thread, that thread can wait without blocking regular page file I/O.

Both threads run at priority 17 and, after initialization, wait for separate event objects to trigger their operation. The modified page writer event is triggered for one of two reasons:

- When the number of modified pages exceeds the maximum value computed at system initialization (*MmModifiedPageMaximum*)—currently 800 pages for all systems
- When the number of available pages (*MmAvailablePages*) goes below *MmMinimumFreePages*

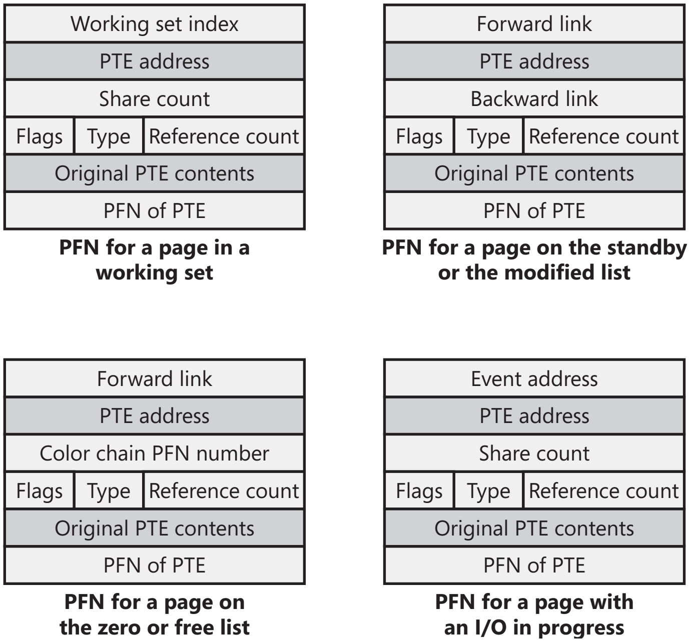
The modified page writer waits for an additional event (*MiMappedPagesTooOldEvent*) that is set after a predetermined number of seconds (*MmModifiedPageLifeInSeconds*) to indicate that mapped pages (not modified pages) should be written to disk. By default, this value is 300 seconds (5 minutes). (You can override this value by adding the DWORD registry value HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management\ModifiedPageLife). The reason for this additional event is to reduce data loss in the case of a system crash or power failure by eventually writing out modified mapped pages even if the modified list hasn't reached its threshold of 800 pages.

When invoked, the mapped page writer attempts to write as many pages as possible to disk with a single I/O request. It accomplishes this by examining the original PTE field of the PFN database elements for pages on the modified page list to locate pages in contiguous locations on the disk. Once a list is created, the pages are removed from the modified list, an I/O request is issued, and at successful completion of the I/O request, the pages are placed at the tail of the standby list.

Pages that are in the process of being written can be referenced by another thread. When this happens, the reference count and the share count in the PFN entry that represents the physical page are incremented to indicate that another process is using the page. When the I/O operation completes, the modified page writer notices that the share count is no longer 0 and doesn't place the page on the standby list.

PFN Data Structures

Although PFN database entries are of fixed length, they can be in several different states, depending on the state of the page. Thus, individual fields have different meanings depending on the state. The states of a PFN entry are shown in Figure 7-36.



PFN for a page in a working set

PFN for a page on the standby or the modified list

PFN for a page on the zero or free list

PFN for a page with an I/O in progress

Figure 7-36 States of PFN database entries

Several fields are the same for several PFN types, but others are specific to a given type of PFN. The following fields appear in more than one PFN type:

- **PTE address** Virtual address of the PTE that points to this page.
- **Reference count** The number of references to this page. The reference count is incremented when a page is first added to a working set and/or when the page is locked in memory for I/O (for example, by a device driver). The reference count is decremented when the share count becomes 0 or when pages are unlocked from memory. When the share count becomes 0, the page is no longer owned by a working set. Then, if the reference count is also zero, the PFN database entry that describes the page is updated to add the page to the free, standby, or modified list.
- **Type** The type of page represented by this PFN. (Types include active/valid, transition, standby, modified, modified-no-write, free, zeroed, bad, and transition.)
- **Flags** The information contained in the flags field is shown in Table 7-21.

- **Original PTE contents** All PFN database entries contain the original contents of the PTE that pointed to the page (which could be a prototype PTE). Saving the contents of the PTE allows it to be restored when the physical page is no longer resident.
- **PFN of PTE** Physical page number of the page table page containing the PTE that points to this page.

Table 7-21 Flags Within PFN Database Entries

Flag	Meaning
Modified state	Indicates whether the page was modified. (If the page is modified, its contents must be saved to disk before removing it from memory.)
Prototype PTE	Indicates that the PTE referenced by the PFN entry is a prototype PTE. (For example, this page is shareable.)
Parity error	Indicates that the physical page contains parity or error correction control errors.
Read in progress	Indicates that an in-page operation is in progress for the page. The first DWORD contains the address of the event object that will be signaled when the I/O is complete; also used to indicate the first PFN for nonpaged pool allocations.
Write in progress	Indicates that a page write operation is in progress. The first DWORD contains the address of the event object that will be signaled when the I/O is complete; also used to indicate the last PFN for nonpaged pool allocations.
Start of nonpaged pool	For nonpaged pool pages, indicates that this is the first PFN for a given nonpaged pool allocation.
End of nonpaged pool	For nonpaged pool pages, indicates that this is the last PFN for a given nonpaged pool allocation.
In-page error	Indicates that an I/O error occurred during the in-page operation on this page. (In this case, the first field in the PFN contains the error code.)

The remaining fields are specific to the type of PFN. For example, the first PFN in Figure 7-36 represents a page that is active and part of a working set. The share count field represents the number of PTEs that refer to this page. (Pages marked read-only, copy-on-write, or shared read/write can be shared by multiple processes.) For page table pages, this field is the number of valid PTEs in the page table. As long as the share count is greater than 0, the page isn't eligible for removal from memory.

The working set index field is an index into the process working set list (or the system or session working set list, or zero if not in any working set) where the virtual address that maps this physical page resides. If the page is a private page, the working set index field refers directly to the entry in the working set list because the page is mapped only at a single virtual address. In the case of a shared page, the working set index is a hint that is guaranteed to be correct only for the first process that made the page valid. (Other processes will try to use the same index where possible.) The process that initially sets this field is guaranteed to refer to the proper index and doesn't need to add a working set list hash entry referenced by the virtual address into its working set hash tree. This guarantee reduces the size of the working set hash tree and makes searches faster for these particular direct entries.

The second PFN in Figure 7-36 is for a page on either the standby or the modified list. In this case, the forward and backward link fields link the elements of the list together within the list. This linking allows pages to be easily manipulated to satisfy page faults. When a page is on one of the lists, the share count is by definition 0 (because no working set is using the page) and therefore can be overlaid with the backward link. The reference count is also 0 if the page is on one of the lists. If it is nonzero (because an I/O could be in progress for this page—for example, when the page is being written to disk), it is first removed from the list.

The third PFN in Figure 7-36 is for a page on the free or zeroed list. Besides being linked together within the two lists, these PFN database entries use an additional field to link physical pages by "color," their location in the processor CPU memory cache. Windows attempts to minimize unnecessary thrashing of CPU memory caches by using different physical pages in the CPU cache. It achieves this optimization by avoiding using the same cache entry for two different pages wherever possible. For systems with direct mapped caches, optimally using the hardware's capabilities can result in a significant performance advantage.

The fourth PFN in Figure 7-36 is for a page that has an I/O in progress (for example, a page read). While the I/O is in progress, the first field points to an event object that will be signaled when the I/O completes. If an in-page error occurs, this field contains the Windows error status code representing the I/O error. This PFN type is used to resolve collided page faults.

EXPERIMENT: Viewing PFN Entries

You can examine individual PFN entries with the kernel debugger *!pfn* command. You first need to supply the PFN as an argument. (For example, *!pfn 0* shows the first entry, *!pfn 1* shows the second, and so on.) In the following example, the PTE for virtual address 0x50000 is displayed, followed by the PFN that contains the page directory, and then the actual page:

```
kd> !pte 50000
00050000  - PDE at C0300000        PTE at C0000140
          contains 00700067      contains 00DAA047
        pfn 00700 --DA--UWV    pfn 00DAA --D---UWV
kd> !pfn 700
  PFN 00000700 at address 827CD800
  flink  00000004 blink / share count 00000010 pteaddress C0300000
  reference count 0001                                 color 0
  restore pte 00000080  containing page        00030  Active     M
  Modified

kd> !pfn daa
  PFN 00000DAA at address 827D77F0
```

```
flink  00000077 blink / share count 00000001 pteaddress C0000140
reference count 0001                                 color 0
restore pte 00000080  containing page        00700  Active     M
Modified
```

In addition to the PFN database, the system variables in Table 7-22 describe the overall state of physical memory.

Table 7-22 System Variables That Describe Physical Memory

Variable	Description
MmNumberOfPhysicalPages	Total number of physical pages available on the system
MmAvailablePages	Total number of available pages on the system—the sum of the pages on the zeroed, free, and standby lists
MmResidentAvailablePages	Total number of physical pages that would be available if every process were at its minimum working set size

Low and High Memory Notification

Windows XP and Windows Server 2003 provide a way for user mode processes to be notified when physical memory is low and/or plentiful. This information can be used to determine memory usage as appropriate. For example, if available memory is low, the application can reduce memory consumption. If available memory is high, the application can allocate more memory.

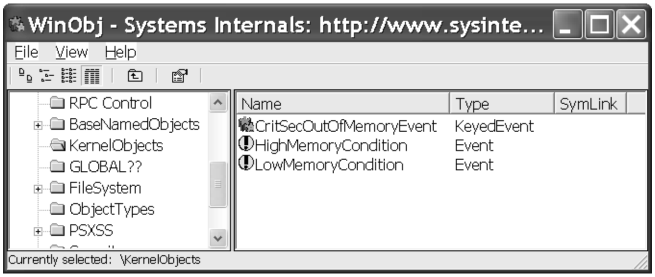
To be notified of low or high memory conditions, call the *CreateMemoryResourceNotification* function, specifying whether low or high memory notification is desired. A handle can be provided to any of the wait functions. When memory is low (or high), the wait completes, thus notifying the thread of the condition. Alternatively, the *QueryMemoryResourceNotification* can be used to query the system memory condition at any time.

Notification is implemented by the memory manager signaling a globally named event object *LowMemoryCondition* or *HighMemoryCondition*. These named objects are not in the normal \BaseNamedObjects object manager directory, but in a special directory called \KernelObjects. When low (or high) memory condition is detected, the appropriate event is signaled, thus waking up any waiting threads.

The default level of available memory that signals a low-memory-resource notification event is approximately 32 MB per 4 GB, to a maximum of 64 MB. The default level that signals a high-memory-resource notification event is three times the default low-memory value. These values can be overridden by adding a DWORD registry value *LowMemoryThreshold* or *HighMemoryThreshold* under HKEY_LOCAL_MACHINE\System\CurrentControlSet\Session Manager\Memory Management that specifies the number of megabytes to use as the low or high threshold.

EXPERIMENT: Viewing the Memory Resource Notification Events

To see the memory resource notification events, run Winobj from *www.sysinternals.com* and click on the KernelObjects folder. You will see both the low and high memory condition events shown on the right pane:



Name	Type	SymLink
CritSecOutOfMemoryEvent	KeyedEvent	
HighMemoryCondition	Event	
LowMemoryCondition	Event	

Currently selected: \KernelObjects

If you double-click either event, you can see how many handles and/or references have been made to the objects.

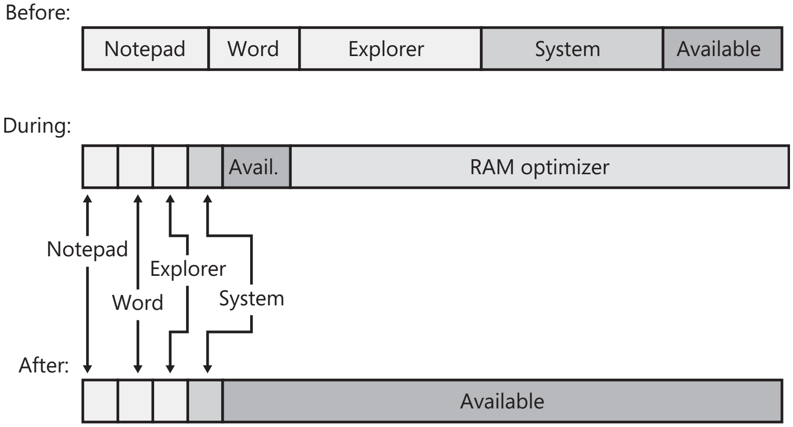
To see whether any processes in the system have requested memory resource notification, search the handle table for references to "LowMemoryCondition" or "HighMemoryCondition". You can do this by using Process Explorer's Find menu and choosing the Handle capability or by using the Oh.exe tool from the Resource Kits. (For a description of the handle table, see the section on the "Object Manager" in Chapter 3.)

RAM Optimizers: Fact or Fiction?

As you've surfed the Web, you've probably seen browser pop-ups such as "Defragment your memory and improve performance" and "Minimize application and system failures and free unused memory." The links lead you to utilities that promise to do all that and more. Do they really work?

Memory optimizers typically present a UI that shows a graph labeled "available memory" and a line representing a threshold below which the product will take action. Another line typically shows the amount of memory that the optimizer will try to free when it runs. You can usually configure one or both levels, as well as trigger manual memory optimization or schedule optimizations. Some tools also display the processes running on the system. When a scheduled optimization job runs, the utility's available memory counter often goes up, sometimes dramatically, which implies that the tool is actually freeing up memory for your applications to use. But what it's really doing is causing useful memory to be zeroed to artificially increase the available memory.

RAM optimizers work by allocating and then freeing large amounts of virtual memory. The figure below shows the effect a RAM optimizer has on a system:



The “before” bar depicts the working sets and available memory before optimization. The “during” bar shows that the RAM optimizer creates a high memory demand, which it does by incurring many page faults in a short time. In response, the memory manager increases the RAM optimizer’s working set. This working-set expansion occurs at the expense of available memory and—when available memory becomes low—at the expense of other process working sets. The “after” bar illustrates how, after the RAM optimizer frees its memory, the memory manager moves all the pages that were assigned to the RAM optimizer to the free page list (which ultimately get zeroed by the zero page thread and moved to the zero page list), thus contributing to the available memory value. Most optimizers hide the rapid decline in available memory that occurs during the first step, but if you run Task Manager during an optimization, you can often see the decline as it takes place.

While gaining more available memory might seem like a good thing, it isn’t. As RAM optimizers force the available memory counter up, they force other processes’ data and code out of memory. If you’re running Word, for example, the text of open documents and the program code that was part of Word’s working set before the optimization (and was therefore present in physical memory) must be reread from disk as you continue to edit your document. The performance degradation can be severe on servers, on which file data that was cached in the standby list and in the system working set (as well as the code and data used by any running server applications) would be discarded.

Some vendors make additional claims for their RAM-optimizer products. One claim you might see is that a product frees memory that’s needlessly consumed by unused processes, such as those that run in the Taskbar tray. That claim could be true only if those processes had sizable working sets at the time of optimization. However, because Windows automatically trims idle processes’ working sets, all such claims are untrue. The memory manager handles all necessary memory optimization.

Developers of RAM optimizers also claim that their products defragment memory. The act of allocating and then freeing a large amount of virtual memory might, as a conceivable side-effect, lead to large blocks of contiguous available memory. However, because virtual memory masks the layout of physical memory from processes, they can't directly benefit from having virtual memory backed by contiguous physical memory. As processes execute and undergo working-set trimming and growth, their virtual-memory-to-physical-memory mappings will become fragmented despite the availability of contiguous memory. Having contiguous available memory can result in increased performance in one case: To try to maximize the behavior of the CPU memory caches, the memory manager uses a mechanism called *page coloring* to decide which page from the free or zeroed list to assign to a process. However, any minor benefit that might result from making available physical memory contiguous is heavily outweighed by the negative effect of discarding valuable code and data from memory.

Finally, vendors often claim that RAM optimizers regain memory lost to leaks. This is perhaps the most patently false assertion of all. The memory manager knows at all times what physical and virtual memory belongs to a process. However, if a process allocates memory and then doesn't free it because of a bug (an occurrence known as a leak), the Memory Manager can't recognize that the allocated memory won't be accessed again at some point and must wait until the process exits to reclaim the memory. Even for leaking processes that do not exit, as a result of working-set trimming, the memory manager eventually will steal from the process's working set any physical pages that are assigned to leaked virtual memory. That process sends the leaked pages to the paging file and lets physical memory be used for other purposes. Thus, a memory leak has only a limited effect on available physical memory. The real effect is on virtual memory consumption, which Task Manager calls both PF Usage and Commit Charge. No utility can do anything about virtual memory consumption other than kill processes that are consuming memory.

In summary, common sense suggests that if RAM optimization were possible (and could be implemented by so many small-time upstarts), Microsoft developers would have long since integrated the technology into the kernel.

Conclusion

In this chapter, we've examined how the Windows memory manager implements virtual memory management. As with most modern operating systems, each process is given access to a private address space, protecting one process's memory from another's but allowing processes to share memory efficiently and securely. Advanced capabilities, such as the inclusion of mapped files and the ability to sparsely allocate memory, are also available. The Windows environment subsystem makes most of the memory manager's capabilities available to applications through the Windows API.

The memory manager's implementation relies on lazy-evaluation techniques whenever possible to avoid performing time-consuming and unnecessary operations unless they are required. It is also self-tuning, adapting to both large multiprocessor servers as well as uniprocessor desktop workstations.

One aspect of the memory manager that we didn't describe in this chapter is its tight integration with the cache manager, which we'll cover in Chapter 11. But before we get to that, let's take a closer look at the Windows security mechanisms.

Chapter 8

Security

Preventing unauthorized access to sensitive data is essential in any environment in which multiple users have access to the same physical or network resources. An operating system, as well as individual users, must be able to protect files, memory, and configuration settings from unwanted viewing and modification. Operating system security includes obvious mechanisms such as accounts, passwords, and file protection. It also includes less obvious mechanisms such as protecting the operating system from corruption, preventing less privileged users from performing actions (rebooting the computer, for example), and not allowing user programs to adversely affect the programs of other users or the operating system.

In this chapter, we explain how every aspect of the design and implementation of Microsoft Windows was influenced in some way by the stringent requirements of providing robust security.

Security Ratings

Having software, including operating systems, rated against well-defined standards helps the government, corporations, and home users protect proprietary and personal data stored in computer systems. The current security rating standard used by the United States and many other countries is the Common Criteria (CC). To understand the security capabilities designed into Windows, however, it's useful to know the history of the security ratings system that influenced the design of Windows, the Trusted Computer System Evaluation Criteria (TCSEC).

Trusted Computer System Evaluation Critiera

The National Computer Security Center (NCSC, at *www.radium.ncsc.mil/tpep/*) was established in 1981 as part of the U.S. Department of Defense's (DoD) National Security Agency (NSA). One goal of the NCSC was to create a range of security ratings, listed in Table 8-1, to be used to indicate the degree of protection commercial operating systems, network components, and trusted applications offer. These security ratings, which can be found at *www.radium.ncsc.mil/tpep/library/rainbow/5200.28-STD.html*, were defined in 1983 and are commonly referred to as "the Orange Book."

Table 8-1 TCSEC Rating Levels

Rating	Description
A1	Verified Design
B3	Security Domains
B2	Structured Protection
B1	Labeled Security Protection
C2	Controlled Access Protection
C1	Discretionary Access Protection (obsolete)
D	Minimal Protection

The TCSEC standard consists of "levels of trust" ratings, where higher levels build on lower levels by adding more rigorous protection and validation requirements. No operating system meets the A1, or "Verified Design," rating. Although a few operating systems have earned one of the B-level ratings, C2 is considered sufficient and the highest rating practical for a general-purpose operating system.

In July 1995, Windows NT 3.5 (Workstation and Server) with Service Pack 3 was the first version of Windows NT to earn the C2 rating. In March 1999, Windows NT 4 with Service Pack 3 achieved an E3 rating from the U.K. government's Information Technology Security (ITSEC) organization, a rating equivalent to a U.S. C2 rating. In November 1999, Windows NT 4 with Service Pack 6a earned a C2 rating in both stand-alone and networked configurations.

The following were the key requirements for a C2 security rating, and they are still considered the core requirements for any secure operating system:

- A *secure logon facility*, which requires that users can be uniquely identified and that they must be granted access to the computer only after they have been authenticated in some way.
- *Discretionary access control*, which allows the owner of a resource to determine who can access the resource and what they can do with it. The owner grants rights that permit various kinds of access to a user or to a group of users.
- *Security auditing*, which affords the ability to detect and record security-related events or any attempts to create, access, or delete system resources. Logon identifiers record the identities of all users, making it easy to trace anyone who performs an unauthorized action.
- *Object reuse protection*, which prevents users from seeing data that another user has deleted or from accessing memory that another user previously used and then released. For example, in some operating systems, it's possible to create a new file of a certain length and then examine the contents of the file to see data that happens to have occupied the location on the disk where the file is allocated. This data might be sensitive information that was stored in another user's file but that has been deleted. Object reuse protection prevents this potential security hole by initializing all objects, including files and memory, before they are allocated to a user.

Windows also meets two requirements of B-level security:

- *Trusted path functionality*, which prevents Trojan horse programs from being able to intercept users' names and passwords as they try to log in. The trusted path functionality in Windows comes in the form of its Ctrl+Alt+Delete logon-attention sequence, which cannot be intercepted by nonprivileged applications. This sequence of keystrokes, which is also known as the secure attention sequence (SAS), always pops up a logon dialog box, so would-be Trojan horses can easily be recognized. A Trojan horse presenting a fake logon dialog box will be bypassed when the SAS is entered.
- *Trusted facility management*, which requires support for separate account roles for administrative functions. For example, separate accounts are provided for administration (Administrators), user accounts charged with backing up the computer, and standard users.

Windows meets all of these requirements through its security subsystem and related components.

The Common Criteria

In January 1996, the United States, United Kingdom, Germany, France, Canada, and the Netherlands released the jointly developed Common Criteria for Information Technology Security Evaluation (CCITSE) security evaluation specification. CCITSE, which is usually referred to as the Common Criteria (CC), is the recognized multinational standard for product security evaluation. The CC home page is at *csrc.nist.gov/cc*.

The CC is more flexible than the TCSEC trust ratings and has a structure closer to the ITSEC than to the TCSEC. The CC includes the concept of a *Protection Profile* (PP) to collect security requirements into easily specified and compared sets, and the concept of a Security Target (ST) that contains a set of security requirements that can be made by reference to a PP.

Windows 2000 was evaluated as meeting the requirements of the Controlled Access PP, which is the equivalent of TCSEC C2, and additional Common Criteria functional and assurance requirements in October of 2002. Details of its conformance can be found at *niap.nist.gov/cc-scheme.html*. The additional requirements satisfied by Windows 2000 can be found in the certified Windows 2000 Common Criteria Security Target at *niap.nist.gov/cc-scheme/CCEVS-VID402-ST.pdf*. Notable requirements not included on the Controlled Access PP but included in the Windows 2000 Security Target include:

- *Discretionary Access Control Functions based on the use of cryptography.* This is implemented by Windows 2000 Encrypting File System and Windows 2000 Data Protection API.
- *Discretionary Access Control Policy for additional User Data Objects such as Desktops, Window Stations (as implemented by Windows 2000 Windowing System), and Active Directory Objects (as implemented by Windows 2000 Directory Service).*

- *Internal Replication for ensuring security-relevant data items are synchronized between physically separated parts of Windows 2000 system as distributed Operating System.* This is implemented Windows 2000 Directory Service using a multimaster loose consistency with convergence directory replication model.
- *Resource Utilization for physical disk spaces.* This is implemented by Windows 2000 NTFS.
- *Interactive Session Locking and Trusted Path for initial user logging on.* This is implemented by Windows 2000 Winlogon.
- *Internal Data Transfer Protection to protect data from disclosure and modification when transmitted between physically separate parts of Windows 2000 system as distributed Operating System.* This is implemented by Windows 2000 IPSEC service.
- *Systematic Flaw Remediation.* This is implemented by procedures undertaken by the Microsoft Security Response Center and Windows Sustained Engineering Team.

At the time of writing this book, Windows XP Embedded, Windows XP Professional, and Windows Server 2003 were still undergoing evaluation under the Common Criteria. The scope of evaluation is extended even further from that for Windows 2000. The Common Criteria authority currently recognizes Windows XP and Windows Server 2003 (Standard, Enterprise, and Datacenter Edition) as an evaluation of the following technology types as shown on *niap.nist.gov/cc-scheme/in_evaluation.html*:

- Distributed Operating System
- Sensitive Data Protection
- Network Management
- Directory Service
- Firewall
- VPN (Virtual Private Network)
- Desktop Management
- Public Key Infrastructure
- Public Key Certificate Issuing and Management
- Embedded System Operating System

Security System Components

These are the core components and databases that implement Windows security:

- **Security reference monitor (SRM)** A component in the Windows executive (\Windows\System32\Ntoskrnl.exe) that is responsible for defining the access token data structure to represent a security context, performing security access checks on objects, manipulating privileges (user rights), and generating any resulting security audit messages.

- **Local security authority subsystem (Lsass)** A user-mode process running the image \Windows\System32\Lsass.exe that is responsible for the local system security policy (such as which users are allowed to log on to the machine, password policies, privileges granted to users and groups, and the system security auditing settings), user authentication, and sending security audit messages to the Event Log. The local security authority service (Lsasrv—\Windows\System32\Lsasrv.dll), a library that Lsass loads, implements most of this functionality.
- **Lsass policy database** A database that contains the local system security policy settings. This database is stored in the registry under HKLM\SECURITY. It includes such information as what domains are entrusted to authenticate logon attempts, who has permission to access the system and how (interactive, network, and service logons), who is assigned which privileges, and what kind of security auditing is to be performed. The Lsass policy database also stores "secrets" that include logon information used for cached domain logons and Windows service user-account logons. (See Chapter 5 for more information on Windows services.)
- **Security Accounts Manager (SAM) service** A set of subroutines responsible for managing the database that contains the usernames and groups defined on the local machine. The SAM service, which is implemented as \Windows\System32\Samsrv.dll, runs in the Lsass process.
- **SAM database** A database that on systems not functioning as domain controllers contains the defined local users and groups, along with their passwords and other attributes. On domain controllers the SAM stores the system's administrator recovery account definition and password. This database is stored in the registry under HKLM\SAM.
- **Active Directory** A directory service that contains a database that stores information about objects in a domain. A *domain* is a collection of computers and their associated security groups that are managed as a single entity. Active Directory stores information about the objects in the domain, including users, groups, and computers. Password information and privileges for domain users and groups are stored in Active Directory, which is replicated across the computers that are designated as *domain controllers* of the domain. The Active Directory server, implemented as \Windows\System32\Ntdsa.dll, runs in the Lsass process. For more information on Active Directory, see Chapter 13.
- **Authentication packages** These include dynamic-link libraries (DLLs) that run both in the context of the Lsass process and client processes and that implement Windows authentication policy. An authentication DLL is responsible for checking whether a given username and password match, and if so, returning to the Lsass information detailing the user's security identity, which LSASS uses to generate a token.
- **Logon process (Winlogon)** A user-mode process running \Windows\System32\Winlogon.exe that is responsible for responding to the SAS and for managing interactive logon sessions. Winlogon creates a user's shell (user-interface) process when the user logs on, for example.

- **Graphical Identification and Authentication (GINA)** A user-mode DLL that runs in the Winlogon process and that Winlogon uses to obtain a user's name and password or smartcard PIN. The standard GINA is \Windows\System32\Msgina.dll.
- **Network logon service (Netlogon)** A Windows service (\Windows\System32\Netlogon.dll) that sets up the secure channel to a domain controller, over which security requests–such an interactive logon (if the domain controller is running Windows NT 4) or LAN Manager and NT LAN Manager (v1 and v2) authentication validation–are sent.
- **Kernel Security Device Driver (KSecDD)** A kernel-mode library of functions that implement the local procedure call (LPC) interfaces that other kernel-mode security components, including the Encrypting File System (EFS), use to communicate with Lsass in user mode. KSecDD is located in \Windows\System32\Drivers\Ksecdd.sys.

Figure 8-1 shows the relationships among some of these components and the databases they manage.

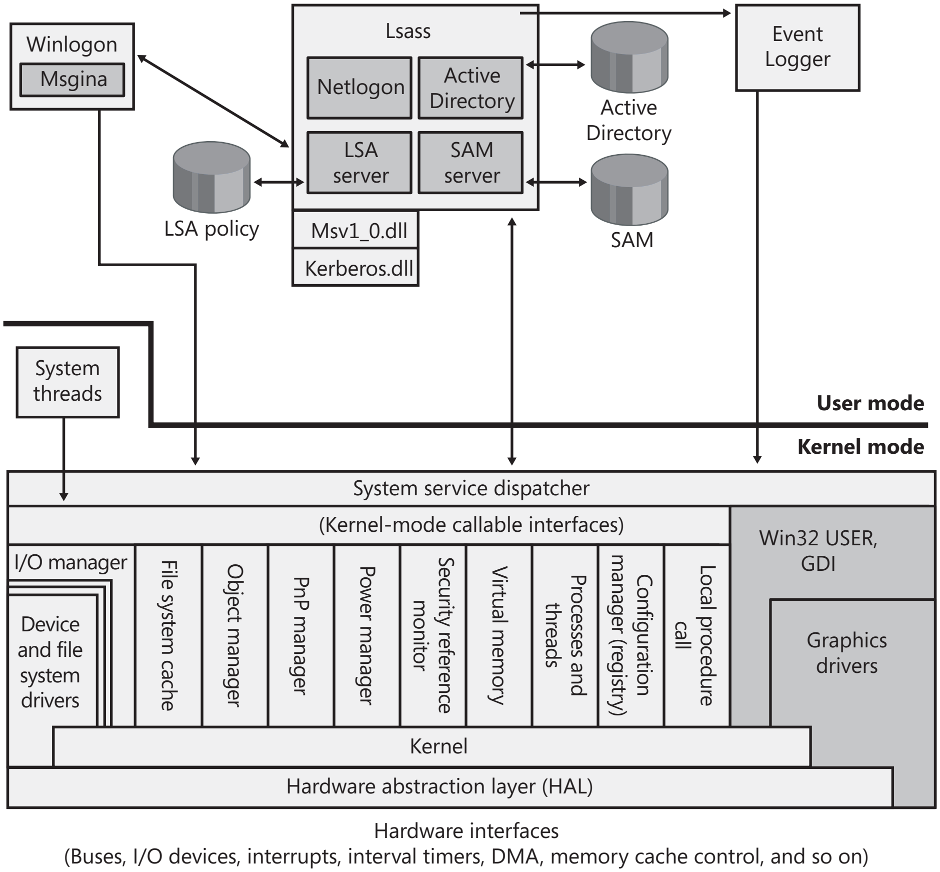
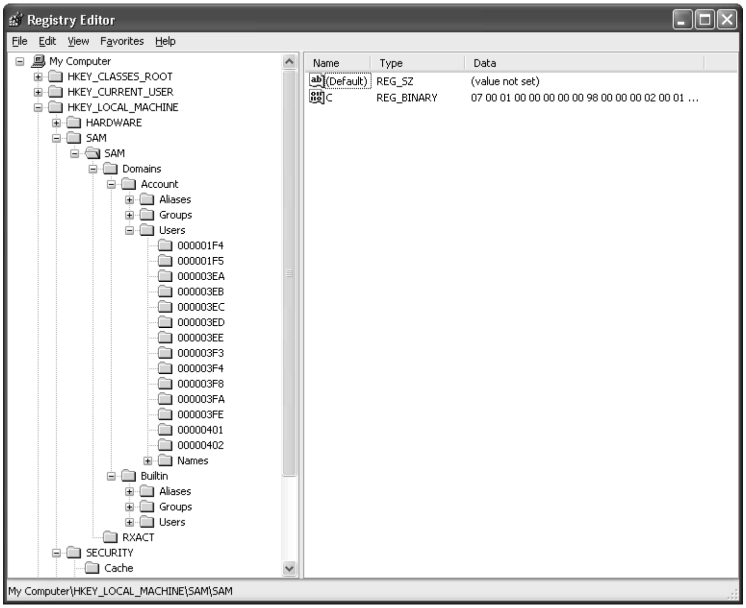


Figure 8-1 Windows security components

EXPERIMENT: Looking Inside HKLM\SAM and HKLM\Security

The security descriptors associated with the SAM and Security keys in the Registry prevent access by any account other than the Local System. One way to gain access to these keys for exploration is to reset their security, but that can weaken the system's security. Another way is to execute Regedit.exe in the Local System account, and PsExec from *www.sysinternals.com* supports an option that enables you to launch processes in the Local System account. Run Regedit using the *PsExec* command, shown below, to gain access to the SAM and Security databases without disturbing their security settings:

```
C:>psexec -s -i -d c:\windows\regedit.exe
```



```
My Computer
  HKEY_CLASSES_ROOT
  HKEY_CURRENT_USER
  HKEY_LOCAL_MACHINE
    HARDWARE
    SAM
      SAM
        Domains
          Account
            Aliases
            Groups
            Users
              000001F4
              000001F5
              000003EA
              000003EB
              000003EC
              000003ED
              000003EE
              000003F3
              000003F4
              000003F8
              000003FA
              000003FE
              00000401
              00000402
              Names
          Builtin
            Aliases
            Groups
            Users
        RXACT
    SECURITY
      Cache
```

Name	Type	Data
(Default)	REG_SZ	(value not set)
C	REG_BINARY	07 00 01 00 00 00 00 00 98 00 00 00 02 00 01 ...

My Computer\HKEY_LOCAL_MACHINE\SAM\SAM

The SRM, which runs in kernel mode, and Lsass, which runs in user mode, communicate using the LPC facility described in Chapter 3. During system initialization, the SRM creates a port, named SeRmCommandPort, to which Lsass connects. When the Lsass process starts, it creates an LPC port named SeLsaCommandPort. The SRM connects to this port, resulting in the creation of private communication ports. The SRM creates a shared memory section for messages longer than 256 bytes, passing a handle in the connect call. Once the SRM and Lsass connect to each other during system initialization, they no longer listen on their respective connect ports. Therefore, a later user process has no way to connect successfully to either of these ports for malicious purposes—the connect request will never complete.

Figure 8-2 shows the communication paths as they exist after system initialization.

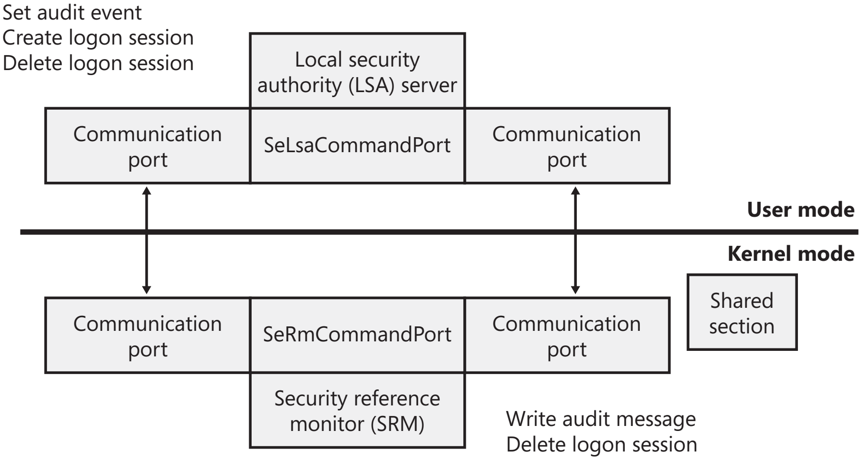


Figure 8-2 Communication between the SRM and Lsass

Protecting Objects

Object protection and access logging is the essence of discretionary access control and auditing. The objects that can be protected on Windows include files, devices, mailslots, pipes (named and anonymous), jobs, processes, threads, events, keyed events, event pairs, mutexes, semaphores, shared memory sections, I/O completion ports, LPC ports, waitable timers, access tokens, volumes, window stations, desktops, network shares, services, registry keys, printers, and Active Directory objects.

Because system resources that are exported to user mode (and hence require security validation) are implemented as objects in kernel mode, the Windows object manager plays a key role in enforcing object security. (For more information on the object manager, see Chapter 3.) To control who can manipulate an object, the security system must first be sure of each user's identity. This need to guarantee the user's identity is the reason that Windows requires authenticated logon before accessing any system resources. When a process requests a handle to an object, the object manager and the security system use the caller's security identification to determine whether the caller should be assigned a handle that grants the process access to the object it desires.

As discussed later in this chapter, a thread can assume a different security context than that of its process. This mechanism is called *impersonation*, and when a thread is impersonating, security validation mechanisms use the thread's security context instead of that of the thread's process. When a thread isn't impersonating, security validation falls back on using the security context of the thread's owning process. It's important to keep in mind that all the threads in a process share the same handle table, so when a thread opens an object—even if it's impersonating—all the threads of the process have access to the object.

Access Checks

The Windows security model requires that a thread specify up front, at the time that it opens an object, what types of actions it wants to perform on the object. The system performs access checks based on a thread's desired access, and if the access is granted, a handle is assigned to the thread's process with which the thread (or other threads in the process) can perform further operations on the object. As explained in Chapter 3, the object manager records the access permissions granted for a handle in the process's handle table.

One event that causes the object manager to perform security access validation is when a process opens an existing object using a name. When an object is opened by name, the object manager performs a lookup of the specified object in the object manager namespace. If the object isn't located in a secondary namespace, such as the configuration manager's registry namespace or a file system driver's file system namespace, the object manager calls the internal function *ObpCreateHandle* once it locates the object. As its name implies, *ObpCreateHandle* creates an entry in the process's handle table that becomes associated with the object. However, *ObpCreateHandle* calls the executive function *ExCreateHandle* instead to create the handle only if another object manager function, *ObpIncrementHandleCount*, indicates that the thread has permission to access the object. Another object manager function, *ObCheckObjectAccess*, actually carries out the security access check and returns the results to *ObpIncrementHandleCount*.

ObpIncrementHandleCount passes *ObCheckObjectAccess* the security credentials of the thread opening the object, the types of access to the object that the thread is requesting (read, write, delete, and so forth), and a pointer to the object. *ObCheckObjectAccess* first locks the object's security and the security context of the thread. The object security lock prevents another thread in the system from changing the object's security while the access check is in progress. The lock on the thread's security context prevents another thread of that process or a different process from altering the security identity of the thread while security validation is in progress. *ObCheckObjectAccess* then calls the object's security method to obtain the security settings of the object. (See Chapter 3 for a description of object methods.) The call to the security method might invoke a function in a different executive component. However, many executive objects rely on the system's default security management support.

When an executive component defining an object doesn't want to override the SRM's default security policy, it marks the object type as having default security. Whenever the SRM calls an object's security method, it first checks to see whether the object has default security. An object with default security stores its security information in its header, and its security method is *SeDefaultObjectMethod*. An object that doesn't rely on default security must manage its own security information and supply a specific security method. Objects that rely on default security include mutexes, events, and semaphores. A file object is an example of an object that overrides default security. The I/O manager, which defines the file object type, has the file system driver on which a file resides manage (or choose not to implement) the security for its files. Thus, when the system queries the security on a file object that represents a file on an NTFS volume, the I/O manager file object security method retrieves the file's security using the NTFS file system driver. Note, however, that *ObCheckObjectAccess* isn't executed when files are opened, because they reside in secondary namespaces; the system invokes a file object's security method only when a thread explicitly queries or sets the security on a file (with the Windows *SetFileSecurity* or *GetFileSecurity* functions, for example).

After obtaining an object's security information, *ObCheckObjectAccess* invokes the SRM function *SeAccessCheck*. *SeAccessCheck* is one of the functions at the heart of the Windows security model. Among the input parameters *SeAccessCheck* accepts are the object's security information, the security identity of the thread as captured by *ObCheckObjectAccess*, and the access that the thread is requesting. *SeAccessCheck* returns True or False, depending on whether the thread is granted the access it requested to the object.

Another event that causes the object manager to execute access validation is when a process references an object using an existing handle. Such references often occur indirectly, as when a process calls on a Windows API to manipulate an object and passes an object handle. For example, a thread opening a file can request access to the object that permits it to read from the file. If the thread has permission to access the object in this way, as dictated by its security context and the security settings of the file, the object manager creates a handle—representing the file—in the handle table of the thread's process. The accesses the process is granted through the handle are stored with the handle by the object manager.

Subsequently, the thread can attempt to write to the file using the *WriteFile* Windows function, passing the file's handle as a parameter. The system service *NtWriteFile*, which *WriteFile* calls via Ntdll.dll, uses the object manager function *ObReferenceObjectByHandle* to obtain a pointer to the file object from the handle. *ObReferenceObjectByHandle* accepts the access that the caller wants from the object as a parameter. After finding the handle entry in the process's handle table, *ObReferenceObjectByHandle* compares the access being requested with the access granted at the time the file was opened. In this case, *ObReferenceObjectByHandle* will indicate that the write operation should fail because the caller didn't obtain write access when the file was opened.

The Windows security functions also enable Windows applications to define their own private objects and to call on the services of the SRM to enforce the Windows security model on those objects. Many kernel-mode functions that the object manager and other executive components use to protect their own objects are exported as Windows user-mode APIs. The user-mode equivalent of *SeAccessCheck* is *AccessCheck*, for example. Windows applications can therefore leverage the flexibility of the security model and transparently integrate with the authentication and administrative interfaces that are present in Windows.

The essence of the SRM's security model is an equation that takes three inputs: the security identity of a thread, the access that the thread wants to an object, and the security settings of the object. The output is either "yes" or "no" and indicates whether or not the security model grants the thread the access it desires. The following sections describe the inputs in more detail and then document the model's access validation algorithm.

Security Identifiers (SIDs)

Instead of using names (which might or might not be unique) to identify entities that perform actions in a system, Windows uses *security identifiers* (SIDs). Users have SIDs, and so do local and domain groups, local computers, domains, and domain members. A SID is a variable-length numeric value that consists of a SID structure revision number, a 48-bit identifier authority value, and a variable number of 32-bit subauthority or *relative identifier* (RID) values. The authority value identifies the agent that issued the SID, and this agent is typically a Windows local system or a domain. Subauthority values identify trustees relative to the issuing authority, and RIDs are simply a way for Windows to create unique SIDs based on a common-base SID. Because SIDs are long and Windows takes care to generate truly random values within each SID, it is virtually impossible for Windows to issue the same SID twice on machines or domains anywhere in the world.

When displayed textually, each SID carries an *S* prefix, and its various components are separated with hyphens:

S-1-5-21-1463437245-1224812800-863842198-1128

In this SID, the revision number is 1, the identifier authority value is 5 (the Windows security authority), and four subauthority values plus one RID (1128) make up the remainder of the SID. This SID is a domain SID, but a local computer on the domain would have a SID with the same revision number, identifier authority value, and number of subauthority values.

When you install Windows, the Windows Setup program issues the computer a SID. Windows assigns SIDs to local accounts on the computer. Each local-account SID is based on the source computer's SID and has a RID at the end. RIDs for user accounts and groups start at 1000 and increase in increments of 1 for each new user or group. Similarly, Dcpromo.exe, the utility used to create a new Windows domain, issues a SID to domains it creates. Windows issues to new domain accounts SIDS that are based on the domain SID and have an appended RID (again starting at 1000 and increasing in increments of 1 for each new user or group). A RID of 1028 indicates that the SID is the 29th SID the domain issued.

Windows issues SIDS that consist of a computer or domain SID with a predefined RID to many predefined accounts and groups. For example, the RID for the administrator account is 500, and the RID for the guest account is 501. A computer's local administrator account, for example, has the computer SID as its base with the RID of 500 appended to it:

S-1-5-21-13124455-12541255-61235125-500

Windows also defines a number of built-in local and domain SIDs to represent groups. For example, a SID that identifies any and every account is the Everyone, or World, SID: S-1−1−0. Another example of a group that a SID can represent is the network group, which is the group that represents users who have logged on to a machine from the network. The network-group SID is S-1-5-2. Table 8-2, reproduced here from the Platform SDK documentation, shows some basic well-known SIDs, their numeric values, and their use.

Finally, Winlogon creates a unique logon SID for each interactive logon session. A typical use of a logon SID is in an access-control entry (ACE) that allows access for the duration of a client's logon session. For example, a Windows service can use the *LogonUser* function to start a new logon session. The *LogonUser* function returns an access token from which the service can extract the logon SID. The service can then use the SID in an ACE that allows the client's logon session to access the interactive window station and desktop. The SID for a logon session is S-1-5-5-0, and the RID is randomly generated.

Table 8-2 Well-Known SIDs

SID	Group	Use
S-1–1–0	Everyone	A group that includes all users.
S-1-2-0	Local	Users who log on to terminals locally (physically) connected to the system.
S-1–3–0	Creator Owner ID	A security identifier to be replaced by the security identifier of the user who created a new object. This SID is used in inheritable ACEs.
S-1–3–1	Creator Group ID	Identifies a security identifier to be replaced by the primary-group SID of the user who created a new object. Use this SID in inheritable ACEs.

EXPERIMENT: Using PsGetSid to View Account SIDs

You can easily see the SID representation for any account you're using by running the PsGetSid utility (from *www.sysinternals.com*). It has the following interface:

```
C:\>psgetsid -?

PsGetSid displays the machine SID for the local or remote Windows system.

Usage: psgetsid [\\RemoteComputer [-u Username [-p Password]]] [account | SID]
     -u       Specifies optional user name for login to
              remote computer.
     -p       Specifies optional password for user name. If you omit this
              you will be prompted to enter a hidden password.
     account  PsGetSid will report the SID for the specified user account
              rather than the computer.
     SID      PsGetSid will report the account for the specified SID.
```

PsGetSid's options allow you to translate machine and user account names to their corresponding SIDs and vice versa.

If you run PsGetSid with no options, it prints the SID assigned to the local computer. By using the fact that the Administrator's account always has a RID of 500, you can determine the name assigned to the account (in cases where a system administrator has renamed the account for security reasons) simply by passing the machine SID appended with "-500" as PsGetSid's command-line argument.

To obtain the SID of a domain account, enter the user name with the domain as a prefix:

```
c:\>psgetsid redmond\daryl
```

You can determine the SID of a domain by specifying the domain's name as the argument to PsGetSid:

```
c:\>psgetsid Redmond
```

Finally, by examining the RID of your own account, you know at least a number of security accounts equal to the number resulting from subtracting 999 from your RID have been created in your domain or on your local machine (depending on whether you are using a domain or local machine account). You can determine what accounts have been assigned RIDs by passing a SID with the RID you want to query to PsGetSid. If PsGetSid reports that no mapping between the SID and an account name was possible and the RID is lower than that of your account, you know that the account assigned the RID has been deleted.

For example, to find out the name of the account assigned the twenty-eighth RID, pass the domain SID appended with "-1027" to PsGetSid:

```
c:\>psgetsid S-1-5-21-1787744166-3910675280-2727264193-1027
Account for S-1-5-21-1787744166-3910675280-2727264193-1027:
User: redmond\daryl
```

Tokens

The SRM uses an object called a *token* (or *access token*) to identify the security context of a process or thread. A security context consists of information that describes the privileges, accounts, and groups associated with the process or thread. During the logon process (described at the end of this chapter), Winlogon creates an initial token to represent the user logging on and attaches the token to the initial process(es) it starts (by default, Userinit.exe). Because child processes by default inherit a copy of the token of its creator, all processes in the user's session run under the same token. You can also generate a token by using the Windows

LogonUser function. You can then use this token to create a process that runs within the security context of the user logged on by the *LogonUser* function by passing the token to the Windows *CreateProcessAsUser* function. The *CreateProcessWithLogon* function also creates a token by creating a new logon session with an initial process. This is how the *Runas* command launches processes under alternate tokens.

Tokens vary in size because different user accounts have different sets of privileges and associated group accounts. However, all tokens contain the same information, shown in Figure 8-3.

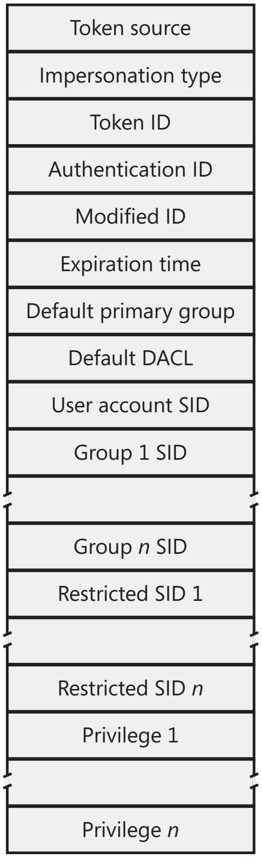


Figure 8-3 Access tokens

The security mechanisms in Windows use two token components to determine what objects can be accessed and what secure operations can be performed. One component comprises the token's user account SID and group SID fields. The SRM uses SIDs to determine whether a process or thread can obtain requested access to a securable object, such as an NTFS file.

The group SIDs in a token signify which groups a user's account is a member of. A server application can disable specific groups to restrict a token's credentials when the server application is performing actions a client requests. Disabling a group produces nearly the same effect as if the group wasn't present in the token. (Disabled SIDs are used as part of security access checks, described later in the chapter.)

The second component in a token that determines what the token's thread or process can do is the privilege array. A token's privilege array is a list of rights associated with the token. An example privilege is the right for the process or thread associated with the token to shut down the computer. Privileges are described in more detail later in this chapter. A token's default primary group field and default discretionary access-control list (DACL) field are security attributes that Windows applies to objects that a process or thread creates when it uses the token. By including security information in tokens, Windows makes it convenient for a process or thread to create objects with standard security attributes, because the process or thread doesn't need to request discrete security information for every object it creates.

Each token's type distinguishes a primary token (a token that identifies the security context of a process) from an impersonation token (a type of token that threads use to temporarily adopt a different security context, usually of another user). Impersonation tokens carry an impersonation level that signifies what type of impersonation is active in the token. (Impersonation is described later in this chapter.)

The remainder of the fields in a token serve informational purposes. The token source field contains a short textual description of the entity that created the token. Programs that want to know where a token originated use the token source to distinguish among sources such as the Windows Session Manager, a network file server, or the remote procedure call (RPC) server. The token identifier is a locally unique identifier (LUID) that the SRM assigns to the token when it creates the token. The Windows executive maintains the executive LUID, a counter it uses to assign a unique numeric identifier to each token.

The token authentication ID is another kind of LUID. A token's creator assigns the token's authentication ID when calling the *LsaLogonUser* function. If the creator doesn't specify a LUID, Lsass obtains the LUID from the executive LUID. Lsass copies the authentication ID for all tokens descended from an initial logon token. A program can obtain a token's authentication ID to see whether the token belongs to the same logon session as other tokens the program has examined.

The executive LUID refreshes the modified ID every time a token's characteristics are modified. An application can test the modified ID to discover changes in a security context since the context's last use.

Tokens contain an expiration time field that has been present but unused in Windows NT technology since Windows NT 3.1. A future version of Windows might allow for tokens that are valid for a period of time before expiring. Consider a user for which the systems administrator sets an account expiration time. Currently, if the user logs on and remains logged on past the account expiration, the system will let the user continue to access resources. The only way to prevent the user from accessing resources is to forcibly log the user off the machine. If Windows supported token expiration, the system could prevent the user from opening resources past the token expiration time.

EXPERIMENT: Viewing Access Tokens

The kernel debugger *dt _TOKEN* command displays the format of an internal token object. Although this structure differs from the user-mode token structure returned by Windows API security functions, the fields are similar. For further information on tokens, see the description in the Platform SDK documentation.

The following output is from the kernel debugger's *dtTOKEN* command:

```
kd> dt _TOKEN
   +0x000 TokenSource      : _TOKEN_SOURCE
   +0x010 TokenId          : _LUID
   +0x018 AuthenticationId : _LUID
   +0x020 ParentTokenId    : _LUID
   +0x028 ExpirationTime   : _LARGE_INTEGER
   +0x030 TokenLock        : Ptr32 _ERESOURCE
   +0x034 ModifiedId       : _LUID
   +0x03c SessionId        : Uint4B
   +0x040 UserAndGroupCount : Uint4B
   +0x044 RestrictedSidCount : Uint4B
   +0x048 PrivilegeCount   : Uint4B
   +0x04c VariableLength   : Uint4B
   +0x050 DynamicCharged   : Uint4B
   +0x054 DynamicAvailable : Uint4B
   +0x058 DefaultOwnerIndex : Uint4B
   +0x05c UserAndGroups    : Ptr32 _SID_AND_ATTRIBUTES
   +0x060 RestrictedSids   : Ptr32 _SID_AND_ATTRIBUTES
   +0x064 PrimaryGroup     : Ptr32 Void
   +0x068 Privileges       : Ptr32 _LUID_AND_ATTRIBUTES
   +0x06c DynamicPart      : Ptr32 Uint4B
   +0x070 DefaultDacl      : Ptr32 _ACL
   +0x074 TokenType        : _TOKEN_TYPE
   +0x078 ImpersonationLevel : _SECURITY_IMPERSONATION_LEVEL
   +0x07c TokenFlags       : UChar
   +0x07d TokenInUse       : UChar
   +0x080 ProxyData        : Ptr32 _SECURITY_TOKEN_PROXY_DATA
   +0x084 AuditData        : Ptr32 _SECURITY_TOKEN_AUDIT_DATA
   +0x088 VariablePart     : Uint4B
```

You can examine the token for a process with the *!token* command. You'll find the address of the token in the output of the *!process* command, as shown here:

```
kd> !process 380 1
!process 380 1
Searching for Process with Cid == 380
PROCESS ff8027a0 SessionId: 0  Cid: 0380  Peb: 7ffdf000  ParentCid: 0124
  DirBase: 06433000  ObjectTable: ff7e0b68  TableSize:  23.
  Image: cmd.exe
  VadRoot 84c30568 Clone 0 Private 77. Modified 0. Locked 0.
  DeviceMap 818a3368
  Token                             e22bc730
  ElapsedTime                       14:22:56.0536
  UserTime                          0:00:00.0040
  KernelTime                        0:00:00.0100
  QuotaPoolUsage[PagedPool]         13628
  QuotaPoolUsage[NonPagedPool]      1616
  Working Set Sizes (now,min,max)  (261, 50, 345)(1044KB, 200KB, 1380KB)
  PeakWorkingSetSize                262
  VirtualSize                       11 Mb
  PeakVirtualSize                   11 Mb
  PageFaultCount                    313
  MemoryPriority                    FOREGROUND
  BasePriority                      8
  CommitCharge                      86

kd> !token e22bc730
_TOKEN e22bc730
TS Session ID: 0
User: S-1-5-21-1787744166-3910675280-2727264193-500
Groups:
 00 S-1-5-21-1787744166-3910675280-2727264193-513
    Attributes - Mandatory Default Enabled
 01 S-1-1-0
    Attributes - Mandatory Default Enabled
 02 S-1-5-32-544
    Attributes - Mandatory Default Enabled Owner
 03 S-1-5-32-545
    Attributes - Mandatory Default Enabled
 04 S-1-5-5-0-92587
    Attributes - Mandatory Default Enabled LogonId
 05 S-1-2-0
    Attributes - Mandatory Default Enabled
 06 S-1-5-14
    Attributes - Mandatory Default Enabled
 07 S-1-5-4
    Attributes - Mandatory Default Enabled
 08 S-1-5-11
    Attributes - Mandatory Default Enabled
Primary Group: S-1-5-21-1787744166-3910675280-2727264193-513
Privs:
 00 0x000000017 SeChangeNotifyPrivilege           Attributes - Enabled Default
 01 0x000000008 SeSecurityPrivilege               Attributes -
 02 0x000000011 SeBackupPrivilege                 Attributes -
 03 0x000000012 SeRestorePrivilege                Attributes -
 04 0x00000000c SeSystemtimePrivilege             Attributes -
 05 0x000000013 SeShutdownPrivilege               Attributes -
 06 0x000000018 SeRemoteShutdownPrivilege         Attributes -
 07 0x000000009 SeTakeOwnershipPrivilege          Attributes -
 08 0x000000014 SeDebugPrivilege                  Attributes -
 09 0x000000016 SeSystemEnvironmentPrivilege      Attributes -
 10 0x00000000b SeSystemProfilePrivilege          Attributes -
 11 0x00000000d SeProfileSingleProcessPrivilege   Attributes -
 12 0x00000000e SeIncreaseBasePriorityPrivilege   Attributes -
 13 0x00000000a SeLoadDriverPrivilege             Attributes - Enabled
 14 0x00000000f SeCreatePagefilePrivilege         Attributes -
 15 0x000000005 SeIncreaseQuotaPrivilege          Attributes -
 16 0x000000019 SeUndockPrivilege                 Attributes - Enabled
 17 0x00000001c SeManageVolumePrivilege           Attributes -
```

```
Authentication ID:         (0,169e4)
Impersonation Level:       Anonymous
TokenType:                 Primary
Source: User32             TokenFlags: 0x9 ( Token in use )
Token ID: 3a86d0d          ParentToken ID: 0
Modified ID:               (0, 3a86d0f)
RestrictedSidCount: 0      RestrictedSids: 00000000
```

You can indirectly view token contents with Process Explorer's (available from *www.sysinternals.com*) security tab of its process properties dialog box. The dialog box shows the groups and privileges included in the token of the process you examine.

Impersonation

Impersonation is a powerful feature Windows uses frequently in its security model. Windows also uses impersonation in its client/server programming model. For example, a server application can export resources such as files, printers, or databases. Clients wanting to access a resource send a request to the server. When the server receives the request, it must ensure that the client has permission to perform the desired operations on the resource. For example, if a user on a remote machine tries to delete a file on an NTFS share, the server exporting the share must determine whether the user is allowed to delete the file. The obvious process to determine whether a user has permission is for the server to query the user's account and group SIDs and scan the security attributes on the file. This process is tedious to program, prone to errors, and wouldn't permit new security features to be supported transparently. Thus, Windows provides impersonation services to simplify the server's job.

Impersonation lets a server notify the SRM that the server is temporarily adopting the security profile of a client making a resource request. The server can then access resources on behalf of the client, and the SRM carries out the access validations. Usually, a server has access to more resources than a client does and loses some of its security credentials during impersonation. However, the reverse can be true: the server can gain security credentials during impersonation.

A server impersonates a client only within the thread that makes the impersonation request. Thread-control data structures contain an optional entry for an impersonation token. However, a thread's primary token, which represents the thread's real security credentials, is always accessible in the process's control structure.

Windows makes impersonation available through several mechanisms. If a server communicates with a client through a named pipe, the server can use the *ImpersonateNamedPipeClient* Windows API function to tell the SRM that it wants to impersonate the user on the other end of the pipe. If the server is communicating with the client through Dynamic Data Exchange (DDE) or an RPC, it can make similar impersonation requests using *DdeImpersonateClient* and *RpcImpersonateClient*. A thread can create an impersonation token that's simply a copy of its process token with the *ImpersonateSelf* function. The thread can then alter its impersonation token, to disable SIDs or privileges, for example. Finally, a Security Support Provider Interface (SSPI) package can impersonate its clients with *ImpersonateSecurityContext*. SSPIs implement a network security model such as LAN Manager version 2 or Kerberos.

After the server thread finishes its task, it reverts to its primary security profile. These forms of impersonation are convenient for carrying out specific actions at the request of a client and for ensuring that object accesses are audited correctly. (For example, the audit that is generated gives the identity of the impersonated client rather than that of the server process.) The disadvantage to these forms of impersonation is that they can't execute an entire program in the context of a client. In addition, an impersonation token can't access files or printers on network shares unless it is a delegation-level impersonation (described shortly) and has sufficient credentials to authenticate to the remote machine, or the file or printer share supports null sessions. (A null session is one that results from an anonymous logon.)

If an entire application must execute in a client's security context or must access network resources, the client must be logged on to the system. The *LogonUser* Windows API function enables this action. *LogonUser* takes an account name, a password, a domain or computer name, a logon type (such as interactive, batch, or service), and a logon provider as input, and it returns a primary token. A server thread can adopt the token as an impersonation token, or the server can start a program that has the client's credentials as its primary token. From a security standpoint, a process created using the token returned from an interactive logon via *LogonUser*, such as with the *CreateProcessAsUser* API, looks like a program a user starts by logging on to the machine interactively. The disadvantage to this approach is that a server must obtain the user's account name and password. If the server transmits this information across the network, the server must encrypt it securely so that a malicious user snooping network traffic can't capture it.

To prevent the misuse of impersonation, Windows doesn't let servers perform impersonation without a client's consent. A client process can limit the level of impersonation that a server process can perform by specifying a *security quality of service* (SQOS) when connecting to the server. A process can specify SECURITY_ANONYMOUS, SECURITY_IDENTIFICATION, SECURITY_IMPERSONATION, and SECURITY_DELEGATION as flags for the Windows *CreateFile* function. Each level lets a server perform different types of operations with respect to the client's security context:

- *SecurityAnonymous* is the most restrictive level of impersonation—the server can't impersonate or identify the client.
- *SecurityIdentification* level lets the server obtain the identity (the SIDs) of the client and the client's privileges, but the server can't impersonate the client.
- *SecurityImpersonation* level lets the server identify and impersonate the client on the local system.
- *SecurityDelegation* is the most permissive level of impersonation. It lets the server impersonate the client on local and remote systems. Windows NT 4 and earlier don't fully support SecurityDelegation-level impersonation.

If the client doesn't set an impersonation level, Windows chooses the SecurityImpersonation level by default. The *CreateFile* function also accepts SECURITY_EFFECTIVE_ONLY and SECURITY_CONTEXT_TRACKING as modifiers for the impersonation setting:

- SECURITY_EFFECTIVE_ONLY prevents a server from enabling or disabling a client's privileges or groups while the server is impersonating.
- SECURITY_CONTEXT_TRACKING specifies that any changes a client makes to its security context is reflected in a server that is impersonating it. If this option isn't specified, the server adopts the context of the client at the time of the impersonation and doesn't receive any changes. This option is honored only when the client and server process are on the same system.

Restricted Tokens

A *restricted token* is created from a primary or impersonation token using the *CreateRestrictedToken* function. The restricted token is a copy of the token it's derived from, with the following possible modifications:

- Privileges can be removed from the token's privilege array.
- SIDs in the token can be marked as *deny-only*.
- SIDs in the token can be marked as *restricted*.

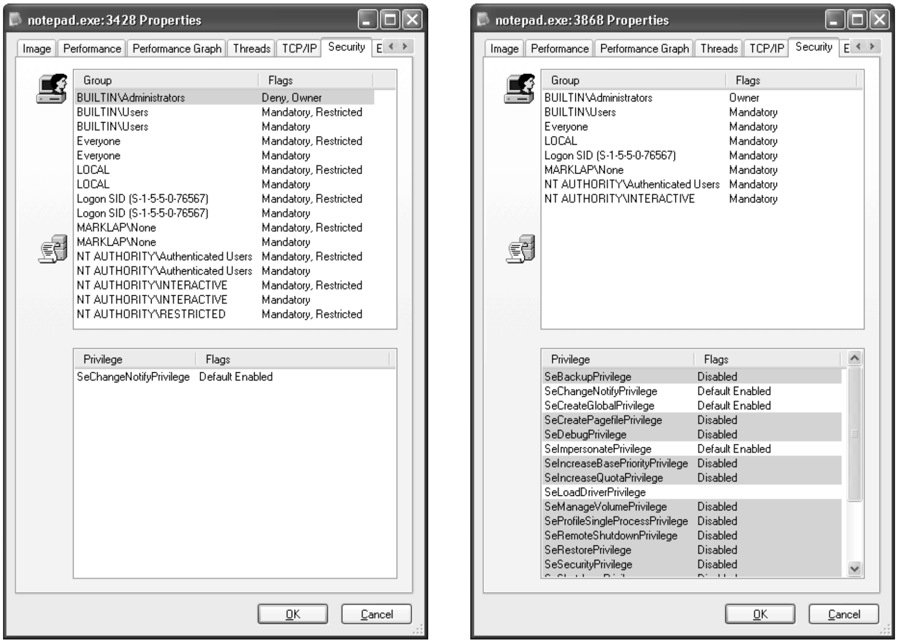
The behavior of deny-only and restricted SIDs is covered shortly. Restricted tokens are useful when an application wants to impersonate a client at a reduced security level, primarily for safety reasons when running untrusted code. For example, the restricted token can have the shutdown-system privilege removed from it to prevent code executed in the restricted token's security context from rebooting the system.

EXPERIMENT: Looking at Restricted Tokens

You can make Explorer create a process with a restricted token by following these steps on Windows XP or Windows Server 2003:

1. Create a shortcut to \Windows\Notepad.exe on your desktop.
2. Edit the properties of the shortcut, and select the Run With Different Credentials check box. Note that the descriptive text for the check box tells you that you can run the program as yourself while protecting your computer from unauthorized program activity.
3. Dismiss the shortcut properties, and launch the shortcut.
4. Accept the default options of running in the current account and protecting your computer from unauthorized program activity that the resulting Run As dialog displays.

5. Run Process Explorer, and view the Security tab of the properties of the Notepad process you launched. Note that the token contains restricted SIDs and a deny-only SID, and that it has only one privilege. The properties on the left in the following screen shot are from a Notepad running with a nonrestricted token, and the properties on the right are from one launched using these steps:



Group	Flags
BUILTIN\Administrators	Deny, Owner
BUILTIN\Users	Mandatory, Restricted
BUILTIN\Users	Mandatory
Everyone	Mandatory, Restricted
Everyone	Mandatory
LOCAL	Mandatory, Restricted
LOCAL	Mandatory
Logon SID (S-1-5-5-0-76567)	Mandatory, Restricted
Logon SID (S-1-5-5-0-76567)	Mandatory
MARKLAP\None	Mandatory, Restricted
MARKLAP\None	Mandatory
NT AUTHORITY\Authenticated Users	Mandatory, Restricted
NT AUTHORITY\Authenticated Users	Mandatory
NT AUTHORITY\INTERACTIVE	Mandatory, Restricted
NT AUTHORITY\INTERACTIVE	Mandatory
NT AUTHORITY\RESTRICTED	Mandatory, Restricted

Privilege	Flags
SeChangeNotifyPrivilege	Default Enabled

Group	Flags
BUILTIN\Administrators	Owner
BUILTIN\Users	Mandatory
Everyone	Mandatory
LOCAL	Mandatory
Logon SID (S-1-5-5-0-76567)	Mandatory
MARKLAP\None	Mandatory
NT AUTHORITY\Authenticated Users	Mandatory
NT AUTHORITY\INTERACTIVE	Mandatory

Privilege	Flags
SeBackupPrivilege	Disabled
SeChangeNotifyPrivilege	Default Enabled
SeCreateGlobalPrivilege	Default Enabled
SeCreatePagefilePrivilege	Disabled
SeDebugPrivilege	Disabled
SeImpersonatePrivilege	Default Enabled
SeIncreaseBasePriorityPrivilege	Disabled
SeIncreaseQuotaPrivilege	Disabled
SeLoadDriverPrivilege	
SeManageVolumePrivilege	Disabled
SeProfileSingleProcessPrivilege	Disabled
SeRemoteShutdownPrivilege	Disabled
SeRestorePrivilege	Disabled
SeSecurityPrivilege	Disabled

There are several side-effects of the restricted token you see:

- All privileges except SeChangeNotifyPrivilege are removed.
- Any Administrators or Power Users SIDs are marked Deny-Only. This SID removes access to any resources for which administrator access is denied with a matching access-denied ACE, but that would otherwise be overridden by an ACE granting the administrator's group access earlier in the security descriptor.
- The RESTRICTED SID is added to the Restricted SIDs list, as are all other token SIDs except for the user SID and any Administrators or Power Users SIDs.
- The SID of the account from which you ran the process is not listed as a restricted SID. This means that the process won't be able to access any objects that permit access to your user account but not to any of the groups that you belong to. For example, your profile directory under \Documents and Settings has a default security descriptor granting your account, the administrator's group, and the System account access. When you try to open that directory from the Notepad you launched, you'll be denied access because the internal second access check, performed with restricted SIDs, will fail because the user SID isn't present in the list of restricted SIDs.

Security Descriptors and Access Control

Tokens, which identify a user's credentials, are only part of the object security equation. Another part of the equation is the security information associated with an object, which specifies who can perform what actions on the object. The data structure for this information is called a *security descriptor*. A security descriptor consists of the following attributes:

- **Revision number** The version of the SRM security model used to create the descriptor.
- **Flags** Optional modifiers that define the behavior or characteristics of the descriptor. An example is the SE_DACL_PROTECTED flag, which prevents the descriptor from inheriting a security setting from another object.
- **Owner SID** The owner's security ID.
- **Group SID** The security ID of the primary group for the object (used only by POSIX).
- **Discretionary access-control list (DACL)** Specifies who has what access to the object.
- **System access-control list (SACL)** Specifies which operations by which users should be logged in the security audit log.

An *access-control list* (ACL) is made up of a header and zero or more *access-control entry* (ACE) structures. There are two types of ACLs: DACLs and SACLs. In a DACL, each ACE contains a SID and an access mask (and a set of flags, explained shortly). Four types of ACEs can appear in a DACL: access allowed, access denied, allowed-object, and denied-object. As you would expect, the access-allowed ACE grants access to a user, and the access-denied ACE denies the access rights specified in the access mask.

The difference between allowed-object and access allowed, and between denied-object and access denied, is that the object types are used only within Active Directory. ACEs of these types have a GUID (globally unique identifier) field that indicates that the ACE applies only to particular objects or subobjects (those that have GUID identifiers). In addition, another optional GUID indicates what type of child object will inherit the ACE when a child is created within an Active Directory container that has the ACE applied to it. (A GUID is a 128-bit identifier guaranteed to be universally unique.)

The accumulation of access rights granted by individual ACEs forms the set of access rights granted by an ACL. If no DACL is present (a *null* DACL) in a security descriptor, everyone has full access to the object. If the DACL is empty (that is, it has 0 ACEs), no user has access to the object.

The ACEs used in DACLs also have a set of flags that control and specify characteristics of the ACE related to inheritance. Some object namespaces have *container objects* and *leaf objects* (or just *objects*). A container can hold other container objects and leaf objects, which are its *child objects*. Examples of containers are directories in the file system namespace and keys in the registry namespace. Certain flags in an ACE control how the ACE propagates to child objects of the container associated with the ACE. Table 8-3, reproduced in part from the Platform SDK, lists the inheritance rules for ACE flags.

Table 8-3 **Inheritance Rules for ACE Flags**

Flag	Inheritance rule
CONTAINER_INHERIT_ACE	Child objects that are containers, such as directories, inherit the ACE as an effective ACE. The inherited ACE is inheritable unless the NO_PROPAGATE_INHERIT_ACE bit flag is also set.
INHERIT_ONLY_ACE	This flag indicates an inherit-only ACE that doesn't control access to the object it's attached to.
INHERITED_ACE	This flag indicates that the ACE was inherited. The system sets this bit when it propagates an inheritable ACE to a child object.
NO_PROPAGATE_INHERIT_ACE	If the ACE is inherited by a child object, the system clears the OBJECT_INHERIT_ACE and CONTAINER_INHERIT_ACE flags in the inherited ACE. This action prevents the ACE from being inherited by subsequent generations of objects.
OBJECT_INHERIT_ACE	Noncontainer child objects inherit the ACE as an effective ACE. For child objects that are containers, the ACE is inherited as an inherit-only ACE unless the NO_PROPAGATE_INHERIT_ACE bit flag is also set.

A SACL contains two types of ACEs, *system audit* ACEs and *system audit-object* ACEs. These ACEs specify which operations performed on the object by specific users or groups should be audited. Audit information is stored in the system Audit Log. Both successful and unsuccessful attempts can be audited. Like their DACL object-specific ACE cousins, system audit-object ACEs specify a GUID indicating the types of objects or subobjects that the ACE applies to and an optional GUID that controls propagation of the ACE to particular child object types. If a SACL is null, no auditing takes place on the object. (Security auditing is described later in this chapter.) The inheritance flags that apply to DACL ACEs also apply to system audit and system audit-object ACEs.

Figure 8-4 is a simplified picture of a file object and its DACL.

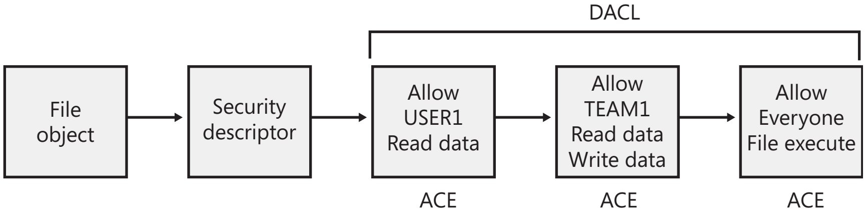


Figure 8-4 Discretionary access-control list (DACL)

As shown in Figure 8-4, the first ACE allows USER1 to query the file. The second ACE allows members of the group TEAM1 to have read and write access to the file, and the third ACE grants all other users (Everyone) execute access.

EXPERIMENT: Viewing a Security Descriptor

Most executive subsystems rely on the object manager's default security functionality to manage security descriptors for their objects. The object manager's default security functions use the security descriptor pointer to store security descriptors for such objects. For example, the Process Manager uses default security so the object manager stores process and thread security descriptors in the object headers of process and thread objects, respectively. The security descriptor pointer of events, mutexes and semaphores, also store their security descriptors. You can use live kernel debugging to view the security descriptors of these objects once you locate their object header, as outlined in the following steps:

1. Start the kernel debugger.
2. Type **!process 0 0** to show the address of a Winlogon. (There will be multiple Winlogon’s running on a system with more than one active Terminal Server session.) Then type **!process** again with the address of one of the Winlogon processes:

```
kd> !process 88d0c020
PROCESS 88d0c020  SessionId: 0  Cid: 0bdc    Peb: 7ffdf000  ParentCid: 00a8
    DirBase: 59a58000  ObjectTable: e21df920  HandleCount:  62.
    Image: winlogon.exe
```

3. Type **!object** with the address following the word PROCESS of the output of the previous command as the argument to show the object data structure:

```
kd> !object 88d0c020
Object: 88d0c020  Type: (8a0ed388) Process
    ObjectHeader: 88d0c008
    HandleCount: 1  PointerCount: 32
```

4. Type **dt _OBJECT_HEADER** and the address of the object header field from the previous command’s output to show the object header data structure, including the security descriptor pointer value:

```
lkd> dt _OBJECT_HEADER 88d0c008
   +0x000 PointerCount     : 36
   +0x004 HandleCount      : 1
   +0x004 NextToFree       : 0x00000001
   +0x008 Type             : 0x8a0ed388
   +0x00c NameInfoOffset   : 0 ''
   +0x00d HandleInfoOffset : 0 ''
   +0x00e QuotaInfoOffset  : 0 ''
   +0x00f Flags            : 0x20 ' '
   +0x010 ObjectCreateInfo : 0x89e596a0
   +0x010 QuotaBlockCharged : 0x89e596a0
   +0x014 SecurityDescriptor : 0xe242b864
   +0x018 Body             : _QUAD
```

5. Security descriptor pointers in the object header use the low three bits as flags, so type the following command to dump the security descriptor by entering the

address displayed in the object header structure, but with the low three bits removed:

```
lkd> !sd 0xe242b864 & -8
->Revision: 0x1
->Sbz1    : 0x0
->Control : 0x8004
            SE_DACL_PRESENT
            SE_SELF_RELATIVE
->Owner   : S-1-5-21-1787744166-3910675280-2727264193-500
->Group   : S-1-5-21-1787744166-3910675280-2727264193-513
->Dacl    :
->Dacl    : ->AclRevision: 0x2
->Dacl    : ->Sbz1       : 0x0
->Dacl    : ->AclSize    : 0x40
->Dacl    : ->AceCount   : 0x2
->Dacl    : ->Sbz2       : 0x0
->Dacl    : ->Ace[0]: ->AceType: ACCESS_ALLOWED_ACE_TYPE
->Dacl    : ->Ace[0]: ->AceFlags: 0x0
->Dacl    : ->Ace[0]: ->AceSize: 0x24
->Dacl    : ->Ace[0]: ->Mask : 0x001f0fff
->Dacl    : ->Ace[0]: ->SID: S-1-5-21-1787744166-3910675280-2727264193-500

->Dacl    : ->Ace[1]: ->AceType: ACCESS_ALLOWED_ACE_TYPE
->Dacl    : ->Ace[1]: ->AceFlags: 0x0
->Dacl    : ->Ace[1]: ->AceSize: 0x14
->Dacl    : ->Ace[1]: ->Mask : 0x001f0fff
->Dacl    : ->Ace[1]: ->SID: S-1-5-18

->Sacl    :  is NULL
```

The security descriptor contains two access-allowed ACEs with one specifying the administrator's account (recognizable by its RID of 500) and the other specifying the System account (which is always S-1-5-18). Without decoding the bits that are set in the access masks of the ACEs and determining which process access types they map to, it's not possible to easily say what accesses each account has to Winlogon's process object. However, doing so, with the help of SDK header files, reveals that both have full access.

ACL Assignment

To determine which DACL to assign to a new object, the security system uses the first applicable rule of the following four assignment rules:

1. If a caller explicitly provides a security descriptor when creating the object, the security system applies it to the object. If the object has a name and resides in a container object (for example, a named event object in the \BaseNamedObjects object manager namespace directory), the system merges any inheritable ACEs (ACEs that might propagate from the object's container) into the DACL unless the security descriptor has the SE_DACL_PROTECTED flag set, which prevents inheritance.

2. If a caller doesn't supply a security descriptor and the object has a name, the security system looks at the security descriptor in the container in which the new object name is stored. Some of the object directory's ACEs might be marked as inheritable, meaning that they should be applied to new objects created in the object directory. If any of these inheritable ACEs are present, the security system forms them into an ACL, which it attaches to the new object. (Separate flags indicate ACEs that should be inherited only by container objects rather than by objects that aren't containers.)
3. If no security descriptor is specified and the object doesn't inherit any ACEs, the security system retrieves the default DACL from the caller's access token and applies it to the new object. Several subsystems on Windows have hard-coded DACLs that they assign on object creation (for example, services, LSA, and SAM objects).
4. If there is no specified descriptor, no inherited ACEs, and no default DACL, the system creates the object with no DACL, which allows everyone (all users and groups) full access to the object. This rule is the same as the third rule in which a token contains a null default DACL.

The rules the system uses when assigning a SACL to a new object are similar to those used for DACL assignment, with some exceptions. The first is that inherited system audit ACEs don't propagate to objects with security descriptors marked with the SE_SACL_PROTECTED flag (similar to the SE_DACL_PROTECTED flag, which protects DACLs). Second, if there are no specified security audit ACEs and there is no inherited SACL, no SACL is applied to the object. This behavior is different from that used to apply default DACLs because tokens don't have a default SACL.

When a new security descriptor containing inheritable ACEs is applied to a container, the system automatically propagates the inheritable ACEs to the security descriptors of child objects. (Note that a security descriptor's DACL doesn't accept inherited DACL ACEs if its SE_DACL_PROTECTED flag is enabled, and its SACL doesn't inherit SACL ACEs if the descriptor has the SE_SACL_PROTECTED flag set.) The order with which inheritable ACEs are merged with an existing child object's security descriptor is such that any ACEs that were explicitly applied to the ACL are kept ahead of ACEs that the object inherits. The system uses the following rules for propagating inheritable ACEs:

- If a child object with no DACL inherits an ACE, the result is a child object with a DACL containing only the inherited ACE.
- If a child object with an empty DACL inherits an ACE, the result is a child object with a DACL containing only the inherited ACE.
- For objects in Active Directory only, if an inheritable ACE is removed from a parent object, automatic inheritance removes any copies of the ACE inherited by child objects.
- For objects in Active Directory only, if automatic inheritance results in the removal of all ACEs from a child object's DACL, the child object has an empty DACL rather than no DACL.

As you'll soon discover, the order of ACEs in an ACL is an important aspect of the Windows security model.

Note Inheritance is generally not directly supported by the object stores such as file systems, the Registry, or Active Directory. Windows APIs that support inheritance, including *SetSecurityInfo* and *SetNamedSecurityInfo*, do so by invoking appropriate functions within the security inheritance support DLL (\Windows\System32\Ntmarta.Dll) that know how to traverse those object stores.

Determining Access

Two algorithms are used for determining access to an object:

- One to determine the maximum access allowed to the object, a form of which is exported to user mode with the Windows *GetEffectiveRightsFromAcl* function
- One to determine whether a specific desired access is allowed, which can be done with the Windows *AccessCheck* function or the *AccessCheckByType* function

The first algorithm examines the entries in the DACL as follows:

1. If the object has no DACL (a null DACL), the object has no protection and the security system grants all access.
2. If the caller has the take-ownership privilege, the security system grants write-owner access before examining the DACL. (Take-ownership privilege and write-owner access are explained in a moment.)
3. If the caller is the owner of the object, the read-control and write-DACL access rights are granted.
4. For each access-denied ACE that contains a SID that matches one in the caller's access token, the ACE's access mask is removed from the granted-access mask.
5. For each access-allowed ACE that contains a SID that matches one in the caller's access token, the ACE's access mask is added to the granted-access mask being computed, unless that access has already been denied.

When all the entries in the DACL have been examined, the computed granted-access mask is returned to the caller as the maximum allowed access to the object. This mask represents the total set of access types that the caller will be able to successfully request when opening the object.

The preceding description applies only to the kernel-mode form of the algorithm. The Windows version implemented by *GetEffectiveRightsFromAcl* differs in that it doesn't perform step 2, and it considers a single user or group SID rather than an access token.

The second algorithm is used to determine whether a specific access request can be granted, based on the caller's access token. Each open function in the Windows API that deals with securable objects has a parameter that specifies the desired access mask, which is the last component of the security equation. To determine whether the caller has access, the following steps are performed:

1. If the object has no DACL (a null DACL), the object has no protection and the security system grants the desired access.
2. If the caller has the take-ownership privilege, the security system grants write-owner access and then examines the DACL. However, if write-owner access was the only access requested by a caller with take-ownership privilege, the security system grants that access and never examines the DACL.
3. If the caller is the owner of the object, the read-control and write-DACL access rights are granted. If these rights were the only access rights that caller requested, access is granted without examining the DACL.
4. Each ACE in the DACL is examined from first to last. An ACE is processed if one of the following conditions is satisfied:
    a. The ACE is an access-deny ACE, and the SID in the ACE matches an *enabled* SID (SIDs can be enabled or disabled) or a deny-only SID in the caller's access token.
    b. The ACE is an access-allowed ACE, and the SID in the ACE matches an enabled SID in the caller's token that isn't of type deny-only.
    c. It is the second pass through the descriptor for restricted-SID checks, and the SID in the ACE matches a restricted SID in the caller's access token.
5. If it is an access-allowed ACE, the rights in the access mask in the ACE that were requested are granted; if all the requested access rights have been granted, the access check succeeds. If it is an access-denied ACE and any of the requested access rights are in the denied-access rights, access is denied to the object.
6. If the end of the DACL is reached and some of the requested access rights still haven't been granted, access is denied.
7. If all accesses are granted but the caller's access token has at least one restricted SID, the system rescans the DACL's ACEs looking for ACEs with access-mask matches for the accesses the user is requesting and a match of the ACE's SID with any of the caller's restricted SIDs. Only if both scans of the DACL grant the requested access rights is the user granted access to the object.

The behavior of both access-validation algorithms depends on the relative ordering of allow and deny ACEs. Consider an object with only two ACEs where one ACE specifies that a certain user is allowed full access to an object and the other ACE denies the user access. If the allow ACE precedes the deny ACE, the user can obtain full access to the object, but if the order is reversed, the user can not gain any access to the object.

Older Windows functions, such as *AddAccessAllowedAce*, add ACEs to the end of a DACL, which usually isn't the desired behavior, so most Windows applications prior to Windows 2000 have no choice but to construct DACLs manually, with deny ACEs placed at the front of the list. Several Windows functions, such as *SetSecurityInfo* and *SetNamedSecurityInfo*, apply ACEs in the preferred order of deny ACEs preceding allow ACEs. Note that the security editor dialog boxes with which you edit permissions on NTFS files and registry keys, for example, use these functions. *SetSecurityInfo* and *SetNamedSecurityInfo* also apply ACE inheritance rules to the security descriptor on which they are applied.

Figure 8-5 shows an example access validation demonstrating the importance of ACE ordering. In the example, access is denied a user wanting to open a file even though an ACE in the object's DACL grants the access because the ACE denying the user access (by virtue of the user's membership in the Writers group) precedes the ACE granting access.

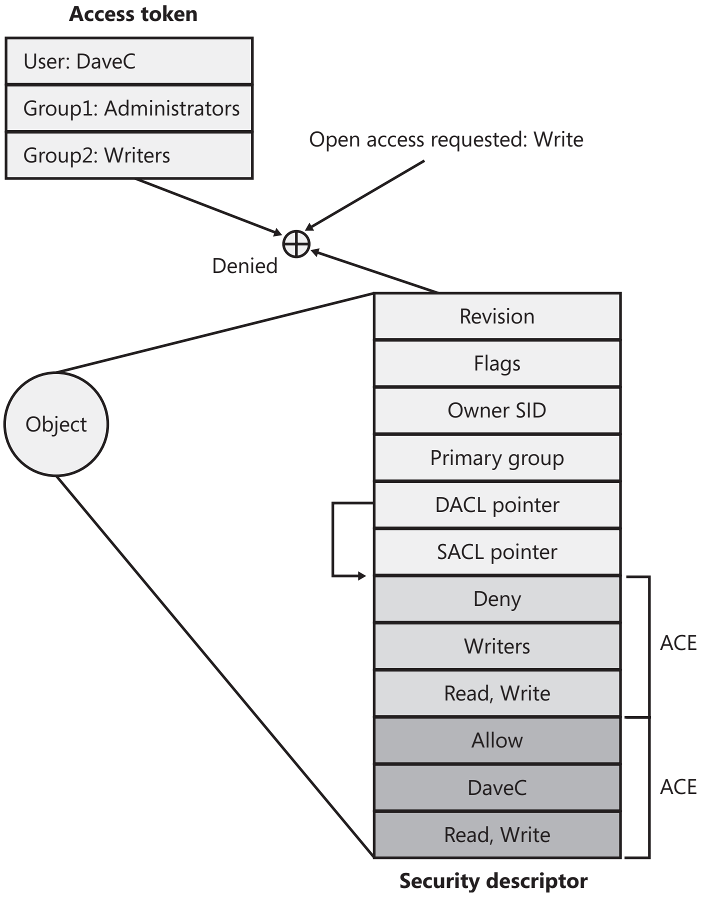


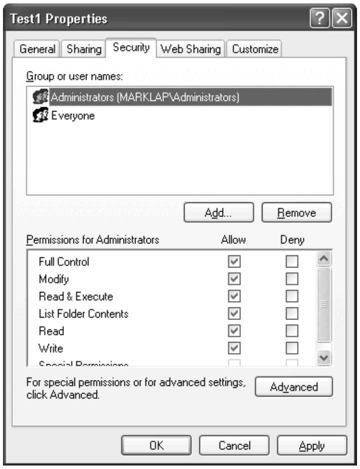
Figure 8-5 Access validation example

As we stated earlier, because it wouldn't be efficient for the security system to process the DACL every time a process uses a handle, the SRM makes this access check only when a handle is opened, not each time the handle is used. Thus, once a process successfully opens a handle, the security system can't revoke the access rights that have been granted, even if the object's DACL changes. Also keep in mind that because kernel-mode code uses pointers rather than handles to access objects, the access check isn't performed when the operating system uses objects. In other words, the Windows executive "trusts" itself in a security sense.

The fact that an object's owner is always granted write-DACL access to an object means that users can never be prevented from accessing the objects they own. If, for some reason, an object had an empty DACL (no access), the owner would still be able to open the object with write-DACL access and then apply a new DACL with the desired access permissions.

A Warning Regarding the GUI Security Editors

When you use the GUI permissions editors to modify security settings on a file, Registry, Active Directory object, or other securable object, the main security dialog box shows you a potentially misleading view of the security that's applied to the object. The top portion of the dialog box shows in alphabetical ordering the groups and users that have ACEs in the object's access control list. If you allow Full Control to the Administrators group and deny the Everyone group Full Control, the list might lead you to believe that the Administrators group access-allowed ACE precedes the Everyone deny ACE because that's the order in which they appear. However, as we've said, the editors place deny ACEs before allow ACEs when they apply the ACL to the object.

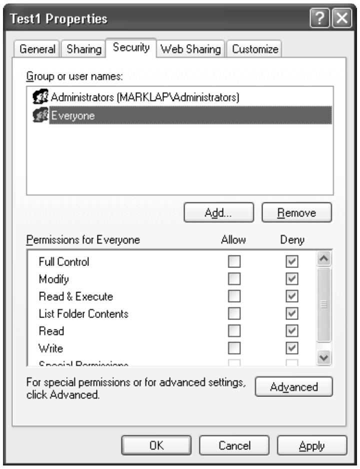


Group or user names:

Administrators (MARKLAP\Administrators)
Everyone

Permissions for Administrators	Allow	Deny
Full Control	☑	□
Modify	☑	□
Read & Execute	☑	□
List Folder Contents	☑	□
Read	☑	□
Write	☑	□
Special Permissions	□	□

For special permissions or for advanced settings, click Advanced.



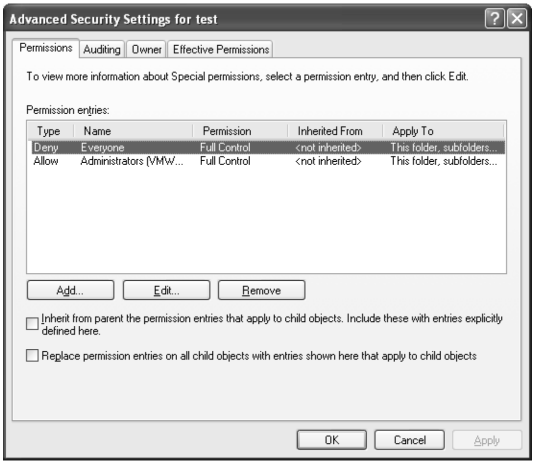
Group or user names:

Administrators (MARKLAP\Administrators)
Everyone

Permissions for Everyone	Allow	Deny
Full Control	□	☑
Modify	□	☑
Read & Execute	□	☑
List Folder Contents	□	☑
Read	□	☑
Write	□	☑
Special Permissions	□	□

For special permissions or for advanced settings, click Advanced.

The Permissions tab of the Advanced Security Settings dialog box shows the order of ACEs in the DACL. However, even this dialog box can be confusing because a complex DACL can have deny ACEs for various accesses followed by allow ACEs for other access types.



Advanced Security Settings for test

Permissions | Auditing | Owner | Effective Permissions

To view more information about Special permissions, select a permission entry, and then click Edit.

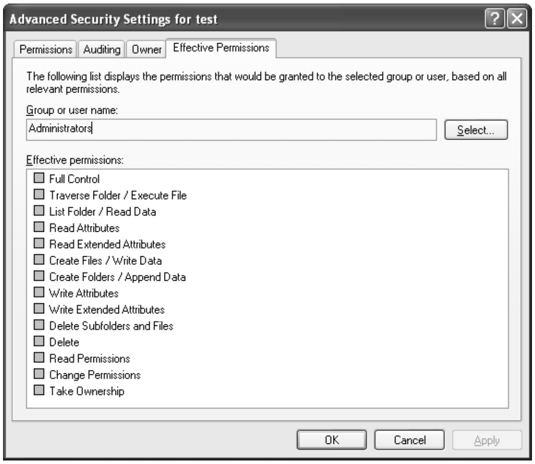
Permission entries:

Type	Name	Permission	Inherited From	Apply To
Deny	Everyone	Full Control	<not inherited>	This folder, subfolders...
Allow	Administrators (VMW...	Full Control	<not inherited>	This folder, subfolders...

□ Inherit from parent the permission entries that apply to child objects. Include these with entries explicitly defined here.

□ Replace permission entries on all child objects with entries shown here that apply to child objects

The only definitive way to know what accesses a particular user or group will have to an object (other than having that user or a member of the group try to access the object) is to use the Effective Permissions tab of the dialog box that displays when you press the Advanced button on the dialog box. Enter the name of the user or group you want to check and the dialog box shows you what permissions they are allowed for the object.



Advanced Security Settings for test

Permissions | Auditing | Owner | Effective Permissions

The following list displays the permissions that would be granted to the selected group or user, based on all relevant permissions.

Group or user name:

Administrators

Effective permissions:

- □ Full Control
- □ Traverse Folder / Execute File
- □ List Folder / Read Data
- □ Read Attributes
- □ Read Extended Attributes
- □ Create Files / Write Data
- □ Create Folders / Append Data
- □ Write Attributes
- □ Write Extended Attributes
- □ Delete Subfolders and Files
- □ Delete
- □ Read Permissions
- □ Change Permissions
- □ Take Ownership

The AuthZ API

The AuthZ Windows API, which was introduced in Windows XP, implements the same security model as the Security Reference Monitor, but implements it totally in user mode in the \Windows\System32\Authz.Dll library. This gives applications that want to protect their own private objects, such as database tables, the ability to leverage the Windows security model without incurring the cost of user-mode to kernel-mode transitions that they would make if they relied on the Security Reference Monitor.

The AuthZ API uses standard security descriptor data structures, SIDs, and privileges. Instead of using tokens to represent clients, AuthZ uses the AUTHZ_CLIENT_CONTEXT. AuthZ includes user-mode equivalents of all access-check and Windows security functions—for example, *AuthzAccessCheck* is the AuthZ version of the *AccessCheck* Windows API that uses the *SeAccessCheck* Security Reference Monitor function.

Another advantage available to applications that use AuthZ is that they can direct AuthZ to cache the results of security checks to improve subsequent checks that use the same client context and security descriptor.

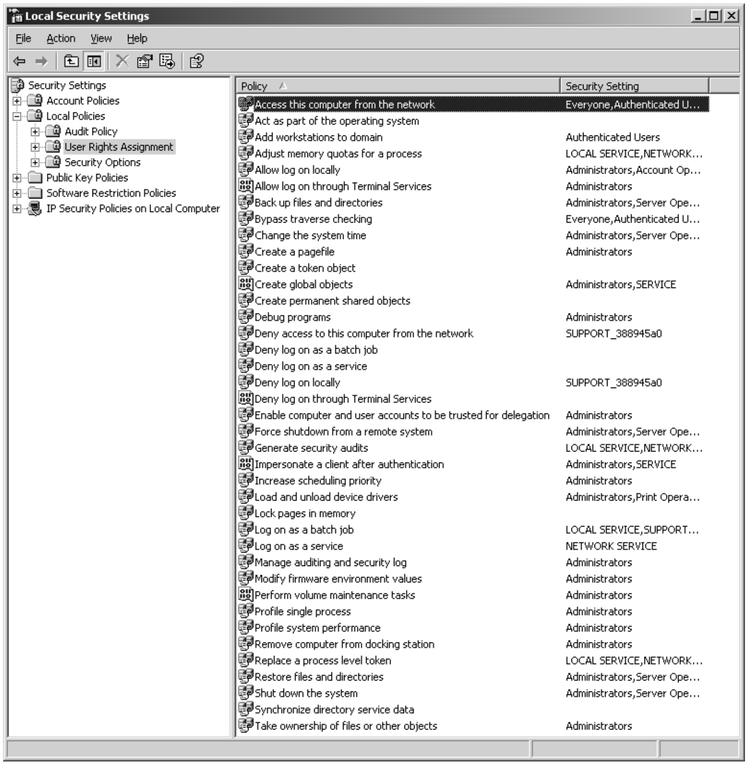
AuthZ is fully documented in the Platform SDK.

Account Rights and Privileges

Many operations performed by processes as they execute cannot be authorized through object access protection because they do not involve interaction with a particular object. For example, the ability to bypass security checks when opening files for backup is an attribute of an account, not of a particular object. Windows uses both privileges and account rights to allow a system administrator to control what accounts can perform security-related operations.

A privilege is the right of an account to perform a particular system-related operation, such as shutting down the computer or changing the system time. An account right grants or denies the account to which it's assigned the ability to perform a particular type of logon, such as a local logon or interactive logon, to a computer.

A system administrator assigns privileges to groups and accounts using tools such as the Active Directory Users and Groups MMC snap-in for domain accounts or the Local Security Policy Editor. You access the Local Security Policy Editor in the Administrative Tools folder of the Control Panel or the Start menu (if you've configured your Start menu to contain an Administrative Tools link). Figure 8-6 shows User Rights Assignment configuration in the Local Security Policy Editor, which displays the complete list of privileges and account rights available on Windows Server 2003. Note that the tool makes no distinction between privileges and account rights. However, you can differentiate between them because any user right that contains the word *logon* is an account privilege.



Policy	Security Setting
Access this computer from the network	Everyone,Authenticated U...
Act as part of the operating system	
Add workstations to domain	Authenticated Users
Adjust memory quotas for a process	LOCAL SERVICE,NETWORK...
Allow log on locally	Administrators,Account Op...
Allow log on through Terminal Services	Administrators
Back up files and directories	Administrators,Server Ope...
Bypass traverse checking	Everyone,Authenticated U...
Change the system time	Administrators,Server Ope...
Create a pagefile	Administrators
Create a token object	
Create global objects	Administrators,SERVICE
Create permanent shared objects	
Debug programs	Administrators
Deny access to this computer from the network	SUPPORT_388945a0
Deny log on as a batch job	
Deny log on as a service	
Deny log on locally	SUPPORT_388945a0
Deny log on through Terminal Services	
Enable computer and user accounts to be trusted for delegation	Administrators
Force shutdown from a remote system	Administrators,Server Ope...
Generate security audits	LOCAL SERVICE,NETWORK...
Impersonate a client after authentication	Administrators,SERVICE
Increase scheduling priority	Administrators
Load and unload device drivers	Administrators,Print Opera...
Lock pages in memory	
Log on as a batch job	LOCAL SERVICE,SUPPORT...
Log on as a service	NETWORK SERVICE
Manage auditing and security log	Administrators
Modify firmware environment values	Administrators
Perform volume maintenance tasks	Administrators
Profile single process	Administrators
Profile system performance	Administrators
Remove computer from docking station	Administrators
Replace a process level token	LOCAL SERVICE,NETWORK...
Restore files and directories	Administrators,Server Ope...
Shut down the system	Administrators,Server Ope...
Synchronize directory service data	
Take ownership of files or other objects	Administrators

Figure 8-6 Local Security Policy Editor User Rights Assignment

Account Rights

Account Rights are not enforced by the Security Reference Monitor, nor are they stored in tokens. The function responsible for logon is *LsaLogonUser*. WinLogon, for example, calls the *LogonUser* API when a user logs on interactively to a computer and *LogonUser* calls *LsaLogonUser*. The function takes a parameter that indicates the type of logon being performed, which includes interactive, network, batch, service, terminal server client, and unlock.

In response to logon requests, the Local Security Authority (LSA) retrieves account rights assigned to a user from the LSA policy database at the time that a user attempts to log on to the system. LSA checks the logon type against the account rights assigned to the user account logging on and denies the logon if the account does not have the right that permits the logon type or it has the right that denies the logon type. Table 8-4 lists the user rights defined by Windows.

Windows applications can add and remove user rights from an account by using the *LsaAddAccountRights* and *LsaRemoveAccountRights* functions, and they can determine what rights are assigned to an account with *LsaEnumerateAccountRights*.

Table 8-4 Account Rights

User Right	Role
Deny logon interactively, Allow logon interactively	Used for interactive logons that originate on the local machine
Deny logon over the network, Allow logon over the network	Used for logons that originate from a remote machine
Deny logon through Terminal Services[1], Allow logon through Terminal Services[1]	Used for logons through a Terminal Server client
Deny logon as a service, Allow logon as a service	Used by the Service Control Manager when starting a service in a particular user account
Deny logon as a batch job, Allow logon as a batch job	Used when performing a logon of type *batch*

[1] Introduced in Windows XP

Privileges

The number of privileges defined by the operating system has grown over time. Unlike user rights, which are enforced in one place by LSA, different privileges are defined by different components and enforced by those components. For example, the debug privilege, which allows a process to bypass security checks when opening a handle to another process with the *OpenProcess* Windows API, is checked for by the Process Manager. Table 8-5 is a full list of privileges, and it describes how and when system components check for them.

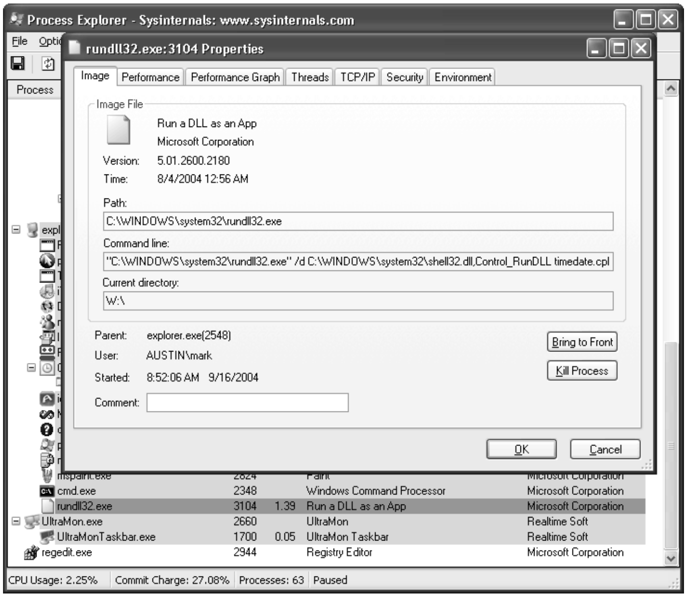
When a component wants to check a token to see whether a privilege is present, it uses the *PrivilegeCheck* or *LsaEnumerateAccountRights* APIs if running in user mode and *SeSinglePrivilegeCheck* or *SePrivilegeCheck* if running in kernel mode. The privilege-related APIs are not account-right aware, but the account-right APIs are privilege-aware.

Unlike account rights, privileges can be enabled and disabled. For a privilege check to succeed, the privilege must be in the specified token and it must be enabled. The idea behind this scheme is that privileges should be enabled only when their use is required so that a process cannot inadvertently perform a privileged security operation.

EXPERIMENT: Seeing a Privilege Get Enabled

By following these steps, you can see that the Date and Time Control Panel applet enables the SeSystemTimePrivilege privilege in anticipation of you using its interface to change the clock or the date of the computer.

1. Log on to a system with an account that has the “Change the system time” user right (such as an account that is a member of the Power Users or Administrators group).
2. Run Process Explorer, and set the refresh rate to Paused.
3. View the Security tab of the process properties dialog for your Explorer shell process. You should see that the SeChangeSystemTimePrivilege privilege is disabled, which is indicated by its flag and the fact that it’s shown in gray.
4. Open the Date and Time applet from the Control Panel, and refresh Process Explorer. A new Rundll process will appear with a green highlight.
5. Open the process properties for the Rundll process (by double-clicking on the process), and verify that the command line contains the text “Timedate.Cpl”. The presence of this argument tells Rundll, which is a Control Panel DLL hosting process, to load the DLL that implements the user interface (UI) that enables you to change the time and date.

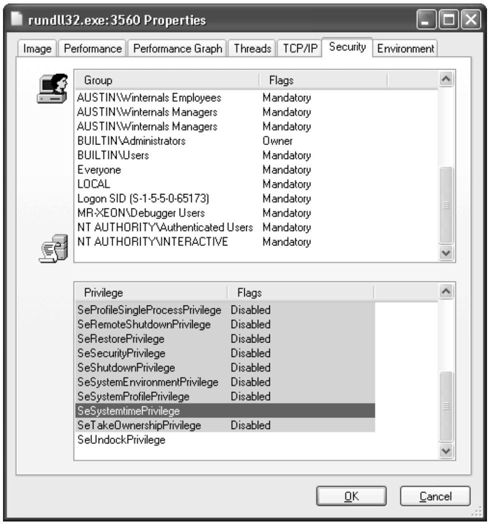


Run a DLL as an App
Microsoft Corporation
Version: 5.01.2600.2180
Time: 8/4/2004 12:56 AM
Path: C:\WINDOWS\system32\rundll32.exe
Command line: "C:\WINDOWS\system32\rundll32.exe" /d C:\WINDOWS\system32\shell32.dll,Control_RunDLL timedate.cpl
Current directory: W:\
Parent: explorer.exe(2548)
User: AUSTIN\mark
Started: 8:52:06 AM 9/16/2004
Comment:

```
mspaint.exe            2824         Paint                       Microsoft Corporation
cmd.exe                2348         Windows Command Processor   Microsoft Corporation
rundll32.exe           3104  1.39   Run a DLL as an App         Microsoft Corporation
UltraMon.exe           2660         UltraMon                    Realtime Soft
UltraMonTaskbar.exe    1700  0.05   UltraMon Taskbar            Realtime Soft
regedit.exe            2944         Registry Editor             Microsoft Corporation
```

CPU Usage: 2.25% | Commit Charge: 27.08% | Processes: 63 | Paused

6. Switch to the Security tab of the Rundll process properties dialog box, and you should see that SeSystemTimePrivilege is enabled.



Group	Flags
AUSTIN\Winternals Employees	Mandatory
AUSTIN\Winternals Managers	Mandatory
AUSTIN\Winternals Managers	Mandatory
BUILTIN\Administrators	Owner
BUILTIN\Users	Mandatory
Everyone	Mandatory
LOCAL	Mandatory
Logon SID (S-1-5-5-0-65173)	Mandatory
MR-XEON\Debugger Users	Mandatory
NT AUTHORITY\Authenticated Users	Mandatory
NT AUTHORITY\INTERACTIVE	Mandatory

Privilege	Flags
SeProfileSingleProcessPrivilege	Disabled
SeRemoteShutdownPrivilege	Disabled
SeRestorePrivilege	Disabled
SeSecurityPrivilege	Disabled
SeShutdownPrivilege	Disabled
SeSystemEnvironmentPrivilege	Disabled
SeSystemProfilePrivilege	Disabled
SeSystemtimePrivilege	
SeTakeOwnershipPrivilege	Disabled
SeUndockPrivilege	

Table 8-5 **Privileges**

Privilege	User Right	Privilege Usage
SeAssignPrimaryTokenPrivilege	Replace a process-level token	Checked for by various components, such as *NtSetInformationJob*, that set a process's token.
SeAuditPrivilege	Generate security audits	Required to generate events for the Security event log with the *ReportEvent* API.
SeBackupPrivilege	Backup files and directories	Causes NTFS to grant the following access to any file or directory, regardless of the security descriptor that's present: READ_CONTROL ACCESS_SYSTEM_SECURITY FILE_GENERIC_READ FILE_TRAVERSE Note that when opening a file for backup, the caller must specify the FILE_FLAG_BACKUP_SEMANTICS flag.
SeChangeNotifyPrivilege	Bypass traverse checking	Used by NTFS to avoid checking permissions on intermediate directories of a multilevel directory lookup. Also used by file systems when applications register for notification of changes to the file system structure.
SeCreateGlobalPrivilege[1]	Create global objects	Required for a process to create section and symbolic link objects in the directories of the Object Manager namespace that are assigned to a different session than the caller.

Table 8-5 **Privileges**

Privilege	User Right	Privilege Usage
SeCreatePagefilePrivilege	Create a pagefile	Checked for by *NtCreatePagingFile*, which is the function used to create a new paging file.
SeCreatePermanentPrivilege	Create permanent shared objects	Checked for by the object manager when creating a permanent object (one that doesn't get deallocated when there are no more references to it).
SeCreateTokenPrivilege	Create a token object	*NtCreateToken*, the function that creates a token object, checks for this privilege.
SeDebugPrivilege	Debug programs	If the caller has this privilege enabled, the Process Manager allows access to any process using *NtOpenProcess*, regardless of the process's security descriptor.
SeEnableDelegationPrivilege[2]	Enable computer and user accounts to be trusted for delegation	Used by Active Directory services to delegate authenticated credentials.
SeImpersonatePrivilege[1]	Impersonate a client after authentication	The Process Manager checks for this when a thread wants to use a token for impersonation and the token represents a different user than that of the thread's process token.
SeIncreaseBasePriorityPrivilege	Increase scheduling priority	Checked for by the Process Manager and is required to raise the priority of a process.
SeIncreaseQuotaPrivilege	Windows 2000: Increase quotas Windows XP/ Windows Server 2003: Adjust memory quotas for a process	Enforced when changing the Registry's quota (Windows 2000 only), a process's working set thresholds, and a process's paged and nonpaged pool quotas.
SeLoadDriverPrivilege	Load and unload device drivers	Checked for by the *NtLoadDriver* and *NtUnloadDriver* driver functions.
SeLockMemoryPrivilege	Lock pages in memory	Checked for by *NtLockVirtualMemory*, the kernel implementation of VirtualLock.
SeMachineAccountPrivilege	Add workstations to the domain	Checked for by the Security Accounts Manager on a domain controller when creating a machine account in a domain.
SeManageVolumePrivilege[2]	Perform volume maintenance tasks	Enforced by file system drivers during a volume open operation, which is required to perform disk checking and defragmenting activities.
SeProfileSingleProcessPrivilege	Profile single process	Checked for by prefetcher's function that returns prefetch information for an individual process.
SeRemoteShutdownPrivilege	Force shutdown from a remote system	Winlogon checks that remote callers of the *InitiateSystemShutdown* function have this privilege.

Table 8-5 **Privileges**

Privilege	User Right	Privilege Usage
SeRestorePrivilege	Restore files and directories	This privilege causes NTFS to grant the following access to any file or directory, regardless of the security descriptor that's present: WRITE_DAC WRITE_OWNER ACCESS_SYSTEM_SECURITY FILE_GENERIC_WRITE FILE_ADD_FILE FILE_ADD_SUBDIRECTORY DELETE Note that when opening a file for restore, the caller must specify the FILE_FLAG_BACKUP_SEMANTICS flag.
SeSecurityPrivilege	Manage auditing and security log	Required to access the SACL of a security descriptor, read and clear the security event log.
SeShutdownPrivilege	Shut down the system	This privilege is checked for by *NtShutdownSystem* and *NtRaiseHardError*, which presents a system error dialog box on the interactive console.
SeSyncAgentPrivilege	Synchronize directory service data	Required to use the LDAP directory synchronization services and allows the holder to read all objects and properties in the directory, regardless of the protection on the objects and properties.
SeSystemEnvironmentPrivilege	Modify firmware environment variables	Required by *NtSetSystemEnvironmentValue* and *NtQuerySystemEnvironmentValue* to modify and read firmware environment variables using the HAL.
SeSystemProfilePrivilege	Profile system performance	Checked for by *NtCreateProfile*, the function used to perform profiling of the system. This is used by the Kernprof tool, for example.
SeSystemtimePrivilege	Change the system time	Required to change the time or date.
SeTakeOwnership	Take ownership of files and other objects	Required to take ownership of an object without being granted discretionary access.
SeTcbPrivilege	Act as part of the operating system	Checked for by the Security Reference Monitor when the session ID is set in a token, by the Plug and Play Manager for Plug and Play event creation and management, the Windows 2000 implementation of LogonUser, BroadcastSystemMessageEx when called with BSM_ALLDESKTOPS, and LsaRegisterLogonProcess.
SeUndockPrivilege	Remove computer from a docking station	Checked for by the user-mode Plug and Play Manager when either a computer undock is initiated or a device eject request is made.

[1] Introduced in Windows Server 2003

[2] Introduced in Windows XP

EXPERIMENT: The Bypass Traverse Checking Privilege

If you are a systems administrator, you must be aware of the Bypass Traverse Checking privilege (internally called SeNotifyPrivilege) and its implications. This experiment demonstrates that not understanding its behavior can lead to improperly applied security.

1. Create a folder and, within that folder, a new text file with some sample text.
2. Navigate in Explorer to the new file, and go to the Security tab of its Properties dialog box. Click the Advanced button, and deselect the check box that controls inheritance. Select Copy when you are prompted as to whether you want to remove or copy inherited permissions.
3. Next, modify the security of the new folder so that your account does not have any access to the folder. Do this by selecting your account and selecting all the Deny boxes in the permissions list.
4. Run Notepad, and browse using the File|Open dialog box to the new directory. You should be denied access to the directory.
5. In the File Name field of the open dialog box, type the full path of the new file. The file should open.

If your account does not have the Bypass Traverse Checking privilege, NTFS performs an access check on each directory of the path to a file when you try to open a file, which results in you being denied access to the file in this example.

Super Privileges

Several privileges are so powerful that a user to which they are assigned is effectively a "super user" who has full control over a computer. These privileges can be used in an infinite number of ways to gain unauthorized access to otherwise off-limit resources and to perform unauthorized operations. However, we'll focus on using the privilege to execute code that grants the user privileges not assigned to the user, with the knowledge that this capability can be leveraged to perform any operation on the local machine that the user desires. This section lists the privileges and discusses the ways that they can be exploited. Other privileges, such as Lock Pages In Physical Memory, can be exploited for denial of service attacks on a system, but are not discussed.

- **Debug programs** A user with this privilege can open any process on the system without regard to the security descriptor present on the process. The user could implement a program that opens the LSASS process, for example, copy executable code into its address space, and then inject a thread with the *CreateRemoteThread* Windows API to execute the injected code in a more-privileged security context. The code could grant the user additional privileges and group memberships.

- **Take ownership** This privilege allows a holder to take ownership of any securable object by writing his own SID into the owner field of the object's security descriptor. Recall that an owner is always granted permission to read and modify the DACL of the security descriptor, so a process with this privilege could modify the DACL to grant itself full access to the object and then close and reopen the object with full access. This would allow the owner to see sensitive data and to even replace system files that execute as part of normal system operation, such as LSASS, with his own programs that grant a user elevated privileges.
- **Restore files and directories** A user assigned this privilege can replace any file on the system with her own. She could exploit this power by replacing system files as described in the preceding paragraph.
- **Load and unload device drivers** A malicious user could use this privilege to load a device driver into the system. Device drivers are considered trusted parts of the operating system that can execute within it with System account credentials, so a driver could launch privileged programs that assign the user other rights.
- **Create a token object** This privilege can be used in the obvious way to generate tokens that represent arbitrary user accounts with arbitrary group membership and privilege assignment.
- **Act as part of operating system** LsaRegisterLogonProcess, the function a process calls to establish a trusted connection to Lsass, checks for this privilege. A malicious user with this privilege can establish a trusted-Lsass connection and then execute LsaLogonUser, a function used to create new logon sessions. LsaLogonUser requires a valid user name and password and accepts an optional list of SIDs that it adds to the initial token created for a new logon session. The user could therefore use her own user name and password to create a new logon session that includes the SIDs of more privileged groups or users in the resulting token.

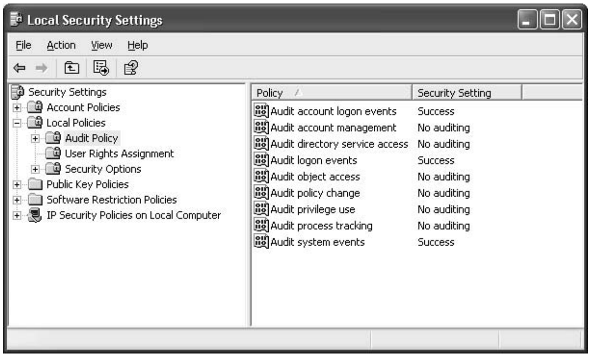
Note that the use of elevated privilege does not extend past the machine boundary to the network, because any interaction with another computer requires authentication with a domain controller and validation of domain passwords. Domain passwords are not stored on a computer either in plain text or encrypted form, so they are not accessible to malicious code.

Security Auditing

The object manager can generate audit events as a result of an access check, and Windows functions available to user applications can generate them directly. Kernel-mode code is always allowed to generate an audit event. Two privileges, SeSecurityPrivilege and SeAuditPrivilege, relate to auditing. A process must have the SeSecurityPrivilege privilege to manage the security Event Log and to view or set an object's SACL. Processes that call audit system services, however, must have the SeAuditPrivilege privilege to successfully generate an audit record.

The audit policy of the local system controls the decision to audit a particular type of security event. The audit policy, also called the local security policy, is one part of the security policy

Lsass maintains on the local system, and it is configured with the Local Security Policy editor shown in Figure 8-7.



- Security Settings
  - Account Policies
  - Local Policies
    - Audit Policy
    - User Rights Assignment
    - Security Options
  - Public Key Policies
  - Software Restriction Policies
  - IP Security Policies on Local Computer

Policy	Security Setting
Audit account logon events	Success
Audit account management	No auditing
Audit directory service access	No auditing
Audit logon events	Success
Audit object access	No auditing
Audit policy change	No auditing
Audit privilege use	No auditing
Audit process tracking	No auditing
Audit system events	Success

Figure 8-7 Security Policy Editor audit policy configuration

Lsass sends messages to the SRM to inform it of the auditing policy at system initialization time and when the policy changes. Lsass is responsible for receiving audit records generated based on the audit events from the SRM, editing the records, and sending them to the Event Logger. Lsass (instead of the SRM) sends these records because it adds pertinent details, such as the information needed to more completely identify the process that is being audited.

The SRM sends audit records via its LPC connection to Lsass. The Event Logger then writes the audit record to the security Event Log. In addition to audit records the SRM passes, both Lsass and the SAM generate audit records that Lsass sends directly to the Event Logger, and the AuthZ API allows for applications to generate application-defined audits. Figure 8-8 depicts this overall flow.

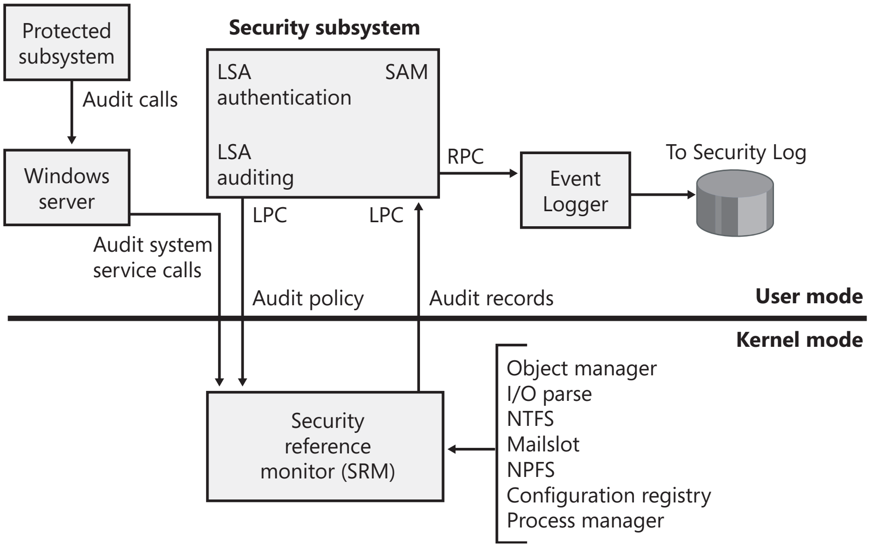


Figure 8-8 Flow of security audit records

Audit records are put on a queue to be sent to the LSA as they are received—they are not submitted in batches. The audit records are moved from the SRM to the security subsystem in one of two ways. If the audit record is small (less than the maximum LPC message size), it is sent as an LPC message. The audit records are copied from the address space of the SRM to the address space of the Lsass process. If the audit record is large, the SRM uses shared memory to make the message available to Lsass and simply passes a pointer in an LPC message.

Figure 8-9 brings together the concepts covered so far in this chapter by illustrating the basic process and thread security structures. In the figure, notice that the process object and the thread objects have ACLs, as do the access-token objects themselves. Also in this figure, thread 2 and thread 3 each has an impersonation token, whereas thread 1 defaults to the process access token.

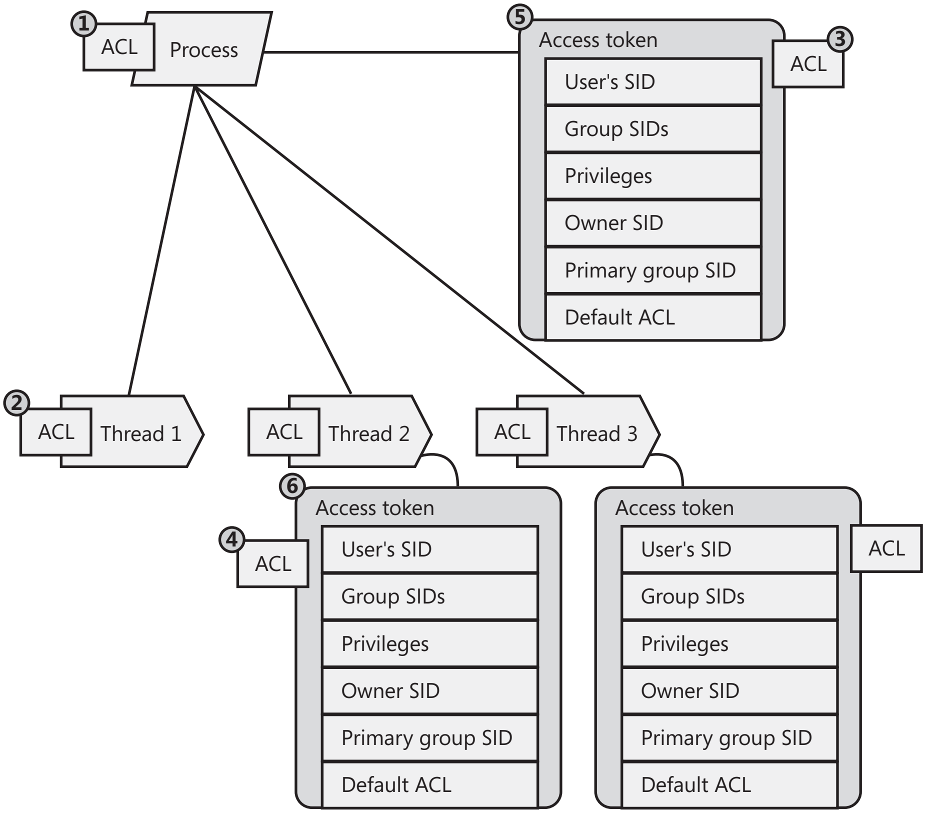


Figure 8-9 Process and thread security structures

Logon

Interactive logon (as opposed to network logon) occurs through the interaction of the logon process (Winlogon), Lsass, one or more authentication packages, and the SAM or Active Directory. *Authentication packages* are DLLs that perform authentication checks. Kerberos is the Windows authentication package for interactive logon to a domain, and MSV1_0 is the Windows authentication package for interactive logon to a local computer, for domain logons to trusted pre-Windows 2000 domains, and for when no domain controller is accessible.

Winlogon is a trusted process responsible for managing security-related user interactions. It coordinates logon, starts the user's first process at logon, handles logoff, and manages various other operations relevant to security, including entering passwords at logon, changing passwords, and locking and unlocking the workstation. The Winlogon process must ensure that operations relevant to security aren't visible to any other active processes. For example, Winlogon guarantees that an untrusted process can't get control of the desktop during one of these operations and thus gain access to the password.

Winlogon relies on a Graphical Identification and Authentication (GINA) DLL to obtain a user's account name and password. The default GINA is Msgina (\Windows\System32\Msgina.dll). Msgina presents the standard Windows logon dialog box. Allowing for other GINAs to replace Msgina enables Windows to use different user identification mechanisms. For example, a third party might supply a GINA that uses a thumbprint recognition device to identify users and extract their passwords from an encrypted database.

Winlogon is the only process that intercepts logon requests from the keyboard. After obtaining a username and password from the GINA, Winlogon calls Lsass to authenticate the user attempting to log on. If the user is authenticated, the logon process activates a logon shell on behalf of that user. The interaction between the components involved in logon is illustrated in Figure 8-10.

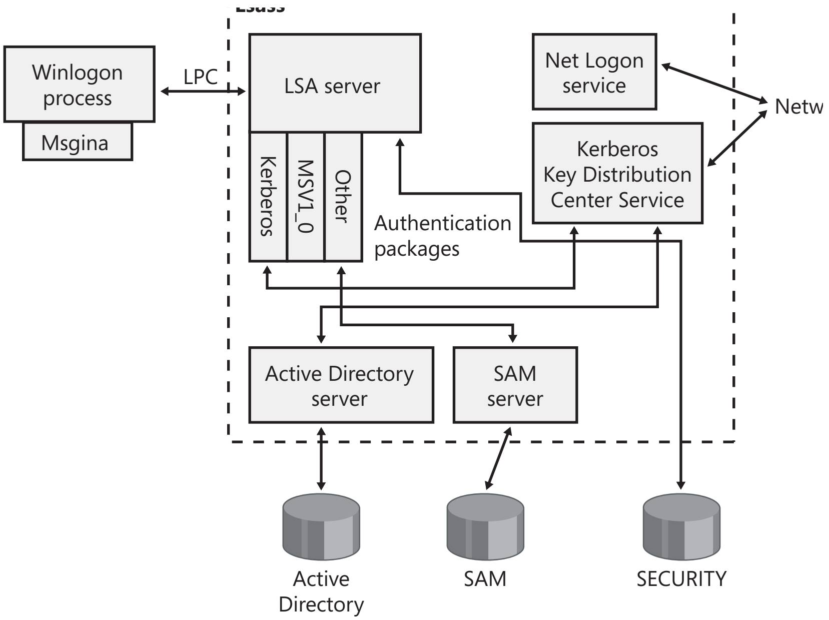


Figure 8-10 Components involved in logon

In addition to supporting alternate GINAs, Winlogon can load additional network provider DLLs that need to perform secondary authentication. This capability allows multiple network providers to gather identification and authentication information all at one time during normal logon. A user logging into a Windows system might simultaneously be authenticated on a UNIX server. That user would then be able to access resources of the UNIX server from the Windows machine without requiring additional authentication. Such a capability is known as one form of *single sign-on*.

Winlogon Initialization

During system initialization, before any user applications are active, Winlogon performs the following steps to ensure that it controls the workstation once the system is ready for user interaction:

1. Creates and opens an interactive window station (for example, \Windows\WindowStations\WinSta0 in the Object Manager namespace) to represent the keyboard, mouse, and monitor. Winlogon creates a security descriptor for the station that has one and only one ACE containing only the System SID. This unique security descriptor ensures that no other process can access the workstation unless explicitly allowed by Winlogon.
2. Creates and opens two desktops: an application desktop (\Windows\WinSta0\Default, also known as the interactive desktop) and a Winlogon desktop (\Windows\WinSta0\Winlogon, also known as the secure desktop). The security on the Winlogon desktop is created so that only Winlogon can access that desktop. The other desktop allows both Winlogon and users to access them. This arrangement means that any time the Winlogon desktop is active, no other process has access to any active code or data associated with the desktop. Windows uses this feature to protect the secure operations that involve passwords and locking and unlocking the desktop.
3. Before anyone logs on to a computer, the visible desktop is Winlogon's. After a user logs on, pressing Ctrl+Alt+Delete switches the desktop from Default to Winlogon. (This explains why all the windows on your interactive desktop seem to disappear when you press Ctrl+Alt+Delete and then return when you dismiss the Windows Security dialog box.) Thus, the SAS always brings up a secure desktop controlled by Winlogon.
4. Establishes an LPC connection with Lsass's *LsaAuthenticationPort*. This connection will be used for exchanging information during logon, logoff, and password operations and is made by calling *LsaRegisterLogonProcess*.

   Winlogon then performs the following Windows operations to set up the window environment:
5. It initializes and registers a window class data structure that associates a Winlogon procedure with the window it subsequently creates.

6. It registers the SAS associating it with the window just created, guaranteeing that Winlogon's window procedure is called whenever the user enters the SAS. This measure prevents Trojan horse programs from gaining control of the screen when the SAS is entered.
7. It registers the window so that the procedure associated with this window gets called if a user logs off or if the screen saver times out. The Windows subsystem checks to verify that the process requesting notification is the Winlogon process.

How SAS Is Implemented

The SAS is secure because no application can intercept the Ctrl+Alt+Delete keystroke combination or prevent Winlogon from receiving it. Winlogon uses a documented Windows API, *RegisterHotKey*, to reserve the Ctrl+Alt+Delete key combination so that whenever the Windows input system sees the combination, it sends a special message to a window that Winlogon creates to receive such notifications. The keystrokes that map to a registered hot key are otherwise not sent to any process other than the one that registered it, and only the thread that registered a hot key can unregister it (using the *UnregisterHotKey* API), so a Trojan horse application cannot deregister Winlogon's ownership of the SAS.

A Windows function, *SetWindowsHook*, enables an application to install a hook procedure that's invoked every time a keystroke is pressed, even before hot keys are processed, and it allows the hook to squash keystrokes. However, the Windows hot key processing code contains a special case for Ctrl+Alt+Delete that disables hooks so that the keystroke sequence can't be intercepted. In addition, if the interactive desktop is locked, only hot keys owned by Winlogon are processed.

Once the Winlogon desktop is created during initialization, it becomes the active desktop. When the Winlogon desktop is active, it is always locked. Winlogon unlocks its desktop only to switch to the application desktop or the screen-saver desktop. (Only the Winlogon process can lock or unlock a desktop.)

User Logon Steps

Logon begins when a user presses the SAS (Ctrl+Alt+Delete). After the SAS is pressed, Winlogon calls the GINA to obtain a username and password. Winlogon also creates a unique local logon SID for this user that it assigns to this instance of the desktop (keyboard, screen, and mouse). Winlogon passes this SID to Lsass as part of the *LsaLogonUser* call. If the user is successfully logged on, this SID will be included in the logon process token—a step that protects access to the desktop. For example, another logon to the same account but on a different system will be unable to write to the first machine's desktop because this second logon won't be in the first logon's desktop token.

When the username and password have been entered, Winlogon retrieves a handle to a package by calling the Lsass function *LsaLookupAuthenticationPackage*. Authentication packages are listed in the Registry under HKLM\SYSTEM\CurrentControlSet\Control\Lsa. Winlogon passes logon information to the authentication package via *LsaLogonUser*. Once a package authenticates a user, Winlogon continues the logon process for that user. If none of the authentication packages indicates a successful logon, the logon process is aborted.

Windows uses two standard authentication packages for interactive logons: Kerberos and MSV1_0. The default authentication package on a stand-alone Windows system is MSV1_0 (\Windows\System32\Msv1_0.dll), an authentication package that implements LAN Manager 2 protocol. Lsass also uses MSV1_0 on domain-member computers to authenticate to pre-Windows 2000 domains and computers that can't locate a domain controller for authentication. (Laptop computers that are disconnected from the network fall into this latter category.) The Kerberos authentication package, \Windows\System32\Kerberos.dll, is used on computers that are members of Windows domains. The Windows Kerberos package, with the cooperation of Kerberos services running on a domain controller, supports version 5, revision 6, of the Kerberos protocol. This protocol is based on Internet RFC 1510. (Visit the Internet Engineering Task Force [IETF] Web site, *www.ietf.org*, for detailed information on the Kerberos standard.)

The MSV1_0 authentication package takes the username and a hashed version of the password and sends a request to the local SAM to retrieve the account information, which includes the password, the groups to which the user belongs, and any account restrictions. MSV1_0 first checks the account restrictions, such as hours or type of accesses allowed. If the user can't log on because of the restrictions in the SAM database, the logon call fails and MSV1_0 returns a failure status to the LSA.

MSV1_0 then compares the hashed password and username to that stored by the SAM. In the case of a cached domain logon, MSV1_0 accesses the cached information by using Lsass functions that store and retrieve "secrets" from the LSA database (the SECURITY hive of the registry). If the information matches, MSV1_0 generates an LUID for the logon session and creates the logon session by calling Lsass, associating this unique identifier with the session and passing the information needed to ultimately create an access token for the user. (Recall that an access token includes the user's SID, group SIDs, and assigned privileges.)

> **Note** MSV1_0 does not cache a user's entire password hash in the registry because that would enable someone with physical access to the system to easily compromise a user's domain account and gain access to encrypted files and to network resources the user is authorized to access. Instead, it caches half of the hash. The cached half-hash is sufficient to verify that user's password is correct, but it isn't sufficient to gain access to EFS keys and to authenticate as the user on a domain because these actions require the full hash.

If MSV1_0 needs to authenticate using a remote system, as when a user logs on to a trusted pre–Windows 2000 domain, MSV1_0 uses the Netlogon service to communicate with an instance of Netlogon on the remote system. Netlogon on the remote system interacts with the MSV1_0 authentication package on that system, passing back authentication results to the system on which the logon is being performed.

The basic control flow for Kerberos authentication is the same as the flow for MSV1_0. However, in most cases, domain logons are performed from member workstations or servers (rather than on a domain controller), so the authentication package must communicate across the network as part of the authentication process. The package does so by communicating via the Kerberos TCP/IP port (port 88) with the Kerberos service on a domain controller. The Kerberos Key Distribution Center service (\Windows\System32\Kdcsvc.dll), which implements the Kerberos authentication protocol, runs in the LSASS process on domain controllers.

After validating hashed username and password information with Active Directory's user account objects (using the Active Directory server \Windows\System32\Ntdsa.dll), Kdcsvc returns domain credentials to LSASS, which returns the result of the authentication and the user's domain logon credentials (if the logon was successful) across the network to the system where the logon is taking place.

> **Note** This description of Kerberos authentication is highly simplified, but it highlights the roles of the various components involved. Although the Kerberos authentication protocol plays a key role in distributed domain security in Windows, its details are outside the scope of this book.

After a logon has been authenticated, LSASS looks in the local policy database for the user's allowed access, including interactive, network, batch, or service process. If the requested logon doesn't match the allowed access, the logon attempt will be terminated. LSASS deletes the newly created logon session by cleaning up any of its data structures and then returns failure to Winlogon, which in turn displays an appropriate message to the user. If the requested access is allowed, LSASS adds the appropriate additional security IDs (such as Everyone, Interactive, and the like). It then checks its policy database for any granted privileges for all the IDs for this user and adds these privileges to the user's access token.

When LSASS has accumulated all the necessary information, it calls the executive to create the access token. The executive creates a primary access token for an interactive or service logon and an impersonation token for a network logon. After the access token is successfully created, LSASS duplicates the token, creating a handle that can be passed to Winlogon, and closes its own handle. If necessary, the logon operation is audited. At this point, LSASS returns success to Winlogon along with a handle to the access token, the LUID for the logon session, and the profile information, if any, that the authentication package returned.

EXPERIMENT: Listing Active Logon Sessions

As long as at least one token exists with a given logon session LUID, Windows considers the logon session to be active. You can use the LogonSessions tool from *www.sysinternals.com*, which uses the *LsaEnumerateLogonSessions* function (documented in the Platform SDK) to list the active logon sessions:

```
 C:\>logonsessions
LogonSessions 1.0
Copyright (C) 2004 Bryce Cogswell
Sysinternals - www.sysinternals.com

[0] Logon session 00000000:000003e7:
    User name:    AUSTIN\MARKLAP$
    Auth package: NTLM
    Logon type:   (none)
    Session:      0
    Sid:          S-1-5-18
    Logon time:   5/13/2004 9:05:43 AM
    Logon server:
    DNS Domain:
    UPN:
[1] Logon session 00000000:0000d37d:
    User name:
    Auth package: NTLM
    Logon type:   (none)
    Session:      0
    Sid:          (none)
    Logon time:   8/17/2004 5:08:10 PM
    Logon server:
    DNS Domain:
    UPN:
[2] Logon session 00000000:000003e4:
    User name:    NT AUTHORITY\NETWORK SERVICE
    Auth package: Negotiate
    Logon type:   Service
    Session:      0
    Sid:          S-1-5-20
    Logon time:   8/17/2004 5:08:18 PM
    Logon server:
    DNS Domain:
    UPN:
…

[8] Logon session 00000000:0079da73:
    User name:    MARKLAP\Administrator
    Auth package: MICROSOFT_AUTHENTICATION_PACKAGE_V1_0
    Logon type:   NetworkCleartext
    Session:      0
    Sid:          S-1-5-21-1787744166-3910675280-2727264193-500
    Logon time:   5/13/2004 12:14:20 PM
    Logon server: MARKLAP
    DNS Domain:
    UPN:
```

Information reported for a session includes the SID and name of the user associated with the session, as well as session's authentication package and logon time. Note that the Negotiate authentication package, seen in logon session 2 in the preceding output, will attempt to authenticate via Kerberos or NTLM, depending on which is most appropriate for the authentication request.

The LUID for a session is displayed on the "Logon Session" line of each session block, and using the Handle utility (also available from *www.sysinternals.com*), you can find the tokens that represent a particular logon session. For example, to find the tokens for logon session 8 in the example output just shown, you could enter this command:

```
C:\>handle -a 79da73
  43c: Token         MARKLAP\Administrator:79da73
```

Winlogon then looks in the registry at the value HKLM\SOFTWARE\Microsoft\Windows NT\Current Version\Winlogon\Userinit and creates a process to run whatever the value of that string is. (This value can be several .exes separated by commas). The default value is Userinit.exe, which loads the user profile and then creates a process to run whatever the value is of HKCU\SOFTWARE\Microsoft\Windows NT\Current Version\Winlogon\Shell, if that value exists. That value does not exit by default. If it doesn't exist, Userinit.exe does the same for HKLM\SOFTWARE\Microsoft\Windows NT\Current Version\Winlogon\Shell, which defaults to Explorer.exe. Userinit then exits (which is why Explorer.exe shows up as having no parent when examined in Process Explorer). For more information on the steps followed during the user logon process, see Chapter 5.

Software Restriction Policies

Malicious code such as viruses and worms have become an increasing problem. Windows XP introduced a mechanism called Software Restriction Policies that enables administrators to control what images and scripts execute on their systems. The Software Restriction Policies node of the Security Policy Editor, shown in Figure 8-11, serves as the management interface for a machine's code execution policies, although per-user policies are also possible using domain group policies.

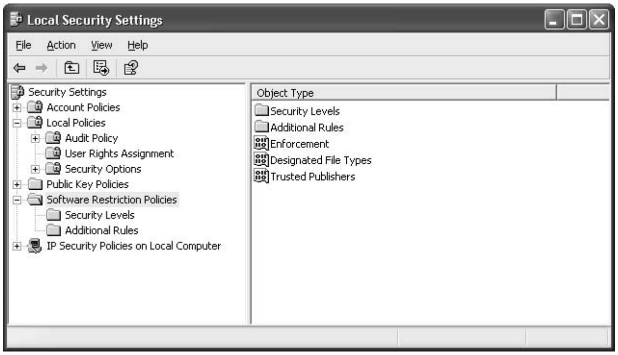


Figure 8-11 Software Restriction Policy configuration

Several global policy settings appear beneath the Software Restriction Policies node.

- The *Enforcement* policy configures whether restriction policies apply to libraries, such as DLLs, and whether policies apply to users only or to administrators as well.
- The *Designated File Types* policy records the extensions for files that are considered executable code.
- *Trusted Publishers* control who can select which certificate publishers are trusted.

When configuring a policy for a particular script or image, an administrator can direct the system to recognize it using its path, its hash, its Internet Zone (as defined by Internet Explorer) or its cryptographic certificate, and she can specify whether it is associated with the *Disallowed* or *Unrestricted* security policy.

Enforcement of Software Restriction Policies takes place within various components where files are treated as containing executable code. Some of these components are listed here:

- The user-mode Windows *CreateProcess* function in \Windows\System32\Kernel32.dll enforces it for executable images.
- The DLL loading code of Ntdll (\Windows\System32\Ntdll.dll) enforces it for DLLs.
- The Windows command prompt (\Windows\System32\Cmd.exe) enforces it for batch file execution.
- Windows Scripting Host components that start scripts—\Windows\System32\Cscript.exe (for command-line scripts), \Windows\System32\Wscript.exe (for UI scripts), and \Windows\System32\Scrobj.dll (for script objects)—enforce it for script execution.

Each of these components determines whether the restriction policies are enabled by reading the Registry value HKEY_LOCAL_MACHINE\Software\Microsoft\Policies\Windows\Safer\CodeIdentifiers\TransparentEnabled, which if set to 1, indicates that policies are in effect. Then it determines whether the code it's about to execute matches one of the rules specified in a subkey of the CodeIdentifiers key, and if so, whether or not the execution should be allowed. If there is no match, the default policy, as specified in the *DefaultLevel* value of the CodeIdentifiers key, determines whether the execution is allowed.

Software Restriction Policies are a powerful tool for preventing the unauthorized access of code and scripts, but only if properly applied. Unless the default policy is set to disallow execution, a user can make minor changes to an image that's been marked as disallowed so that he can bypass the rule and execute it. For example, a user can change an innocuous byte of a process image so that a hash rule fails to recognize it or copy a file to a different location to avoid a path-based rule.

EXPERIMENT: Watching Software Restriction Policy Enforcement

You can indirectly see Software Restriction Policies being enforced by watching accesses to the Registry when you attempt to execute an image that you've disallowed.

1. Run secpol.msc to open the Local Security Policy Editor, and navigate to the Software Restriction Policies node.
2. Choose Create New Policies from the context menu if no policies are defined.
3. Create a path-based disallow restriction policy for \Windows\System32\Notepad.exe.
4. Run Regmon, and set an include filter for "Safer." (See Chapter 4 for a description of Regmon.)
5. Open a command prompt, and run Notepad from the prompt.

Your attempt to run Notepad should result in a message telling you that you cannot execute the specified program, and Regmon should show the command prompt (cmd.exe) querying the local machine restriction policies.

Conclusion

Windows provides an extensive array of security functions that meet the key requirements of both government agencies and commercial installations. In this chapter, we've taken a brief tour of the internal components that are the basis of these security features.

In the next chapter, we'll look at the last major executive component considered in this book: the I/O system.

Chapter 9

I/O System

The Microsoft Windows I/O system consists of several executive components that together manage hardware devices and provide interfaces to hardware devices for applications and the system. In this chapter, we'll first list the design goals of the I/O system, which have influenced its implementation. We'll then cover the components that make up the I/O system, including the I/O manager, Plug and Play (PnP) manager, and power manager. Then we'll examine the structure and components of the I/O system and the various types of device drivers. We'll look at the key data structures that describe devices, device drivers, and I/O requests, after which we'll describe the steps necessary to complete I/O requests as they move through the system. Finally, we'll present the way device detection, driver installation, and power management work.

I/O System Components

The design goals for the Windows I/O system are to provide an abstraction of devices, both hardware (physical) and software (virtual or logical), to applications with the following features:

- Uniform security and naming across devices to protect shareable resources. (See Chapter 8 for a description of the Windows security model.)
- High performance asynchronous packet-based I/O to allow for the implementation of scalable applications.
- Services that allow drivers to be written in a high-level language and easily ported between different machine architectures.
- Layering and extensibility to allow for the addition of drivers that transparently modify the behavior of other drivers or devices, without requiring any changes to the driver whose behavior or device is modified.
- Dynamic loading and unloading of device drivers so that drivers can be loaded on-demand and not consume system resources when unneeded.
- Support for Plug and Play where the system locates and installs drivers for newly detected hardware and then assigns them hardware resources they require.
- Support for power management so that the system or individual devices can enter low power states.

- Support for multiple installable file systems, including FAT, the CD-ROM file system (CDFS), the Universal Disk Format (UDF) file system, and the Windows file system (NTFS). (See Chapter 12 for more specific information on file system types and architecture.)
- Windows Management Instrumentation support and diagnosability so that drivers can be managed and monitored through WMI applications and scripts. (WMI is described in Chapter 4.)

To implement these features the Windows I/O system consists of several executive components as well as device drivers, which are shown in Figure 9-1.

- The I/O manager is the heart of the I/O system; it connects applications and system components to virtual, logical, and physical devices; and it defines the infrastructure that supports device drivers.
- A device driver typically provides an I/O interface for a particular type of device. Device drivers receive commands routed to them by the I/O manager that are directed at devices they manage, and they inform the I/O manager when those commands complete. Device drivers often use the I/O manager to forward I/O commands to other device drivers that share in the implementation of a device's interface or control.
- The PnP manager works closely with the I/O manager and a type of device driver called a *bus driver* to guide the allocation of hardware resources as well as to detect and respond to the arrival and removal of hardware devices. The PnP manager and bus drivers are responsible for loading a device's driver when the device is detected. When a device is added to a system that doesn't have an appropriate device driver, the executive Plug and Play component calls on the device installation services of a user-mode PnP manager.
- The power manager also works closely with the I/O manager to guide the system, as well as individual device drivers, through power-state transitions.
- Windows Management Instrumentation (WMI) support routines, called the Windows Driver Model (WDM) WMI provider, allow device drivers to indirectly act as providers, using the WDM WMI provider as an intermediary to communicate with the WMI service in user mode. (For more information on WMI, see the section "Windows Management Instrumentation" in Chapter 4.)
- The registry serves as a database that stores a description of basic hardware devices attached to the system as well as driver initialization and configuration settings. (See the "The Registry" section in Chapter 4 for more information.)

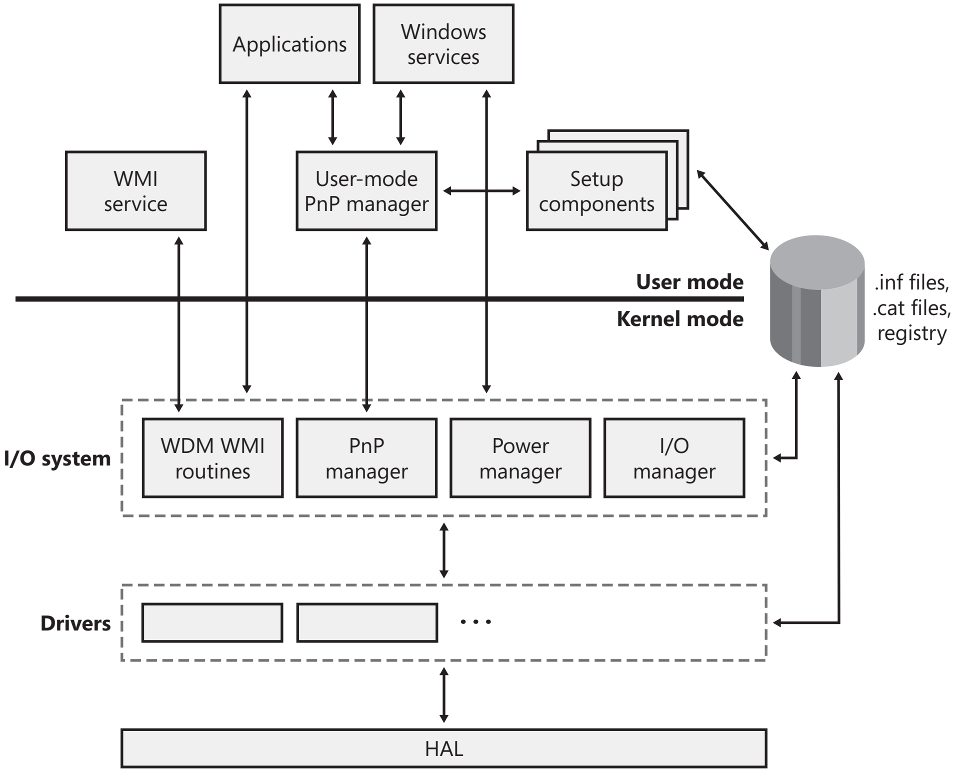


Figure 9-1 I/O system components

- INF files, which are designated by the .inf extension, are driver installation files. INF files are the link between a particular hardware device and the driver that assumes primary control of the device. They are made up of scriptlike instructions describing the device they correspond to, the source and target locations of driver files, required driver-installation registry modifications, and driver dependency information. Digital signatures that Windows uses to verify that a driver file has passed testing by the Microsoft Windows Hardware Quality Lab (WHQL) are stored in .cat files.
- The hardware abstraction layer (HAL) insulates drivers from the specifics of the processor and interrupt controller by providing APIs that hide differences between platforms. In essence, the HAL is the bus driver for all the devices on the computer's motherboard that aren't controlled by other drivers.

The I/O Manager

The *I/O manager* is the core of the I/O system because it defines the orderly framework, or model, within which I/O requests are delivered to device drivers. The I/O system is *packet driven*. Most I/O requests are represented by an *I/O request packet* (IRP), which travels from one I/O system component to another. (As you'll discover in "Fast I/O", fast I/O is the exception; it doesn't use IRPs.) The design allows an individual application thread to manage multiple I/O requests concurrently. An IRP is a data structure that contains information completely describing an I/O request. (You'll find more information about IRPs in the section "I/O Request Packets.")

The I/O manager creates an IRP that represents an I/O operation, passing a pointer to the IRP to the correct driver and disposing of the packet when the I/O operation is complete. In contrast, a driver receives an IRP, performs the operation the IRP specifies, and passes the IRP back to the I/O manager, either for completion or to be passed on to another driver for further processing.

In addition to creating and disposing of IRPs, the I/O manager supplies code that is common to different drivers and that the drivers call to carry out their I/O processing. By consolidating common tasks in the I/O manager, individual drivers become simpler and more compact. For example, the I/O manager provides a function that allows one driver to call other drivers. It also manages buffers for I/O requests, provides timeout support for drivers, and records which installable file systems are loaded into the operating system. There are close to a hundred different routines in the I/O manager that can be called by device drivers.

The I/O manager also provides flexible I/O services that allow environment subsystems, such as Windows and POSIX, to implement their respective I/O functions. These services include sophisticated services for asynchronous I/O that allow developers to build scalable high-performance server applications.

The uniform, modular interface that drivers present allows the I/O manager to call any driver without requiring any special knowledge of its structure or internal details. The operating system treats all I/O requests as if they were directed at a file; the driver converts the requests from requests made to a virtual file to hardware-specific requests. Drivers can also call each other (using the I/O manager) to achieve layered, independent processing of an I/O request.

Besides providing the normal open, close, read, and write functions, the Windows I/O system provides several advanced features, such as asynchronous, direct, buffered, and scatter/gather I/O, which are described in the "Types of I/O" section later in this chapter.

Typical I/O Processing

Most I/O operations don't involve all the components of the I/O system. A typical I/O request starts with an application executing an I/O-related function (for example, reading data from a device) that is processed by the I/O manager, one or more device drivers, and the HAL.

As just mentioned, in Windows, threads perform I/O on virtual files. The operating system abstracts all I/O requests as operations on a virtual file, hiding the fact that the target of an I/O operation might not be a file-structured device. This abstraction generalizes an application's interface to devices. A virtual file refers to any source or destination for I/O that is treated as if it were a file (such as files, directories, pipes, and mailslots). All data that is read or written is regarded as a simple stream of bytes directed to these virtual files. User-mode applications (whether Windows or POSIX) call documented functions, which in turn call internal I/O system functions to read from a file, write to a file, and perform other operations. The I/O manager dynamically directs these virtual file requests to the appropriate device driver. Figure 9-2 illustrates the basic structure of a typical I/O request flow.

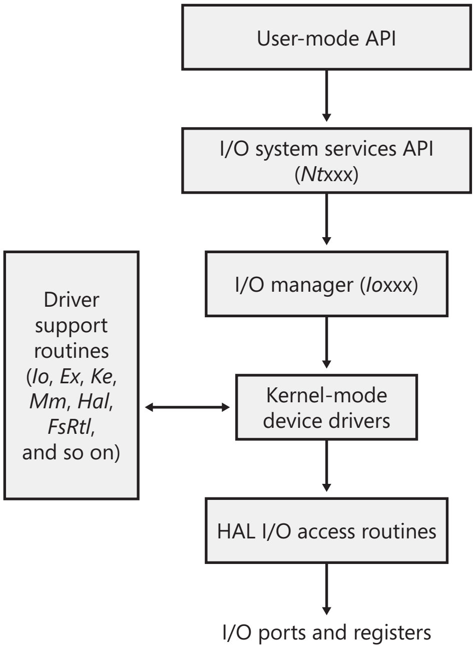

User-mode API
I/O system services API (*Nt*xxx)
I/O manager (*Io*xxx)
Driver support routines (*Io*, *Ex*, *Ke*, *Mm*, *Hal*, *FsRtl*, and so on)
Kernel-mode device drivers
HAL I/O access routines
I/O ports and registers

Figure 9-2 The flow of a typical I/O request

In the following sections, we'll be looking at these components more closely, examining covering the various types of device drivers, how they are structured, how they load and initialize, and how they process I/O requests. Then we'll cover the operation and roles of the PnP manager and the power manager.

Device Drivers

To integrate with the I/O manager and other I/O system components, a device driver must conform to implementation guidelines specific to the type of device it manages and the role it plays in managing the device. In this section, we'll look at the types of device drivers Windows supports as well as the internal structure of a device driver.

Types of Device Drivers

Windows supports a wide range of device driver types and programming environments. Even within a type of device driver, programming environments can differ, depending on the specific type of device for which a driver is intended. The broadest classification of a driver is whether it is a user-mode or kernel-mode driver. Windows supports several types of user-mode drivers:

- *Virtual device drivers* (VDDs) are used to emulate 16-bit MS-DOS applications. They trap what an MS-DOS application thinks are references to I/O ports and translate them into native Windows I/O functions, which are then passed to the actual device driver. Because Windows is a fully protected operating system, user-mode MS-DOS applications can't access hardware directly and thus must go through a real kernel-mode device driver.
- Windows subsystem *printer drivers* translate device-independent graphics requests to printer-specific commands. These commands are then typically forwarded to a kernel-mode port driver such as the parallel port driver (Parport.sys) or the universal serial bus (USB) printer port driver (Usbprint.sys).

In this chapter, the focus is on kernel-mode device drivers. There are many types of kernel-mode drivers, which can be divided into the following basic categories:

- *File system drivers* accept I/O requests to files, and satisfy the requests by issuing their own, more explicit requests to mass storage or network device drivers.
- *Plug and Play drivers* work with hardware and integrate with the Windows power manager and PnP manager. They include drivers for mass storage devices, video adapters, input devices and network adapters.
- *Non–Plug and Play drivers*, also called *kernel extensions*, extend the functionality of the system by providing user-mode access to kernel-mode services and drivers. They do not integrate with the PnP or power managers. Examples include network API and protocol drivers. Regmon's driver, described in Chapter 4, is also an example.

Within the category of kernel-mode drivers are further classifications based on the driver model that the driver adheres to and its role in servicing device requests.

WDM Drivers

WDM drivers are device drivers that adhere to the Windows Driver Model (WDM). WDM includes support for Windows power management, Plug and Play, and WMI, and most Plug and Play drivers adhere to WDM. WDM is implemented on Windows, Windows 98, and Windows Millennium Edition, so WDM drivers are source-compatible between these operating systems and in many cases are also binary compatible. There are three types of WDM drivers:

- *Bus drivers* manage a logical or physical bus. Examples of buses include PCMCIA, PCI, USB, IEEE 1394, and ISA. A bus driver is responsible for detecting and informing the PnP manager of devices attached to the bus it controls as well as managing the power setting of the bus.

- *Function drivers* manage a particular type of device. Bus drivers present devices to function drivers via the PnP manager. The function driver is the driver that exports the operational interface of the device to the operating system. In general, it's the driver with the most knowledge about the operation of the device.
- *Filter drivers* logically layer above or below function drivers, augmenting or changing the behavior of a device or another driver. For example, a keyboard capture utility could be implemented with a keyboard filter driver that layers above the keyboard function driver.

In WDM, no one driver is responsible for controlling all aspects of a particular device. The bus driver is responsible for detecting bus membership changes (device addition or removal), assisting the PnP manager in enumerating the devices on the bus, accessing bus-specific configuration registers, and in some cases, controlling power to devices on the bus. The function driver is generally the only driver that accesses the device's hardware.

> **Note** The role of the HAL in Windows 2000, Windows XP, and Windows Server 2003 differs from the role it had in Windows NT. Prior to Windows 2000, third-party hardware vendors that wanted to add support for hardware buses not natively supported had to implement a custom HAL. Windows 2000, Windows XP, and Windows Server 2003 allow third parties to implement a bus driver to provide support for hardware buses not natively supported.

Layered Drivers

Support for an individual piece of hardware is often divided among several drivers, each providing a part of the functionality required to make the device work properly. In addition to WDM bus drivers, function drivers, and filter drivers, hardware support might be split between the following components:

- *Class drivers* implement the I/O processing for a particular class of devices, such as disk, tape, or CD-ROM, where the hardware interfaces have been standardized and so one driver can serve devices from a wide variety of manufacturers.
- *Port drivers* implement the processing of an I/O request specific to a type of I/O port, such as SCSI, and are implemented as kernel-mode libraries of functions rather than actual device drivers.
- *Miniport drivers* map a generic I/O request to a type of port into an adapter type, such as a specific SCSI adapter. Miniport drivers are actual device drivers that import the functions supplied by a port driver.

An example will help demonstrate how device drivers work. A file system driver accepts a request to write data to a certain location within a particular file. It translates the request into a request to write a certain number of bytes to the disk at a particular "logical" location. It then passes this request (via the I/O manager) to a simple disk driver. The disk driver, in turn, translates the request into a physical location (cylinder/track/sector) on the disk and manipulates the disk heads to write the data. This layering is illustrated in Figure 9-3.

This figure illustrates the division of labor between two layered drivers. The I/O manager receives a write request that is relative to the beginning of a particular file. The I/O manager passes the request to the file system driver, which translates the write operation from a file-relative operation to a starting location (a sector boundary on the disk) and a number of bytes to read. The file system driver calls the I/O manager to pass the request to the disk driver, which translates the request to a physical disk location and transfers the data.

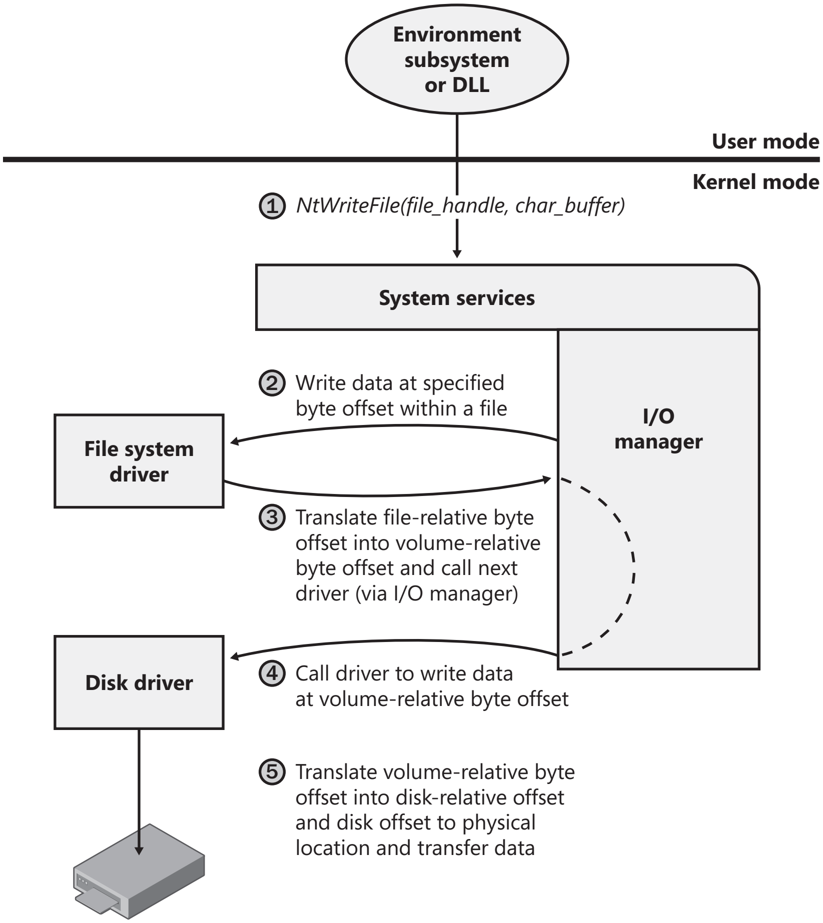

Environment subsystem or DLL
User mode
Kernel mode
1 NtWriteFile(file_handle, char_buffer)
System services
2 Write data at specified byte offset within a file
File system driver
I/O manager
3 Translate file-relative byte offset into volume-relative byte offset and call next driver (via I/O manager)
Disk driver
4 Call driver to write data at volume-relative byte offset
5 Translate volume-relative byte offset into disk-relative offset and disk offset to physical location and transfer data

Figure 9-3 Layering of a file system driver and a disk driver

Because all drivers—both device drivers and file system drivers—present the same framework to the operating system, another driver can easily be inserted into the hierarchy without altering the existing drivers or the I/O system. For example, several disks can be made to seem like a very large single disk by adding a driver. Such a driver exists in Windows to provide fault-tolerant disk support. (Whereas the driver is present on all versions of Windows, fault-tolerant disk support is available only on Windows Server versions.) This logical, volume manager driver is located between the file system and the disk drivers, as shown in Figure 9-4. Volume manager drivers are described in more detail in Chapter 10.

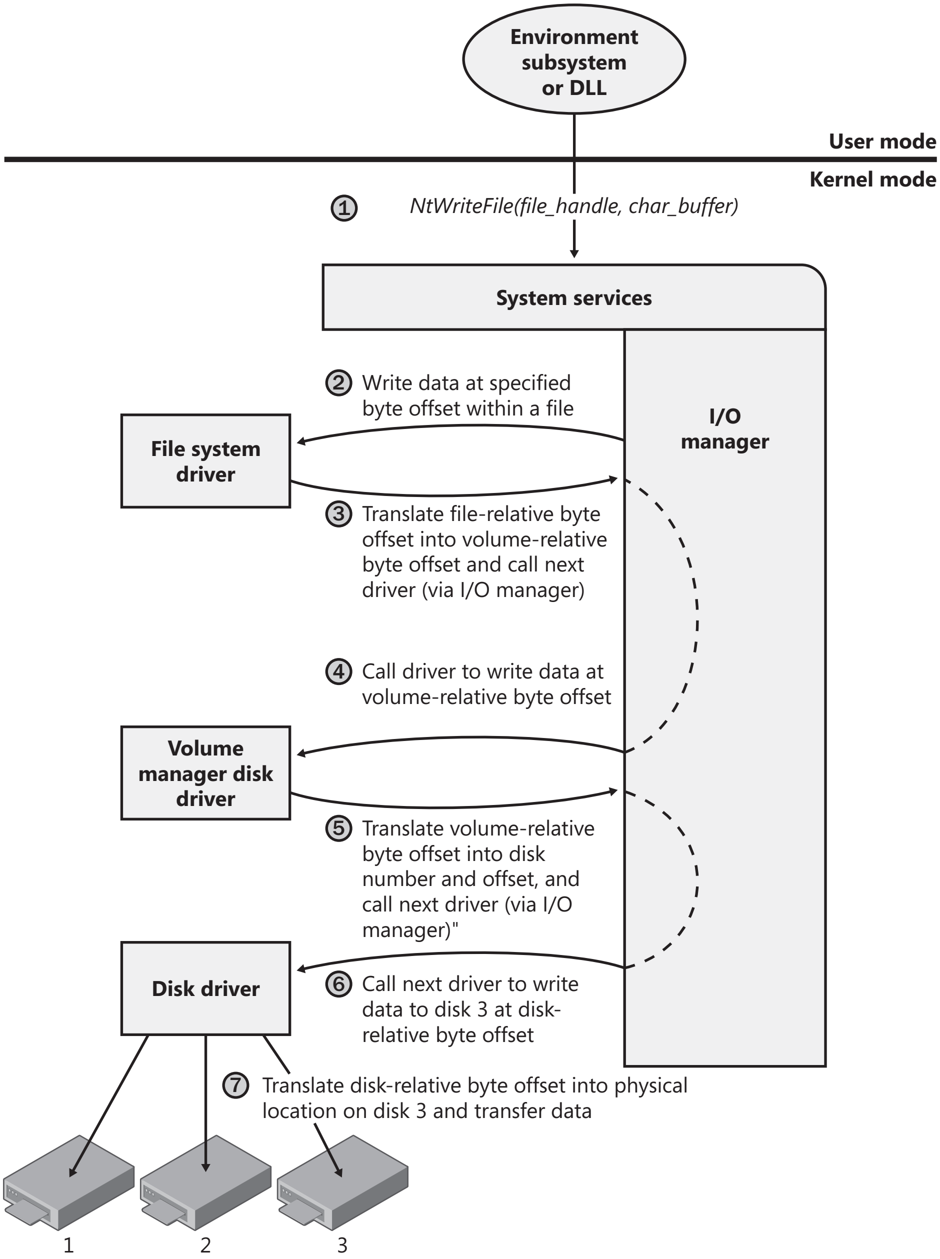
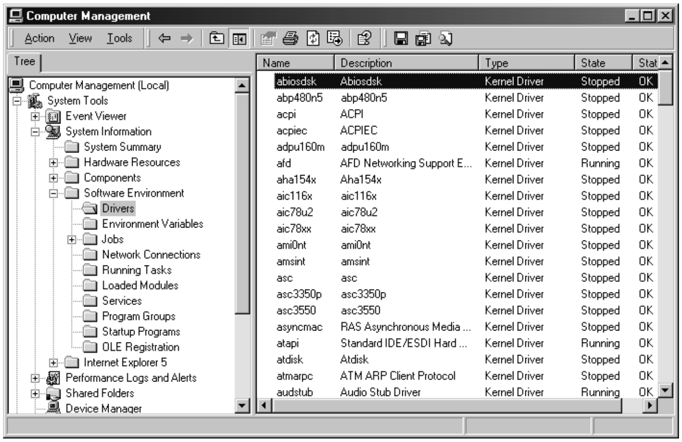

Environment subsystem or DLL
User mode
Kernel mode
1 NtWriteFile(file_handle, char_buffer)
System services
I/O manager
2 Write data at specified byte offset within a file
File system driver
3 Translate file-relative byte offset into volume-relative byte offset and call next driver (via I/O manager)
4 Call driver to write data at volume-relative byte offset
Volume manager disk driver
5 Translate volume-relative byte offset into disk number and offset, and call next driver (via I/O manager)"
Disk driver
6 Call next driver to write data to disk 3 at disk-relative byte offset
7 Translate disk-relative byte offset into physical location on disk 3 and transfer data
1
2
3

Figure 9-4 Adding a layered driver

EXPERIMENT: Viewing the Loaded Driver List

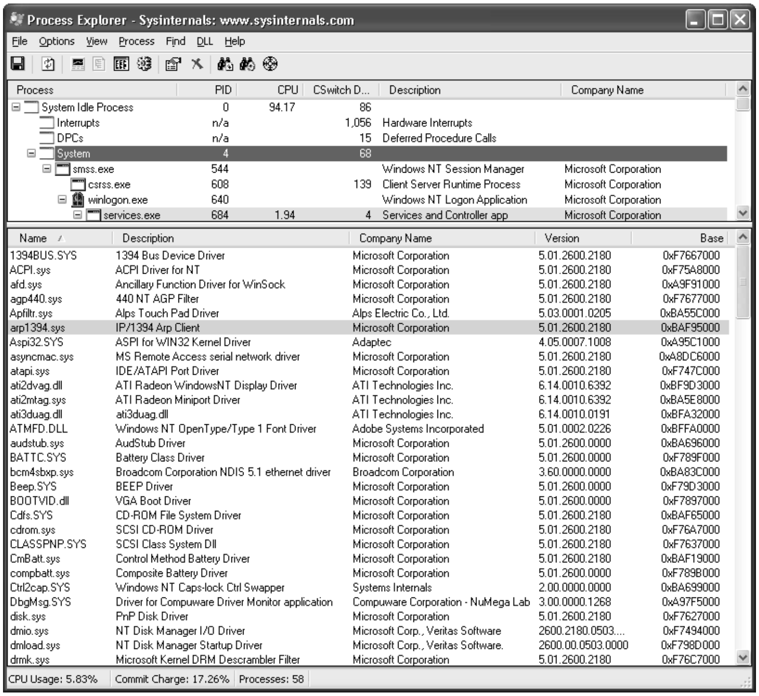
You can see a list of registered drivers on a Windows 2000 system by going to the Drivers section of the Computer Management Microsoft Management Console (MMC) snap-in or by right-clicking the My Computer icon on the desktop and selecting Manage from the context menu. (The Computer Management snap-in is in the Programs/Administrative Tools folder of the Start menu.) Navigate to the Drivers section within Computer Management by expanding System Tools, System Information, Software Environment and selecting Drivers, as shown here:



Name	Description	Type	State	Stat
abiosdsk	Abiosdsk	Kernel Driver	Stopped	OK
abp480n5	abp480n5	Kernel Driver	Stopped	OK
acpi	ACPI	Kernel Driver	Stopped	OK
acpiec	ACPIEC	Kernel Driver	Stopped	OK
adpu160m	adpu160m	Kernel Driver	Stopped	OK
afd	AFD Networking Support E...	Kernel Driver	Running	OK
aha154x	Aha154x	Kernel Driver	Stopped	OK
aic116x	aic116x	Kernel Driver	Stopped	OK
aic78u2	aic78u2	Kernel Driver	Stopped	OK
aic78xx	aic78xx	Kernel Driver	Stopped	OK
ami0nt	ami0nt	Kernel Driver	Stopped	OK
amsint	amsint	Kernel Driver	Stopped	OK
asc	asc	Kernel Driver	Stopped	OK
asc3350p	asc3350p	Kernel Driver	Stopped	OK
asc3550	asc3550	Kernel Driver	Stopped	OK
asyncmac	RAS Asynchronous Media ...	Kernel Driver	Stopped	OK
atapi	Standard IDE/ESDI Hard ...	Kernel Driver	Running	OK
atdisk	Atdisk	Kernel Driver	Stopped	OK
atmarpc	ATM ARP Client Protocol	Kernel Driver	Stopped	OK
audstub	Audio Stub Driver	Kernel Driver	Running	OK

In Windows XP and Windows Server 2003, you can obtain the identical information as reported by the Windows 2000 Computer Management MMC snap-in by executing the Msinfo32.exe utility from the Run dialog box of the Start menu. Select the System Drivers entry under Software Environment to see the list of drivers configured on the system. Those that are loaded have the text "Yes" in the Started column.

You can also view the list of loaded kernel-mode drivers with Process Explorer from *www.sysinternals.com*. Run Process Explorer, select the System process, and select DLLs from the Lower Pane menu entry in the View menu. Process Explorer lists the loaded drivers, their names, version information including company and description, and load address (assuming you have configured Process Explorer to display the corresponding columns):



Process	PID	CPU	CSwitch D...	Description	Company Name
⊟ System Idle Process	0	94.17	86		
Interrupts	n/a		1,056	Hardware Interrupts	
DPCs	n/a		15	Deferred Procedure Calls	
⊟ System	4		68		
⊟ smss.exe	544			Windows NT Session Manager	Microsoft Corporation
csrss.exe	608		139	Client Server Runtime Process	Microsoft Corporation
⊟ winlogon.exe	640			Windows NT Logon Application	Microsoft Corporation
⊟ services.exe	684	1.94	4	Services and Controller app	Microsoft Corporation

Name	Description	Company Name	Version	Base
1394BUS.SYS	1394 Bus Device Driver	Microsoft Corporation	5.01.2600.2180	0xF7667000
ACPI.sys	ACPI Driver for NT	Microsoft Corporation	5.01.2600.2180	0xF75A8000
afd.sys	Ancillary Function Driver for WinSock	Microsoft Corporation	5.01.2600.2180	0xA9F91000
agp440.sys	440 NT AGP Filter	Microsoft Corporation	5.01.2600.2180	0xF7677000
Apfiltr.sys	Alps Touch Pad Driver	Alps Electric Co., Ltd.	5.03.0001.0205	0xBA55C000
arp1394.sys	IP/1394 Arp Client	Microsoft Corporation	5.01.2600.2180	0xBAF95000
Aspi32.SYS	ASPI for WIN32 Kernel Driver	Adaptec	4.05.0007.1008	0xA95C1000
asyncmac.sys	MS Remote Access serial network driver	Microsoft Corporation	5.01.2600.2180	0xA8DC6000
atapi.sys	IDE/ATAPI Port Driver	Microsoft Corporation	5.01.2600.2180	0xF747C000
ati2dvag.dll	ATI Radeon WindowsNT Display Driver	ATI Technologies Inc.	6.14.0010.6392	0xBF9D3000
ati2mtag.sys	ATI Radeon Miniport Driver	ATI Technologies Inc.	6.14.0010.6392	0xBA5E8000
ati3duag.dll	ati3duag.dll	ATI Technologies Inc.	6.14.0010.0191	0xBFA32000
ATMFD.DLL	Windows NT OpenType/Type 1 Font Driver	Adobe Systems Incorporated	5.01.0002.0226	0xBFFA0000
audstub.sys	AudStub Driver	Microsoft Corporation	5.01.2600.0000	0xBA696000
BATTC.SYS	Battery Class Driver	Microsoft Corporation	5.01.2600.0000	0xF789F000
bcm4sbxp.sys	Broadcom Corporation NDIS 5.1 ethernet driver	Broadcom Corporation	3.60.0000.0000	0xBA83C000
Beep.SYS	BEEP Driver	Microsoft Corporation	5.01.2600.0000	0xF79D3000
BOOTVID.dll	VGA Boot Driver	Microsoft Corporation	5.01.2600.0000	0xF7897000
Cdfs.SYS	CD-ROM File System Driver	Microsoft Corporation	5.01.2600.2180	0xBAF65000
cdrom.sys	SCSI CD-ROM Driver	Microsoft Corporation	5.01.2600.2180	0xF76A7000
CLASSPNP.SYS	SCSI Class System Dll	Microsoft Corporation	5.01.2600.2180	0xF7637000
CmBatt.sys	Control Method Battery Driver	Microsoft Corporation	5.01.2600.2180	0xBAF19000
compbatt.sys	Composite Battery Driver	Microsoft Corporation	5.01.2600.0000	0xF789B000
Ctrl2cap.SYS	Windows NT Caps-lock Ctrl Swapper	Systems Internals	2.00.0000.0000	0xBA699000
DbgMsg.SYS	Driver for Compuware Driver Monitor application	Compuware Corporation - NuMega Lab	3.00.0000.1268	0xA97F5000
disk.sys	PnP Disk Driver	Microsoft Corporation	5.01.2600.2180	0xF7627000
dmio.sys	NT Disk Manager I/O Driver	Microsoft Corp., Veritas Software	2600.2180.0503....	0xF7494000
dmload.sys	NT Disk Manager Startup Driver	Microsoft Corp., Veritas Software.	2600.00.0503.0000	0xF798D000
drmk.sys	Microsoft Kernel DRM Descrambler Filter	Microsoft Corporation	5.01.2600.2180	0xF76C7000

Finally, if you're looking at a crash dump (or live system) with the kernel debugger, you can get a similar display with the kernel debugger *lm kv* command:

```
kd> lm kv
start    end        module name
804d4000 806aa280   nt          (pdb symbols)          c:\Symbols\ntoskrnl.pdb\
FB1EDACE71FB4812A5D5132819D72E523\ntoskrnl.pdb
    Loaded symbol image file: ntoskrnl.exe
    Image path: ntoskrnl.exe
    Timestamp: Thu Apr 24 10:57:43 2003 (3EA80977)  Checksum: 001E311B
    ImageSize : 001D6280
    File version:     5.1.2600.1151
    Product version:  5.1.2600.1151
    File flags:       0 (Mask 3F)
    File OS:          40004 NT Windows
    File type:        1.0 App
    File date:        00000000.00000000
    Translations:     0409.04b0
    CompanyName:      Microsoft Corporation
    ProductName:      Microsoft" Windows« Operating System
    InternalName:     ntoskrnl.exe
    OriginalFilename: ntoskrnl.exe
    ProductVersion:   5.1.2600.1151
    FileVersion:      5.1.2600.1151 (xpsp2.030422-1633)
    FileDescription:  NT Kernel & System
    LegalCopyright:    Microsoft Corporation. All rights reserved.
```

```
806ab000 806bde80   hal          (deferred)
    Image path: halacpi.dll
    Timestamp: Thu Aug 29 03:05:02 2002 (3D6DD5AE)  Checksum: 000203BD
    ImageSize : 00012E80
    Translations:     0000.04b0 0000.04e0 0409.04b0 0409.04e0
a8b8e000 a8bb4e80   kmixer       (deferred)
    Image path: \SystemRoot\system32\drivers\kmixer.sys
    Timestamp: Thu Aug 29 03:32:28 2002 (3D6DDC1C)  Checksum: 00032574
    ImageSize : 00026E80
    Translations:     0000.04b0 0000.04e0 0409.04b0 0409.04e0
```

Structure of a Driver

The I/O system drives the execution of device drivers. Device drivers consist of a set of routines that are called to process the various stages of an I/O request. Figure 9-5 illustrates the key driver-function routines.

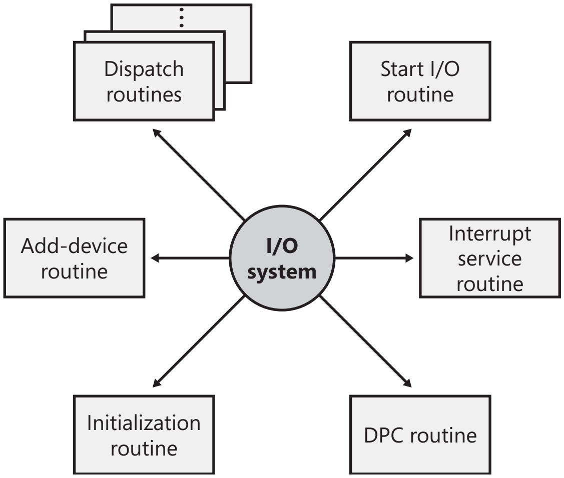


Figure 9-5 Primary device driver routines

- **An initialization routine** The I/O manager executes a driver's initialization routine, which is typically named *DriverEntry*, when it loads the driver into the operating system. The routine fills in system data structures to register the rest of the driver's routines with the I/O manager and performs any global driver initialization that's necessary.
- **An add-device routine** A driver that supports Plug and Play implements an add-device routine. The PnP manager sends a driver notification via this routine whenever a device for which the driver is responsible is detected. In this routine, a driver typically allocates a device object (described later in this chapter) to represent the device.

- **A set of dispatch routines** Dispatch routines are the main functions that a device driver provides. Some examples are open, close, read, and write and any other capabilities the device, file system, or network supports. When called on to perform an I/O operation, the I/O manager generates an IRP and calls a driver through one of the driver's dispatch routines.
- **A start I/O routine** The driver can use a start I/O routine to initiate a data transfer to or from a device. This routine is defined only in drivers that rely on the I/O manager to queue its incoming I/O requests. The I/O manager serializes IRPs for a driver by ensuring that the driver processes only one IRP at a time. Most drivers process multiple IRPs concurrently, but serialization makes sense for some drivers, such as a keyboard driver.
- **An interrupt service routine (ISR)** When a device interrupts, the kernel's interrupt dispatcher transfers control to this routine. In the Windows I/O model, ISRs run at device interrupt request level (DIRQL), so they perform as little work as possible to avoid blocking lower-level interrupts unnecessarily. (See Chapter 3 for more information on IRQLs.) An ISR queues a deferred procedure call (DPC), which runs at a lower IRQL (DPC/dispatch level), to execute the remainder of interrupt processing. (Only drivers for interrupt-driven devices have ISRs; a file system driver, for example, doesn't have one.)
- **An interrupt-servicing DPC routine** A DPC routine performs most of the work involved in handling a device interrupt after the ISR executes. The DPC routine executes at a lower IRQL (DPC/dispatch level) than that of the ISR, which runs at device level, to avoid blocking other interrupts unnecessarily. A DPC routine initiates I/O completion and starts the next queued I/O operation on a device.

Although the following routines aren't shown in Figure 9-5, they're found in many types of device drivers:

- **One or more I/O completion routines** A layered driver might have I/O completion routines that will notify it when a lower-level driver finishes processing an IRP. For example, the I/O manager calls a file system driver's I/O completion routine after a device driver finishes transferring data to or from a file. The completion routine notifies the file system driver about the operation's success, failure, or cancellation, and it allows the file system driver to perform cleanup operations.
- **A cancel I/O routine** If an I/O operation can be canceled, a driver can define one or more cancel I/O routines. When the driver receives an IRP for an I/O request that can be canceled, it assigns a cancel routine to the IRP. If a thread that issues an I/O request exits before the request is completed or cancels the operation (with the *CancelIo* Windows function, for example), the I/O manager executes the IRP's cancel routine if one is assigned to it. A cancel routine is responsible for performing whatever steps are necessary to release any resources acquired during the processing that has already taken place for the IRP as well as completing the IRP with a canceled status.

- **An unload routine** An unload routine releases any system resources a driver is using so that the I/O manager can remove them from memory. Any resources acquired in the initialization routine are usually released in the unload routine. A driver can be loaded and unloaded while the system is running.
- **A system shutdown notification routine** This routine allows driver cleanup on system shutdown.
- **Error-logging routines** When unexpected errors occur (for example, when a disk block goes bad), a driver's error-logging routines note the occurrence and notify the I/O manager. The I/O manager writes this information to an error log file.

> **Note** Most device drivers are written in C. Use of assembly language is highly discouraged because of the complexity it introduces and its effect of making a driver difficult to port between hardware architectures such as the x86, x64, and IA64.

Driver Objects and Device Objects

When a thread opens a handle to a file object (described in the "I/O Processing" section later in this chapter), the I/O manager must determine from the file object's name which driver (or drivers) it should call to process the request. Furthermore, the I/O manager must be able to locate this information the next time a thread uses the same file handle. The following system objects fill this need:

- A *driver object* represents an individual driver in the system. The I/O manager obtains the address of each of the driver's dispatch routines (entry points) from the driver object.
- A *device object* represents a physical or logical device on the system and describes its characteristics, such as the alignment it requires for buffers and the location of its device queue to hold incoming IRPs.

The I/O manager creates a driver object when a driver is loaded into the system, and it then calls the driver's initialization routine (for example, *DriverEntry*), which fills in the object attributes with the driver's entry points.

After loading, a driver can create device objects to represent devices, or even an interface to the driver, at any time by calling *IoCreateDevice* or *IoCreateDeviceSecure*. However, most Plug and Play drivers create devices with their add-device routine when the PnP manager informs them of the presence of a device for them to manage. Non–Plug and Play drivers, on the other hand, usually create device objects when the I/O manager invokes their initialization routine. The I/O manager unloads a driver when its last device object has been deleted and no references to the device remain.

When a driver creates a device object, the driver can optionally assign the device a name. A name places the device object in the object manager namespace, and a driver can either explicitly define a name or let the I/O manager autogenerate one. (The object manager namespace is described in Chapter 3.) By convention, device objects are placed in the \Device directory in the namespace, which is inaccessible by applications using the Windows API.

> **Note** Some drivers place device objects in directories other than \Device. For example, the Windows Logical Disk Manager volume manager creates device objects that represent disk partitions in the \Device\HarddiskDmVolumes directory. See Chapter 10 for a description of storage architecture, including the way storage drivers use device objects.

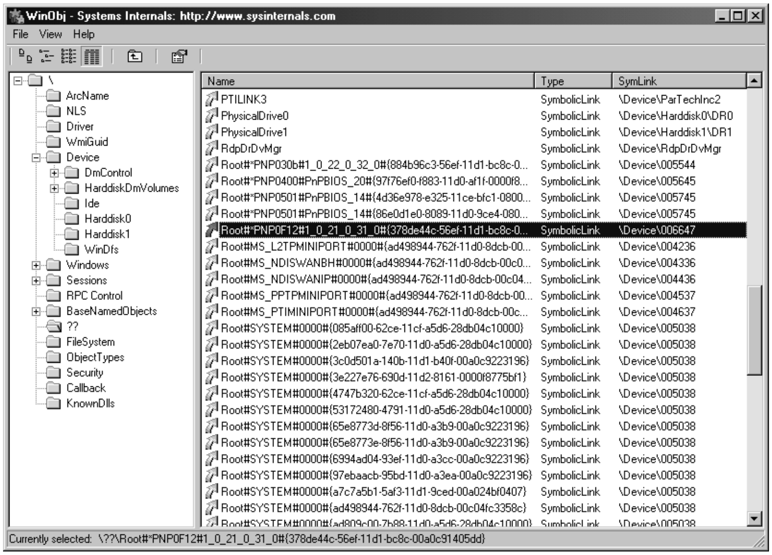
If a driver needs to make it possible for applications to open the device object, it must create a symbolic link in the \Global?? directory (or \?? on Windows 2000) to the device object's name in the \Device directory. (See Chapter 3 for more information on \??.) Non–Plug and Play and file system drivers typically create a symbolic link with a well-known name (for example, \Device\Hardware2). Because well-known names don't work well in an environment in which hardware appears and disappears dynamically, PnP drivers expose one or more interfaces by calling the *IoRegisterDeviceInterface* function, specifying a GUID (globally unique identifier) that represents the type of functionality exposed. GUIDs are 128-bit values that you can generate by using a tool included with the DDK and the Platform SDK, called Guidgen. Given the range of values that 128 bits represents, it's statistically almost certain that each GUID that Guidgen creates will be forever and globally unique.

IoRegisterDeviceInterface determines the symbolic link that is associated with a device instance; however, a driver must call *IoSetDeviceInterfaceState* to enable the interface to the device before the I/O manager actually creates the link. Drivers usually do this when the PnP manager starts the device by sending the driver a *start-device* command.

An application wanting to open a device object represented with a GUID can call Plug and Play setup functions in user space, such as *SetupDiEnumDeviceInterfaces*, to enumerate the interfaces present for a particular GUID and to obtain the names of the symbolic links it can use to open the device objects. For each device reported by *SetupDiEnumDeviceInterfaces*, an application executes *SetupDiGetDeviceInterfaceDetail* to obtain additional information about the device, such as its autogenerated name. After obtaining a device's name from *SetupDiGetDeviceInterfaceDetail*, the application can execute the Windows function *CreateFile* to open the device and obtain a handle.

EXPERIMENT: Looking at the \Device Directory

You can use the Winobj tool from *www.sysinternals.com* or the *!object* kernel debugger command to view the device names under \Device in the object manager namespace. The following screen shot shows an I/O manager–assigned symbolic link that points to a device object in \Device with an autogenerated name:



Name	Type	SymLink
PTILINK3	SymbolicLink	\Device\ParTechInc2
PhysicalDrive0	SymbolicLink	\Device\Harddisk0\DR0
PhysicalDrive1	SymbolicLink	\Device\Harddisk1\DR1
RdpDrDvMgr	SymbolicLink	\Device\RdpDrDvMgr
Root#*PNP030b#1_0_22_0_32_0#{884b96c3-56ef-11d1-bc8c-0...	SymbolicLink	\Device\005544
Root#*PNP0400#PnPBIOS_20#{97f76ef0-f883-11d0-af1f-0000f8...	SymbolicLink	\Device\005645
Root#*PNP0501#PnPBIOS_14#{4d36e978-e325-11ce-bfc1-0800...	SymbolicLink	\Device\005745
Root#*PNP0501#PnPBIOS_14#{86e0d1e0-8089-11d0-9ce4-080...	SymbolicLink	\Device\005745
Root#*PNP0F12#1_0_21_0_31_0#{378de44c-56ef-11d1-bc8c-0...	SymbolicLink	\Device\006647
Root#MS_L2TPMINIPORT#0000#{ad498944-762f-11d0-8dcb-00...	SymbolicLink	\Device\004236
Root#MS_NDISWANBH#0000#{ad498944-762f-11d0-8dcb-00c0...	SymbolicLink	\Device\004336
Root#MS_NDISWANIP#0000#{ad498944-762f-11d0-8dcb-00c04...	SymbolicLink	\Device\004436
Root#MS_PPTPMINIPORT#0000#{ad498944-762f-11d0-8dcb-00...	SymbolicLink	\Device\004537
Root#MS_PTIMINIPORT#0000#{ad498944-762f-11d0-8dcb-00c...	SymbolicLink	\Device\004637
Root#SYSTEM#0000#{085aff00-62ce-11cf-a5d6-28db04c10000}	SymbolicLink	\Device\005038
Root#SYSTEM#0000#{2eb07ea0-7e70-11d0-a5d6-28db04c10000}	SymbolicLink	\Device\005038
Root#SYSTEM#0000#{3c0d501a-140b-11d1-b40f-00a0c9223196}	SymbolicLink	\Device\005038
Root#SYSTEM#0000#{3e227e76-690d-11d2-8161-0000f8775bf1}	SymbolicLink	\Device\005038
Root#SYSTEM#0000#{4747b320-62ce-11cf-a5d6-28db04c10000}	SymbolicLink	\Device\005038
Root#SYSTEM#0000#{53172480-4791-11d0-a5d6-28db04c10000}	SymbolicLink	\Device\005038
Root#SYSTEM#0000#{65e8773d-8f56-11d0-a3b9-00a0c9223196}	SymbolicLink	\Device\005038
Root#SYSTEM#0000#{65e8773e-8f56-11d0-a3b9-00a0c9223196}	SymbolicLink	\Device\005038
Root#SYSTEM#0000#{6994ad04-93ef-11d0-a3cc-00a0c9223196}	SymbolicLink	\Device\005038
Root#SYSTEM#0000#{97ebaacb-95bd-11d0-a3ea-00a0c9223196}	SymbolicLink	\Device\005038
Root#SYSTEM#0000#{a7c7a5b1-5af3-11d1-9ced-00a024bf0407}	SymbolicLink	\Device\005038
Root#SYSTEM#0000#{ad498944-762f-11d0-8dcb-00c04fc3358c}	SymbolicLink	\Device\005038
Root#SYSTEM#0000#{ad809c00-7b88-11d0-a5d6-28db04c10000}	SymbolicLink	\Device\005038

Currently selected: \??\Root#*PNP0F12#1_0_21_0_31_0#{378de44c-56ef-11d1-bc8c-00a0c91405dd}

When you run the *!object* kernel debugger command and specify the \Device directory, you should see output similar to the following:

```
kd> !object \device
Object: e100c4a0  Type: (8a4f3178) Directory
    ObjectHeader: e100c488
    HandleCount: 0  PointerCount: 301
    Directory Object: e10011e8  Name: Device
    65535 symbolic links snapped through this directory

    Hash Address  Type          Name
    ---- -------  ----          ----
     00  8a437398 Device        KsecDD
         8a4a56f0 Device        Ndis
         8a0ed5c0 Device        ProcExp
         8a1ddb40 Device        Beep
         8a336d38 Device        0000008e
         8a4ed730 Device        00000032
         8a4ee4f0 Device        00000025
         8a4b5030 Device        00000019
```

```
  01  8a2303e8 Device        Netbios
      8a258030 Device        0000008f
      8a4ed4f0 Device        00000033
      8a4ee2b0 Device        00000026
  02  8a1756d8 Device        KSENUM#00000001
      8a4ec730 Device        00000040
      8a20fd58 Device        Ip
      8a2ef660 Device        RDP_CONSOLE0
      8a4ed2b0 Device        00000034
      8a4b4030 Device        00000027
  03  8a4ec4f0 Device        00000041
      8a15bf18 Device        0000009e
      8a2947d8 Device        {EFF45047-C948-4D32-86B5-736480DDBB9C}
      8a1b38e0 Device        Fips
      8a0b6038 Device        Video0
      8a288b48 Device        RDP_CONSOLE1
      8a2501f8 Device        KeyboardClass0
      8a4b3030 Device        00000035
      8a4b4df0 Device        00000028
  04  8a181030 Device        NDProxy
      8a4ec2b0 Device        00000042
      8a00b038 Device        Video1
      8a189030 Device        KeyboardClass1
§
```

When you execute *!object* and specify an object manager directory object, the kernel debugger dumps the contents of the directory according to the way the object manager organizes it internally. For fast lookups, a directory stores objects in a hash table based on a hash of the object names, so the output shows the objects stored in each bucket of the directory's hash table.

As Figure 9-6 illustrates, a device object points back to its driver object, which is how the I/O manager knows which driver routine to call when it receives an I/O request. It uses the device object to find the driver object representing the driver that services the device. It then indexes into the driver object by using the function code supplied in the original request; each function code corresponds to a driver entry point. (The function codes shown in Figure 9-6 are described in the section "IRP Stack Locations" later in this chapter.)

A driver object often has multiple device objects associated with it. The list of device objects represents the physical and logical devices that the driver controls. For example, each partition of a hard disk has a separate device object that contains partition-specific information. However, the same hard disk driver is used to access all partitions. When a driver is unloaded from the system, the I/O manager uses the queue of device objects to determine which devices will be affected by the removal of the driver.

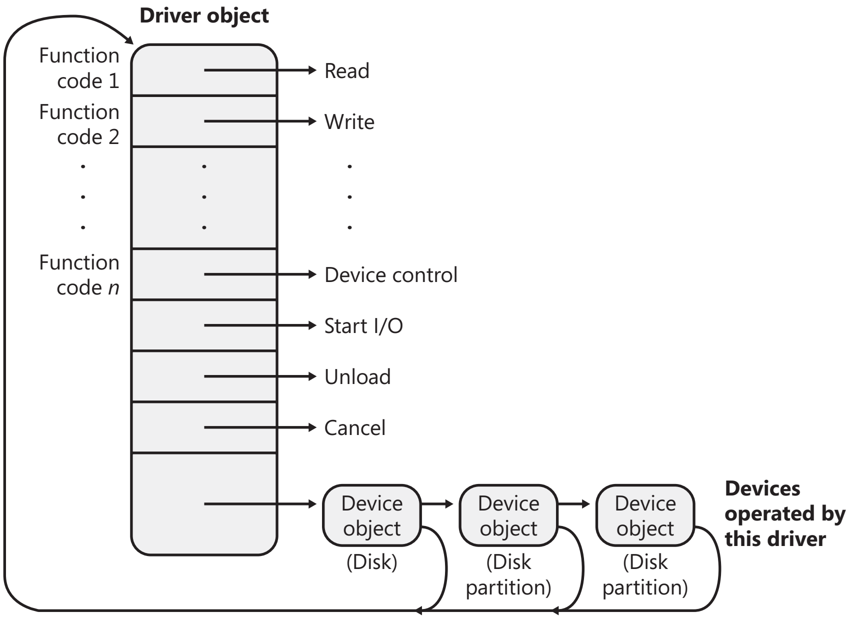


Figure 9-6 The driver object

EXPERIMENT: Displaying Driver and Device Objects

You can display driver and device objects with the kernel debugger *!drvobj* and *!devobj* commands, respectively. In the following example, the driver object for the keyboard class driver is examined, and its lone device object viewed:

```
kd> !drvobj kbdclass
Driver object (81869cb0) is for:
 \Driver\Kbdclass
Driver Extension List: (id , addr)

Device Object list: 81869310
kd> !devobj 81869310
Device object (81869310) is for:
 KeyboardClass0 \Driver\Kbdclass DriverObject 81869cb0
Current Irp a57a0e90 RefCount 0 Type 0000000b Flags 00002044
DevExt 818693c8 DevObjExt 818694b8
ExtensionFlags (0000000000)   AttachedDevice (Upper) 818691e0 \Driver\Ctrl2cap
AttachedTo (Lower) 81869500 \Driver\i8042prt
Device queue is busy -- Queue empty.
```

Notice that the *!devobj* command also shows you the addresses and names of any device objects that the object you're viewing is layered over (the *AttachedTo* line) as well as the device objects layered on top of the object specified (the *AttachedDevice* line).

Using objects to record information about drivers means that the I/O manager doesn't need to know details about individual drivers. The I/O manager merely follows a pointer to locate a driver, thereby providing a layer of portability and allowing new drivers to be loaded easily. Representing devices and drivers with different objects also makes it easy for the I/O system to assign drivers to control additional or different devices if the system configuration changes.

Opening Devices

File objects are the kernel-mode constructs for handles to files or devices. File objects clearly fit the criteria for objects in Windows: they are system resources that two or more user-mode processes can share, they can have names, they are protected by object-based security, and they support synchronization. Although most shared resources in Windows are memory-based resources, most of those that the I/O system manages are located on physical devices or represent actual physical devices. Despite this difference, shared resources in the I/O system, like those in other components of the Windows executive, are manipulated as objects. (See Chapter 3 for a description of the object manager and Chapter 8 for information on object security.)

File objects provide a memory-based representation of resources that conform to an I/O-centric interface, in which they can be read from or written to. Table 9-1 lists some of the file object's attributes. For specific field declarations and sizes, see the structure definition for FILE_OBJECT in Ntddk.h.

Table 9-1 File Object Attributes

Attribute	Purpose
Filename	Identifies the physical file that the file object refers to
Current byte offset	Identifies the current location in the file (valid only for synchronous I/O)
Share modes	Indicate whether other callers can open the file for read, write, or delete operations while the current caller is using it
Open mode flags	Indicate whether I/O will be synchronous or asynchronous, cached or noncached, sequential or random, and so on
Pointer to device object	Indicates the type of device the file resides on
Pointer to the volume parameter block (VPB)	Indicates the volume, or partition, that the file resides on
Pointer to section object pointers	Indicates a root structure that describes a mapped file
Pointer to private cache map	Identifies which parts of the file are cached by the cache manager and where they reside in the cache

EXPERIMENT: Viewing the File Object Data Structure

You can view the contents of the kernel-mode file object data structure with the kernel-debuggers dt command:

```
kd> dt nt!_file_object
nt!_FILE_OBJECT
   +0x000 Type             : Int2B
   +0x002 Size             : Int2B
   +0x004 DeviceObject     : Ptr32 _DEVICE_OBJECT
   +0x008 Vpb              : Ptr32 _VPB
   +0x00c FsContext        : Ptr32 Void
   +0x010 FsContext2       : Ptr32 Void
   +0x014 SectionObjectPointer : Ptr32 _SECTION_OBJECT_POINTERS
   +0x018 PrivateCacheMap  : Ptr32 Void
   +0x01c FinalStatus      : Int4B
   +0x020 RelatedFileObject : Ptr32 _FILE_OBJECT
   +0x024 LockOperation    : UChar
   +0x025 DeletePending    : UChar
   +0x026 ReadAccess       : UChar
   +0x027 WriteAccess      : UChar
   +0x028 DeleteAccess     : UChar
   +0x029 SharedRead       : UChar
   +0x02a SharedWrite      : UChar
   +0x02b SharedDelete     : UChar
   +0x02c Flags            : Uint4B
   +0x030 FileName         : _UNICODE_STRING
   +0x038 CurrentByteOffset : _LARGE_INTEGER
   +0x040 Waiters          : Uint4B
   +0x044 Busy             : Uint4B
   +0x048 LastLock         : Ptr32 Void
   +0x04c Lock             : _KEVENT
   +0x05c Event            : _KEVENT
   +0x06c CompletionContext : Ptr32 _IO_COMPLETION_CONTEXT
```

When a caller opens a file or a simple device, the I/O manager returns a handle to a file object. Figure 9-7 illustrates what occurs when a file is opened.

In this example, (1) a C program calls the run-time library function *fopen*, which in turn (2) calls the Windows *CreateFile* function. The Windows subsystem DLL (in this case, Kernel32.dll) then (3) calls the native *NtCreateFile* function in Ntdll.dll. The routine in Ntdll.dll contains the appropriate instruction to cause a transition into kernel mode to the system service dispatcher, which then (4) calls the real *NtCreateFile* routine in Ntoskrnl.exe. (See Chapter 3 for more information about system service dispatching.)

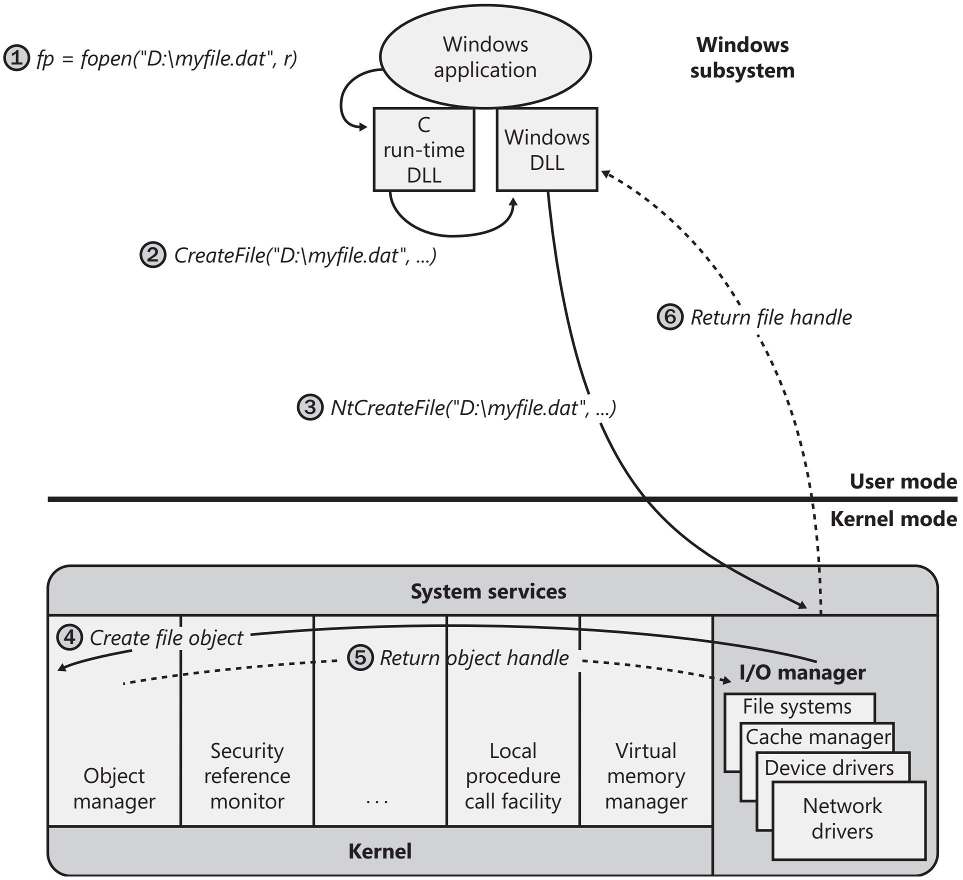


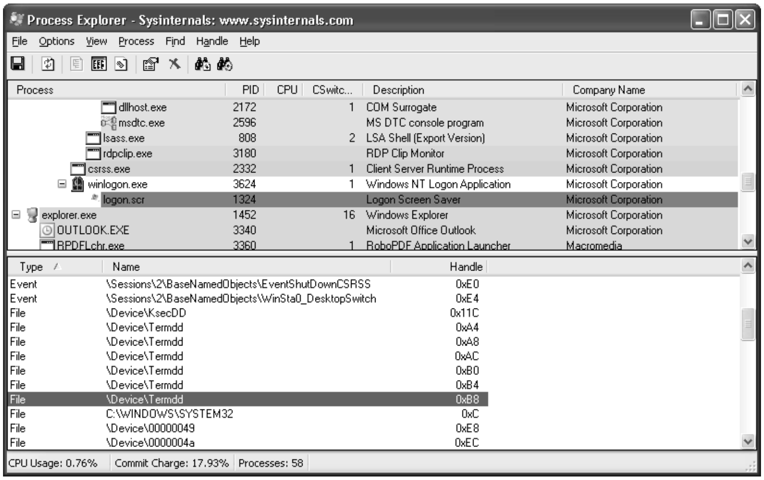
Figure 9-7 Opening a file object

> **Note** File objects represent open instances of files, not files themselves. Unlike UNIX systems, which use *vnodes*, Windows does not define the representation of a file; Windows system drivers define their own representations.

Like other executive objects, files are protected by a security descriptor that contains an access-control list (ACL). The I/O manager consults the security subsystem to determine whether the file's ACL allows the process to access the file in the way its thread is requesting. If it does, (5,6) the object manager grants the access and associates the granted access rights with the file handle that it returns. If this thread or another thread in the process needs to perform additional operations not specified in the original request, the thread must open another handle, which prompts another security check. (See Chapter 8 for more information about object protection.)

EXPERIMENT: Viewing Device Handles

Any process that has an open handle to a device will have a file object in its handle table corresponding to the open instance. You can view these handles with Process Explorer from *www.sysinternals.com* by selecting a process, checking Show Lower Pane in the View menu and Handles in the Lower Pane View submenu of the View menu. Sort by the Type column and scroll to where you see the handles that represent file objects, which are labeled as "File":


Process Explorer - Sysinternals: www.sysinternals.com
File Options View Process Find Handle Help

Process	PID	CPU	CSwitc...	Description	Company Name
dllhost.exe	2172		1	COM Surrogate	Microsoft Corporation
msdtc.exe	2596			MS DTC console program	Microsoft Corporation
lsass.exe	808		2	LSA Shell (Export Version)	Microsoft Corporation
rdpclip.exe	3180			RDP Clip Monitor	Microsoft Corporation
csrss.exe	2332		1	Client Server Runtime Process	Microsoft Corporation
winlogon.exe	3624		1	Windows NT Logon Application	Microsoft Corporation
logon.scr	1324			Logon Screen Saver	Microsoft Corporation
explorer.exe	1452		16	Windows Explorer	Microsoft Corporation
OUTLOOK.EXE	3340			Microsoft Office Outlook	Microsoft Corporation
RPDFLchr.exe	3360		1	RoboPDF Application Launcher	Macromedia

Type	Name	Handle
Event	\Sessions\2\BaseNamedObjects\EventShutDownCSRSS	0xE0
Event	\Sessions\2\BaseNamedObjects\WinSta0_DesktopSwitch	0xE4
File	\Device\KsecDD	0x11C
File	\Device\Termdd	0xA4
File	\Device\Termdd	0xA8
File	\Device\Termdd	0xAC
File	\Device\Termdd	0xB0
File	\Device\Termdd	0xB4
File	\Device\Termdd	0xB8
File	C:\WINDOWS\SYSTEM32	0xC
File	\Device\00000049	0xE8
File	\Device\0000004a	0xEC

CPU Usage: 0.76% Commit Charge: 17.93% Processes: 58

In this example the Csrss process has handles open to file objects that represent open instances of devices with autogenerated names as well as ones that belong to the Terminal Server Driver. You can look at the specific file object in the kernel debugger by first identifying the address of the object. The following command reports information on the highlighted handle (handle value 0xB8) in the preceding screen shot, which is in the Csrss.exe process that has a process ID of 2332 (0x91c):

```
0: kd> !handle b8 f 91c
processor number 0
Searching for Process with Cid == 91c
PROCESS 86a6c020  SessionId: 0  Cid: 091c    Peb: 7ffde000  ParentCid: 028c
    DirBase: 1158a000  ObjectTable: e1b5d080  HandleCount: 643.
    Image: csrss.exe

New version of handle table at e2b44000 with 643 Entries in use
00B8: Object: 866ae9e8  GrantedAccess: 0012019f
Object: 866ae9e8  Type: (86fe8ad0) File
    ObjectHeader: 866ae9d0
        HandleCount: 1  PointerCount: 3
```

Because the object is a file object, you can get information about it with the *!fileobj* command:

```
0: kd> !fileobj 866ae9e8

File object name:

Device Object: 0x86d0c8b8   \Driver\TermDD
Vpb is NULL

Flags:  0x40000
        Handle Created

FsContext: 0x866a31d8   FsContext2: 0x00000001
CurrentByteOffset: 0
```

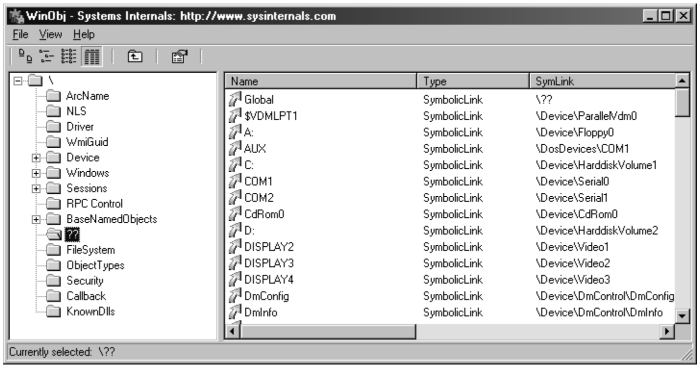
Because a file object is a memory-based representation of a shareable resource and not the resource itself, it's different from other executive objects. A file object contains only data that is unique to an object handle, whereas the file itself contains the data or text to be shared. Each time a thread opens a file handle, a new file object is created with a new set of handle-specific attributes. For example, the current byte offset attribute refers to the location in the file at which the next read or write operation using that handle will occur. Each handle to a file has a private byte offset even though the underlying file is shared. A file object is also unique to a process, except when a process duplicates a file handle to another process (by using the Windows *DuplicateHandle* function) or when a child process inherits a file handle from a parent process. In these situations, the two processes have separate handles that refer to the same file object.

Although a file handle might be unique to a process, the underlying physical resource is not. Therefore, as with any shared resource, threads must synchronize their access to shareable files, file directories, and devices. If a thread is writing to a file, for example, it should specify exclusive write access when opening the file handle to prevent other threads from writing to the file at the same time. Alternatively, by using the Windows *LockFile* function, the thread could lock a portion of the file while writing to it.

When a file is opened, the filename includes the name of the device object on which the file resides. For example, the name \Device\Floppy0\Myfile.dat refers to the file Myfile.dat on floppy disk drive A. The substring \Device\Floppy0 is the name of the internal Windows device object representing that floppy disk drive. When opening Myfile.dat, the I/O manager creates a file object and stores a pointer to the Floppy0 device object in the file object and then returns a file handle to the caller. Thereafter, when the caller uses the file handle, the I/O manager can find the Floppy0 device object directly. Keep in mind that internal Windows device names can't be used in Windows applications—instead, the device name must appear in a special directory in the object manager's namespace, which is \?? on Windows 2000 and \Global?? on Windows XP and Windows Server 2003. This directory contains symbolic links to the real, internal Windows device names. Device drivers are responsible for creating links in this directory so that their devices will be accessible to Windows applications. You can examine or even change these links programmatically with the Windows *QueryDosDevice* and *DefineDosDevice* functions.

EXPERIMENT: Viewing Windows Device Name to Windows Device Name Mappings

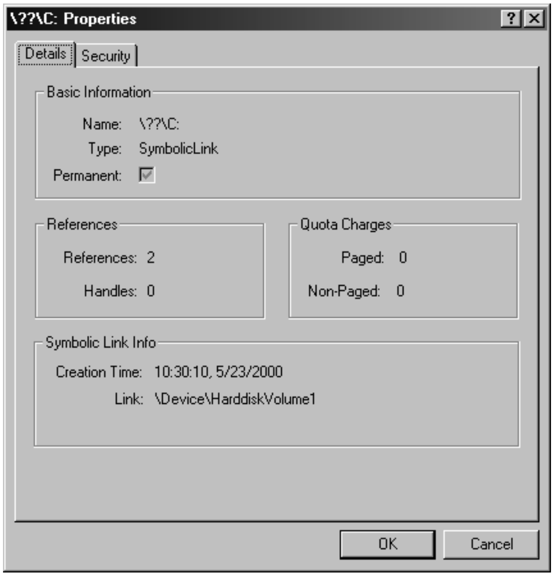
You can examine the symbolic links that define the Windows device namespace with the Winobj utility from *www.sysinternals.com*. Run Winobj, and click on the \?? Directory on Windows 2000 or \Global?? on Windows XP or Windows Server 2003, as shown here:



Name	Type	SymLink
Global	SymbolicLink	\??
$VDMLPT1	SymbolicLink	\Device\ParallelVdm0
A:	SymbolicLink	\Device\Floppy0
AUX	SymbolicLink	\DosDevices\COM1
C:	SymbolicLink	\Device\HarddiskVolume1
COM1	SymbolicLink	\Device\Serial0
COM2	SymbolicLink	\Device\Serial1
CdRom0	SymbolicLink	\Device\CdRom0
D:	SymbolicLink	\Device\HarddiskVolume2
DISPLAY2	SymbolicLink	\Device\Video1
DISPLAY3	SymbolicLink	\Device\Video2
DISPLAY4	SymbolicLink	\Device\Video3
DmConfig	SymbolicLink	\Device\DmControl\DmConfig
DmInfo	SymbolicLink	\Device\DmControl\DmInfo

Currently selected: \??

Notice the symbolic links on the right. Try double-clicking on the device C:. You should see something like this:



Basic Information

Name: \??\C:

Type: SymbolicLink

Permanent: ☑

References

References: 2

Handles: 0

Quota Charges

Paged: 0

Non-Paged: 0

Symbolic Link Info

Creation Time: 10:30:10, 5/23/2000

Link: \Device\HarddiskVolume1

C: is a symbolic link to the internal device named \Device\HarddiskVolume1, or the first volume on the first hard drive in the system. The COM1 entry in Winobj is a symbolic link to \Device\Serial0, and so forth. Try creating your own links with the *subst* command at a command prompt.

I/O Processing

Now that we've covered the structure and types of drivers and the data structures that support them, let's look at how I/O requests flow through the system. I/O requests pass through several predictable stages of processing. The stages vary depending on whether the request is destined for a device operated by a single-layered driver or for a device reached through a multilayered driver. Processing varies further depending on whether the caller specified synchronous or asynchronous I/O, so we'll begin our discussion of I/O types with these two then move on to others.

Types of I/O

Applications have several options for the I/O requests they issue. For example, they can specify synchronous or asynchronous I/O, I/O that maps a device's data into the application's address space for access via application virtual memory rather than I/O APIs, and I/O that transfers data between a device and noncontiguous application buffers in a single request. Furthermore, the I/O manager gives the drivers the choice of implementing a shortcut I/O interface that can often mitigate IRP allocation for I/O processing. In this section, we'll explain each of these I/O variations.

Synchronous I/O and Asynchronous I/O

Most I/O operations that applications issue are *synchronous*; that is, the application waits while the device performs the data transfer and returns a status code when the I/O is complete. The program can then continue and access the transferred data immediately. When used in their simplest form, the Windows *ReadFile* and *WriteFile* functions are executed synchronously. They complete an I/O operation before returning control to the caller.

Asynchronous I/O allows an application to issue an I/O request and then continue executing while the device transfers the data. This type of I/O can improve an application's throughput because it allows the application thread to continue with other work while an I/O operation is in progress. To use asynchronous I/O, you must specify the FILE_FLAG_OVERLAPPED flag when you call the Windows *CreateFile* function. Of course, after issuing an asynchronous I/O operation, the thread must be careful not to access any data from the I/O operation until the device driver has finished the data transfer. The thread must synchronize its execution with the completion of the I/O request by monitoring a handle of a synchronization object (whether that's an event object, an I/O completion port, or the file object itself) that will be signaled when the I/O is complete.

Regardless of the type of I/O request, internally, I/O operations issued to a driver on behalf of the application are performed asynchronously; that is, once an I/O request has been initiated, the device driver returns to the I/O system. Whether or not the I/O system returns immediately to the caller depends on whether the file was opened for synchronous or asynchronous

I/O. Figure 9-8 illustrates the flow of control when a read operation is initiated. Notice that if a wait is done, which depends on the overlapped flag in the file object, it is done in kernel mode by the *NtReadFile* function.

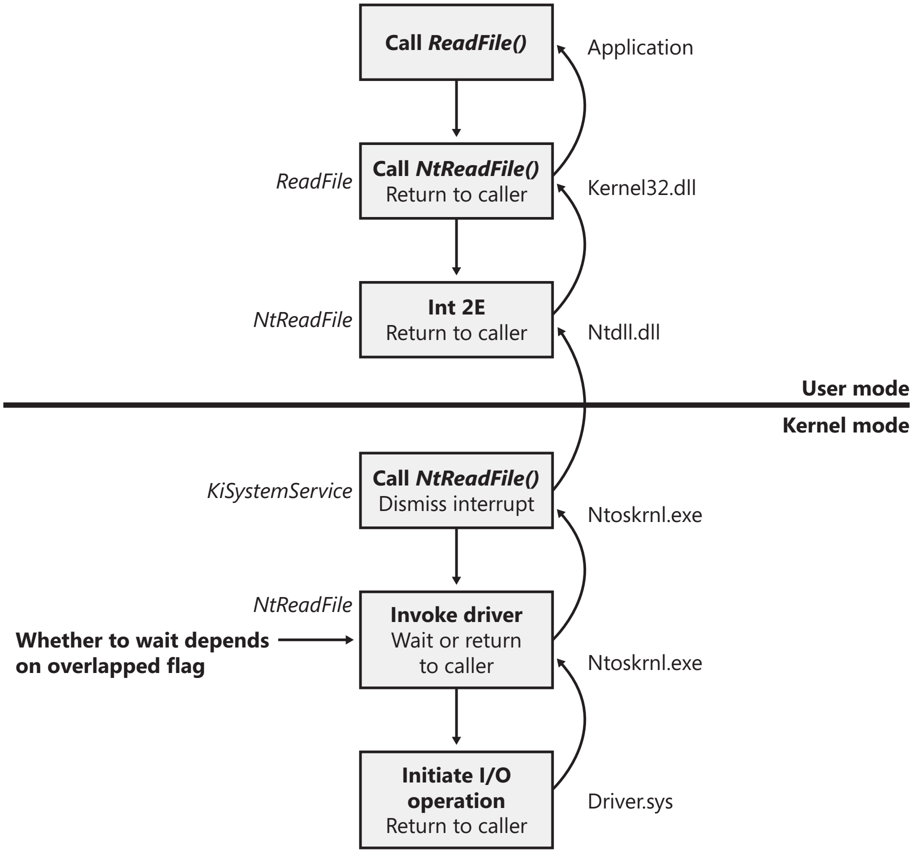


Figure 9-8 Control flow for an I/O operation

You can test the status of a pending asynchronous I/O with the Windows *HasOverlappedIoCompleted* function. If you're using I/O completion ports (described in the "I/O Completion Ports" section later in this chapter), you can use the *GetQueuedCompletionStatus* function.

Fast I/O

Fast I/O is a special mechanism that allows the I/O system to bypass generating an IRP and instead go directly to the file system driver or cache manager to complete an I/O request. (Fast I/O is described in detail in chapters 11 and 12.) A driver registers its fast I/O entry points by entering them in a structure pointed to by the PFAST_IO_DISPATCH pointer in its driver object.

EXPERIMENT: Looking at a Driver’s Registered Fast I/O Routines

The *!drvobj* kernel debugger command can list the fast I/O routines that a driver registers in its driver object. However, typically only file system drivers have any use for fast I/O routines. The following output shows the fast I/O table for the NTFS file system driver object:

```
kd> !drvobj \filesystem\ntfs 2
Driver object (8a4372a0) is for:
 \FileSystem\Ntfs
DriverEntry:   f7bd7398 Ntfs!DriverEntry
DriverStartIo: 00000000
DriverUnload:  00000000

Dispatch routines:
[00] IRP_MJ_CREATE                      f7b76390        Ntfs!NtfsFsdCreate


Fast I/O routines:
FastIoCheckIfPossible                   f7b74a0b        Ntfs!NtfsFastIoCheckIfPossible
FastIoRead                              f7b77bbc        Ntfs!NtfsCopyReadA
FastIoWrite                             f7b8a9cc        Ntfs!NtfsCopyWriteA
FastIoQueryBasicInfo                    f7b7cd5e        Ntfs!NtfsFastQueryBasicInfo
FastIoQueryStandardInfo                 f7b7779e        Ntfs!NtfsFastQueryStdInfo
FastIoLock                              f7b8b738        Ntfs!NtfsFastLock
FastIoUnlockSingle                      f7b8b66c        Ntfs!NtfsFastUnlockSingle
FastIoUnlockAll                         f7ba5cd6        Ntfs!NtfsFastUnlockAll
FastIoUnlockAllByKey                    f7bcdab2        Ntfs!NtfsFastUnlockAllByKey
AcquireFileForNtCreateSection           f7b77771        Ntfs!NtfsAcquireForCreate
Section
ReleaseFileForNtCreateSection           f7b77758        Ntfs!NtfsReleaseForCreate
Section
FastIoQueryNetworkOpenInfo              f7bbec06        Ntfs!NtfsFastQueryNetworkOpen
Info
AcquireForModWrite                      f7b8663d        Ntfs!NtfsAcquireFileFor
ModWrite
MdlRead                                 f7bbed20        Ntfs!NtfsMdlReadA
```

The output shows that NTFS has registered its *NtfsFastIoCheckIfPossible* routine as the fast I/O table’s *FastIoCheckIfPossible* entry. As the name of this fast I/O entry implies, the I/O manager sometimes calls this function before issuing a fast I/O request, giving a driver an opportunity to indicate when fast I/O operations on a file are not feasible.

Mapped File I/O and File Caching

Mapped file I/O is an important feature of the I/O system, one that the I/O system and the memory manager produce jointly. (See Chapter 7 for details on how mapped files are implemented.) *Mapped file I/O* refers to the ability to view a file residing on disk as part of a process's virtual memory. A program can access the file as a large array without buffering data or performing disk I/O. The program accesses memory, and the memory manager uses its paging mechanism to load the correct page from the disk file. If the application writes to its virtual address space, the memory manager writes the changes back to the file as part of normal paging.

Mapped file I/O is available in user mode through the Windows *CreateFileMapping* and *MapViewOfFile* functions. Within the operating system, mapped file I/O is used for important operations such as file caching and image activation (loading and running executable programs). The other major consumer of mapped file I/O is the cache manager. File systems use the cache manager to map file data in virtual memory to provide better response time for I/O-bound programs. As the caller uses the file, the memory manager brings accessed pages into memory. Whereas most caching systems allocate a fixed number of bytes for caching files in memory, the Windows cache grows or shrinks depending on how much memory is available. This size variability is possible because the cache manager relies on the memory manager to automatically expand (or shrink) the size of the cache, using the normal working set mechanisms explained in Chapter 7. By taking advantage of the memory manager's paging system, the cache manager avoids duplicating the work that the memory manager already performs. (The workings of the cache manager are explained in detail in Chapter 11.)

Scatter/Gather I/O

Windows also supports a special kind of high-performance I/O that is called *scatter/gather*, available via the Windows *ReadFileScatter* and *WriteFileGather* functions. These functions allow an application to issue a single read or write from more than one buffer in virtual memory to a contiguous area of a file on disk instead of issuing a separate I/O request for each buffer. To use scatter/gather I/O, the file must be opened for noncached I/O, the user buffers being used have to be page-aligned, and the I/Os must be asynchronous (overlapped). Furthermore, if the I/O is directed at a mass storage device, the I/O must be aligned on a device sector boundary and have a length that is a multiple of the sector size.

I/O Request Packets

The I/O request packet (IRP) is where the I/O system stores information it needs to process an I/O request. When a thread calls an I/O service, the I/O manager constructs an IRP to represent the operation as it progresses through the I/O system. If possible, the I/O manager allocates IRPs from one of two per-processor IRP nonpaged look-aside lists: the small-IRP look-aside list stores IRPs with one stack location (IRP stack locations are described shortly), and the large-IRP look-aside list contains IRPs with multiple stack locations. By default, the system stores IRPs with eight stack locations on the large-IRP look-aside list, but once per minute the system adjusts the number of stack locations allocated based on how many stack locations have been required. If an IRP requires more stack locations than are contained in the IRPs on the large-IRP look-aside list, the I/O manager allocates IRPs from nonpaged pool. After allocating and initializing an IRP, the I/O manager stores a pointer to the caller's file object in the IRP.

Note If defined, the DWORD registry value HKLM\System\CurrentControlSet\Session Manager\I/O System\LargIrpStackLocations specifies how many stack locations are contained in IRPs stored on the large-IRP look-aside list.

Figure 9-9 shows a sample I/O request that demonstrates the relationship between an IRP and the file, device, and driver objects described in the preceding sections. Although this example shows an I/O request to a single-layered device driver, most I/O operations aren't this direct; they involve one or more layered drivers. (This case will be shown later in this section.)

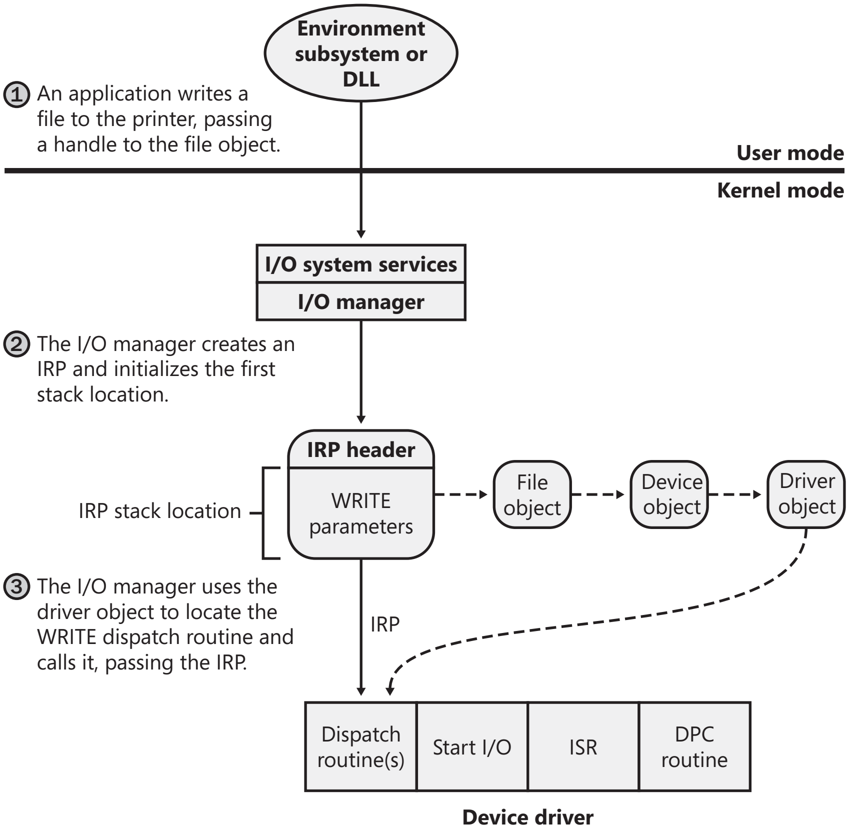

Environment subsystem or DLL
1 An application writes a file to the printer, passing a handle to the file object.
User mode
Kernel mode
I/O system services
I/O manager
2 The I/O manager creates an IRP and initializes the first stack location.
IRP header
IRP stack location
WRITE parameters
File object
Device object
Driver object
3 The I/O manager uses the driver object to locate the WRITE dispatch routine and calls it, passing the IRP.
IRP
Dispatch routine(s)
Start I/O
ISR
DPC routine
Device driver

Figure 9-9 Data structures involved in a single-layered driver I/O request

IRP Stack Locations

An IRP consists of two parts: a fixed header (often referred to as the IRP's *body*) and one or more stack locations. The fixed portion contains information such as the type and size of the request, whether the request is synchronous or asynchronous, a pointer to a buffer for buffered I/O, and state information that changes as the request progresses. An IRP stack location contains a function code (consisting of a major code and a minor code), function-specific parameters, and a pointer to the caller's file object. The *major function code* identifies which of a driver's dispatch routines the I/O manager invokes when passing an IRP to a driver. An optional *minor function code* sometimes serves as a modifier of the major function code. Power and Plug and Play commands always have minor function codes.

Most drivers specify dispatch routines to handle only a subset of possible major function codes, including create (open), read, write, device I/O control, power, Plug and Play, System (for WMI commands), and close. (See the following experiment for a complete listing of major function codes.) File system drivers are an example of a driver type that often fills in most or all of its dispatch entry points with functions. The I/O manager sets any dispatch entry points that a driver doesn't fill to point to its own *IopInvalidDeviceRequest*, which returns an error code to the caller indicating that the function specified for the device is invalid.

EXPERIMENT: Looking at Driver Dispatch Routines

You can obtain a listing of the functions a driver has defined for its dispatch routines by entering a **7** after the driver object's name (or address) in the *!drvobj* kernel debugger command. The following output shows that drivers support 28 IRP types.

```
kd> !drvobj kbdclass 7
Driver object (8a238900) is for:
 \Driver\Kbdclass
Driver Extension List: (id , addr)

Device Object list:
8a189030  8a2501f8

DriverEntry:   f7822d22 kbdclass!DriverEntry
DriverStartIo: 00000000
DriverUnload:  00000000

Dispatch routines:
[00] IRP_MJ_CREATE                      f781fd3b        kbdclass!KeyboardClassCreate
[01] IRP_MJ_CREATE_NAMED_PIPE           804eef8e        nt!IopInvalidDeviceRequest
[02] IRP_MJ_CLOSE                       f781ff4c        kbdclass!KeyboardClassClose
[03] IRP_MJ_READ                        f7820ba5        kbdclass!KeyboardClassRead
[04] IRP_MJ_WRITE                       804eef8e        nt!IopInvalidDeviceRequest
[05] IRP_MJ_QUERY_INFORMATION           804eef8e        nt!IopInvalidDeviceRequest
[06] IRP_MJ_SET_INFORMATION             804eef8e        nt!IopInvalidDeviceRequest
[07] IRP_MJ_QUERY_EA                    804eef8e        nt!IopInvalidDeviceRequest
[08] IRP_MJ_SET_EA                      804eef8e        nt!IopInvalidDeviceRequest
[09] IRP_MJ_FLUSH_BUFFERS               f781fcbe        kbdclass!KeyboardClassFlush
[0a] IRP_MJ_QUERY_VOLUME_INFORMATION    804eef8e        nt!IopInvalidDeviceRequest
```

```
[0b] IRP_MJ_SET_VOLUME_INFORMATION      804eef8e        nt!IopInvalidDeviceRequest
[0c] IRP_MJ_DIRECTORY_CONTROL           804eef8e        nt!IopInvalidDeviceRequest
[0d] IRP_MJ_FILE_SYSTEM_CONTROL         804eef8e        nt!IopInvalidDeviceRequest
[0e] IRP_MJ_DEVICE_CONTROL              f7821829        kbdclass!KeyboardClassDevice
                                                        Control
[0f] IRP_MJ_INTERNAL_DEVICE_CONTROL     f7821200        kbdclass!KeyboardClassPass
                                                        Through
[10] IRP_MJ_SHUTDOWN                    804eef8e        nt!IopInvalidDeviceRequest
[11] IRP_MJ_LOCK_CONTROL                804eef8e        nt!IopInvalidDeviceRequest
[12] IRP_MJ_CLEANUP                     f781fc84        kbdclass!KeyboardClassCleanup
[13] IRP_MJ_CREATE_MAILSLOT             804eef8e        nt!IopInvalidDeviceRequest
[14] IRP_MJ_QUERY_SECURITY              804eef8e        nt!IopInvalidDeviceRequest
[15] IRP_MJ_SET_SECURITY                804eef8e        nt!IopInvalidDeviceRequest
[16] IRP_MJ_POWER                       f7821f51        kbdclass!KeyboardClassPower
[17] IRP_MJ_SYSTEM_CONTROL              f7821649        kbdclass!KeyboardClassSystem
                                                        Control
[18] IRP_MJ_DEVICE_CHANGE               804eef8e        nt!IopInvalidDeviceRequest
[19] IRP_MJ_QUERY_QUOTA                 804eef8e        nt!IopInvalidDeviceRequest
[1a] IRP_MJ_SET_QUOTA                   804eef8e        nt!IopInvalidDeviceRequest
[1b] IRP_MJ_PNP                         f78206c1        kbdclass!KeyboardPnP
```

While active, each IRP is usually stored in an IRP list associated with the thread that requested the I/O. This arrangement allows the I/O system to find and cancel any outstanding IRPs if a thread terminates or is terminated with outstanding I/O requests.

EXPERIMENT: Looking at a Thread's Outstanding IRPs

When you use the *!thread* command, it prints any IRPs associated with the thread. Run the kernel debugger with live debugging, and locate the Service Control Manager process (Services.exe) in the output generated by the *!process* command:

```
lkd> !process 0 0
**** NT ACTIVE PROCESS DUMP ****
…
PROCESS 8a238da8  SessionId: 0  Cid: 02a8    Peb: 7ffdf000  ParentCid: 027c
    DirBase: 14fac000  ObjectTable: e1c3e008  HandleCount: 365.
    Image: SERVICES.EXE
…
```

Then dump the threads for the process by executing the *!process* command on the process object. You should see many threads, with most of them having IRPs reported in the IRP List area of the thread information (note that the debugger will show only the first 17 IRPs for a thread that has more than 17 outstanding I/O requests):

```
kd> !process 8a238da8
PROCESS 8a238da8  SessionId: 0  Cid: 02a8    Peb: 7ffdf000  ParentCid: 027c
    DirBase: 14fac000  ObjectTable: e1c3e008  HandleCount: 365.
    Image: SERVICES.EXE
    VadRoot 8a1be328 Vads 88 Clone 0 Private 346. Modified 37. Locked 0.
    DeviceMap e10087c0
…
```

```
     THREAD 8a124870  Cid 02a8.0338  Teb: 7ffd8000 Win32Thread: 00000000 WAIT:
(WrQueue) UserMode Non-Alertable
            8a2dc620  Unknown
            8a124960  NotificationTimer
        IRP List:            8a2c2c00: (0006,0094) Flags: 00000900  Mdl: 00000000
            8a20f770: (0006,0094) Flags: 00000900  Mdl: 00000000
            8a437780: (0006,0094) Flags: 00000900  Mdl: 00000000
            89b1de68: (0006,0094) Flags: 00000900  Mdl: 00000000
            8a0e6058: (0006,0094) Flags: 00000900  Mdl: 00000000
            8a0f1550: (0006,0094) Flags: 00000900  Mdl: 00000000
            8a3b3c18: (0006,0094) Flags: 00000900  Mdl: 00000000
            8a429190: (0006,0094) Flags: 00000900  Mdl: 00000000
            8a49f008: (0006,0094) Flags: 00000900  Mdl: 00000000
            8a227bc0: (0006,0094) Flags: 00000900  Mdl: 00000000
…
```

Choose an IRP, and examine it with the *!irp* command:

```
lkd> !irp 8a2c2c00
Irp is active with 1 stacks 1 is current (= 0x8a2c2c70)
 No Mdl Thread 8a124870:  Irp stack trace.
     cmd  flg cl Device   File     Completion-Context
>[  3, 0]   0  1 8a0e5680 8a26e4b8 00000000-00000000    pending
           \Driver\Npfs
            Args: 00000400 00000000 00000000 00000000
```

This IRP has a major function of 3, which corresponds to IRP_MJ_READ. It has one stack location and is targeted at a device owned by the Npfs driver (the Named Pipe File System driver). (Npfs is described in Chapter 13.)

IRP Buffer Management

When an application or a device driver indirectly creates an IRP by using the *NtReadFile*, *NtWriteFile*, or *NtDeviceIoControlFile* system services (or the Windows API functions corresponding to these services, which are *ReadFile*, *WriteFile*, and *DeviceIoControl*), the I/O manager determines whether it needs to participate in the management of the caller's input or output buffers. The I/O manager performs three types of buffer management:

- **Buffered I/O** The I/O manager allocates a buffer in nonpaged pool of equal size to the caller's buffer. For write operations, the I/O manager copies the caller's buffer data into the allocated buffer when creating the IRP. For read operations, the I/O manager copies data from the allocated buffer to the user's buffer when the IRP completes and then frees the allocated buffer.
- **Direct I/O** When the I/O manager creates the IRP, it locks the user's buffer into memory (makes it nonpaged). When the I/O manager has finished using the IRP, it unlocks the buffer. The I/O manager stores a description of the memory in the form of a *memory descriptor list* (MDL). An MDL specifies the physical memory occupied by a buffer. (See the Windows DDK for more information on MDLs.) Devices that perform direct memory access (DMA) require only physical descriptions of buffers, so an MDL is sufficient

for the operation of such devices. (Devices that support DMA transfer data directly between the device and the computer's memory, without using the CPU.) If a driver must access the contents of a buffer, however, it can map the buffer into the system's address space.

- **Neither I/O** The I/O manager doesn't perform any buffer management. Instead, buffer management is left to the discretion of the device driver, which can choose to manually perform the steps the I/O manager performs with the other buffer management types.

For each type of buffer management, the I/O manager places applicable references in the IRP to the locations of the input and output buffers. The type of buffer management the I/O manager performs depends on the type of buffer management a driver requests for each type of operation. A driver registers the type of buffer management it desires for read and write operations in the device object that represents the device. Device I/O control operations (those performed by *NtDeviceIoControlFile*) are specified with driver-defined I/O control codes, and a control code includes a description of the buffer management the I/O manager should use when issuing IRPs that contain that code.

Drivers commonly use buffered I/O when callers transfer requests smaller than one page (4 KB on x86 processors) and use direct I/O for larger requests. A page is approximately the buffer size at which the trade-off between the copy operation of buffered I/O matches the overhead of the memory lock performed by direct I/O. File system drivers commonly use neither I/O because no buffer management overhead is incurred when data can be copied from the file system cache into the caller's original buffer. The reason that most drivers don't use either I/O is that a pointer to a caller's buffer is valid only while a thread of the caller's process is executing. If a driver must transfer data from or to a device in an ISR or a DPC routine, it must ensure that the caller's data is accessible from any process context, which means that the buffer must have a system virtual address.

Drivers that use neither I/O to access buffers that might be located in user-space must take special care to ensure that buffer addresses are both valid and do not reference kernel-mode memory. Failure to do so could result in crashes or in security vulnerabilities, where applications have access to kernel-mode memory or can inject code into the kernel. The *ProbeForRead* and *ProbeForWrite* functions that the kernel makes available to drivers verify that a buffer resides entirely in the user-mode portion of the address space. To avoid a crash from referencing an invalid user-mode address, drivers can access user-mode buffers from within exception-handling code (called try/except blocks) that catch any invalid memory faults and translate them into error codes to return to the application.

I/O Request to a Single-Layered Driver

This section traces a synchronous I/O request to a single-layered kernel-mode device driver. Handling a synchronous I/O to a single-layered driver consists of seven steps:

1. The I/O request passes through a subsystem DLL.

2. The subsystem DLL calls the I/O manager's *NtWriteFile* service.
3. The I/O manager allocates an IRP describing the request and sends it to the driver (a device driver in this case) by calling its own *IoCallDriver* function.
4. The driver transfers the data in the IRP to the device and starts the I/O operation.
5. The driver signals I/O completion by interrupting the CPU.
6. When the device completes the operation and interrupts the CPU, the device driver services the interrupt.
7. The driver calls the I/O manager's *IoCompleteRequest* function to inform it that it has finished processing the IRP's request, and the I/O manager completes the I/O request.

These seven steps are illustrated in Figure 9-10.

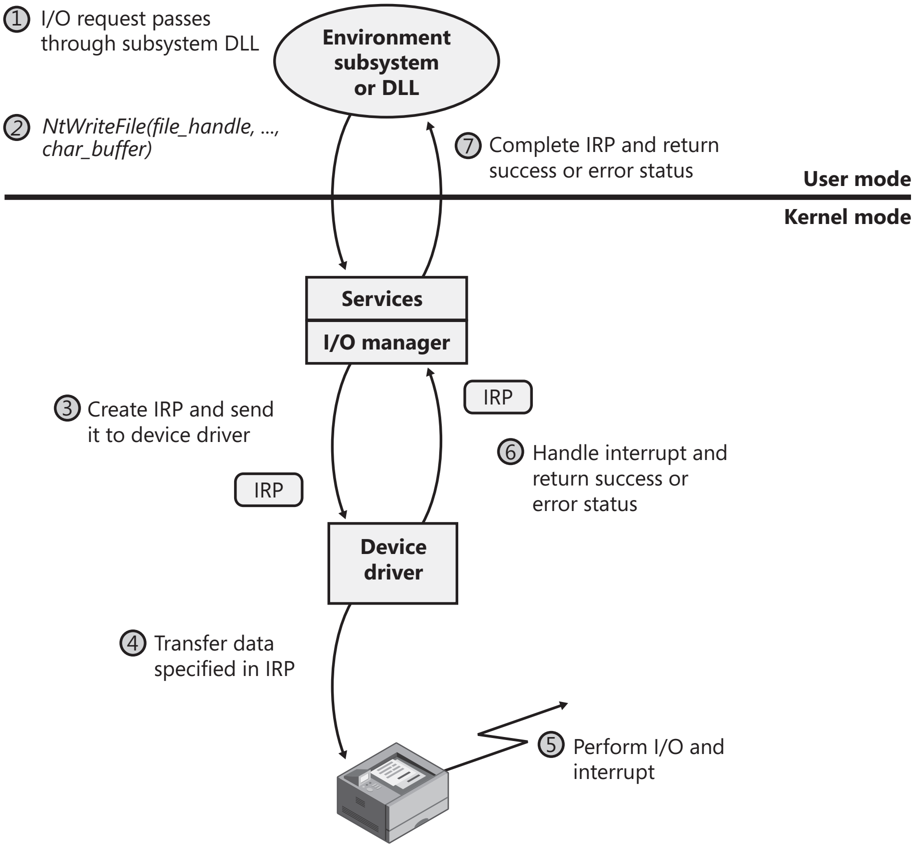

1 I/O request passes through subsystem DLL
Environment subsystem or DLL
2 NtWriteFile(file_handle, ..., char_buffer)
7 Complete IRP and return success or error status
User mode
Kernel mode
Services
I/O manager
IRP
3 Create IRP and send it to device driver
6 Handle interrupt and return success or error status
IRP
Device driver
4 Transfer data specified in IRP
5 Perform I/O and interrupt

Figure 9-10 Queuing and completing a synchronous request

Now that we've seen how an I/O is initiated, let's take a closer look at interrupt processing and I/O completion.

Servicing an Interrupt

After an I/O device completes a data transfer, it interrupts for service and the Windows kernel, I/O manager, and device driver are called into action. Figure 9-11 illustrates the first phase of the process. (Chapter 3 describes the interrupt dispatching mechanism, including DPCs. We've included a brief recap here because DPCs are key to I/O processing.)

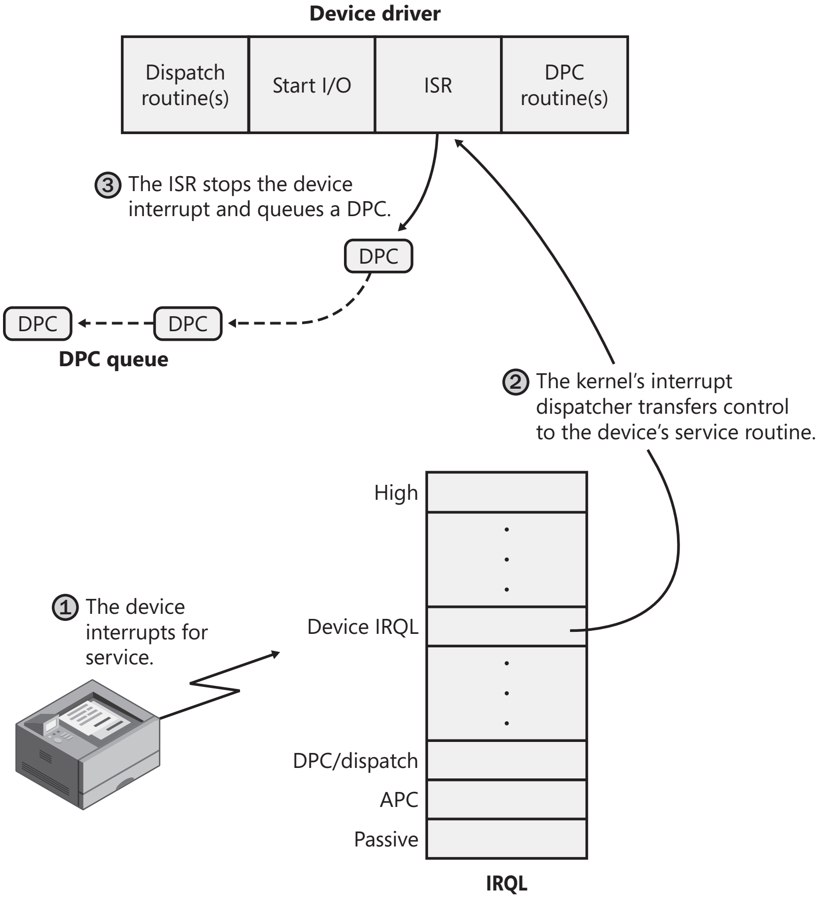

Device driver
Dispatch routine(s)
Start I/O
ISR
DPC routine(s)
3 The ISR stops the device interrupt and queues a DPC.
DPC
DPC
DPC
DPC queue
2 The kernel's interrupt dispatcher transfers control to the device's service routine.
High
Device IRQL
DPC/dispatch
APC
Passive
IRQL
1 The device interrupts for service.

Figure 9-11 Servicing a device interrupt (phase 1)

When a device interrupt occurs, the processor transfers control to the kernel trap handler, which indexes into its interrupt dispatch table to locate the ISR for the device. ISRs in Windows typically handle device interrupts in two steps. When an ISR is first invoked, it usually remains at device IRQL only long enough to capture the device status and then stop the device's interrupt. It then queues a DPC and exits, dismissing the interrupt. Later, when the DPC routine is called, the device finishes processing the interrupt. When that's done, the device calls the I/O manager to complete the I/O and dispose of the IRP. It might also start the next I/O request that is waiting in the device queue.

The advantage of using a DPC to perform most of the device servicing is that any blocked interrupt whose priority lies between the device IRQL and the DPC/dispatch IRQL is allowed to occur before the lower-priority DPC processing occurs. Intermediate-level interrupts are thus serviced more promptly than they otherwise would be. This second phase of an I/O (the DPC processing) is illustrated in Figure 9-12.

Completing an I/O Request

After a device driver's DPC routine has executed, some work still remains before the I/O request can be considered finished. This third stage of I/O processing is called *I/O completion* and is initiated when a driver calls *IoCompleteRequest* to inform the I/O manager that it has completed processing the request specified in the IRP (and the stack location that it owns). The steps I/O completion entails vary with different I/O operations. For example, all the I/O services record the outcome of the operation in an *I/O status block,* a data structure the caller supplies. Similarly, some services that perform buffered I/O require the I/O system to return data to the calling thread.

In both cases, the I/O system must copy some data that is stored in system memory into the caller's virtual address space. If the IRP completed synchronously, the caller's address space is current and directly accessible, but if the IRP completed asynchronously, the I/O manager must delay IRP completion until it can access the caller's address space. To gain access to the caller's virtual address space, the I/O manager must transfer the data "in the context of the caller's thread"—that is, while the caller's thread is executing (which means that caller's process is the current process and has its address space active on the processor). It does so by queuing a kernel-mode asynchronous procedure call (APC) to the thread. This process is illustrated in Figure 9-13.

As explained in Chapter 3, APCs execute in the context of a particular thread, whereas a DPC executes in arbitrary thread context, meaning that the DPC routine can't touch the user-mode process address space. Remember too that DPCs have a higher software interrupt priority than APCs.

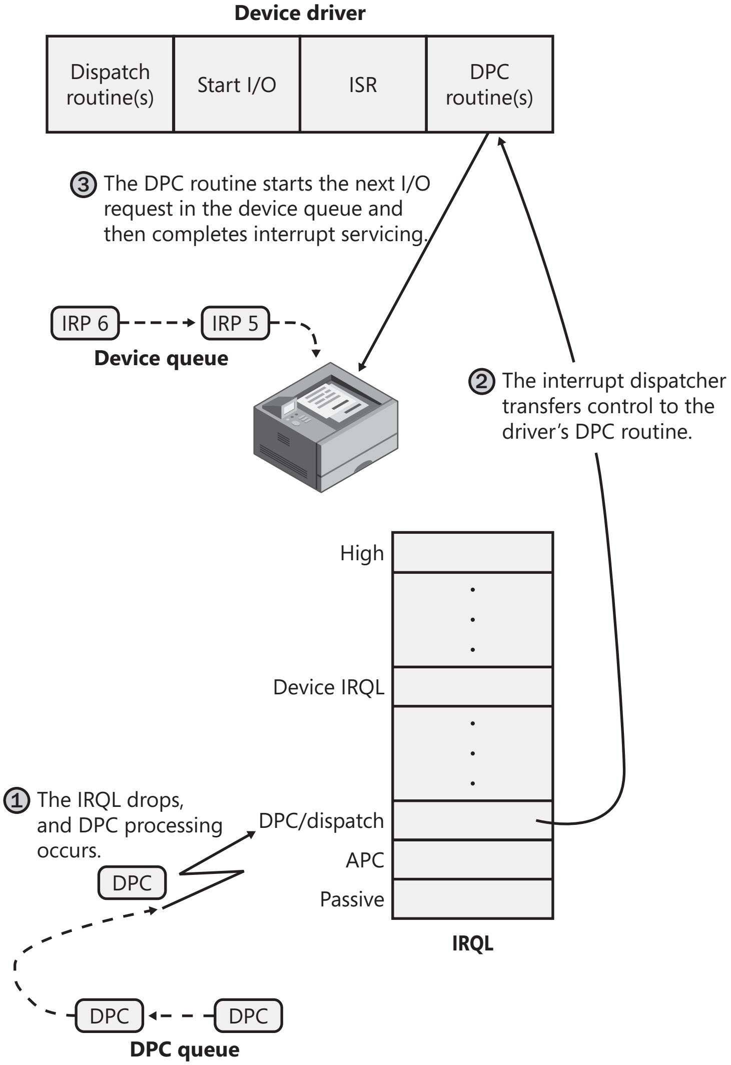

Device driver
Dispatch routine(s)
Start I/O
ISR
DPC routine(s)
3 The DPC routine starts the next I/O request in the device queue and then completes interrupt servicing.
IRP 6
IRP 5
Device queue
2 The interrupt dispatcher transfers control to the driver's DPC routine.
High
Device IRQL
DPC/dispatch
APC
Passive
IRQL
1 The IRQL drops, and DPC processing occurs.
DPC
DPC
DPC
DPC queue

Figure 9-12 Servicing a device interrupt (phase 2)

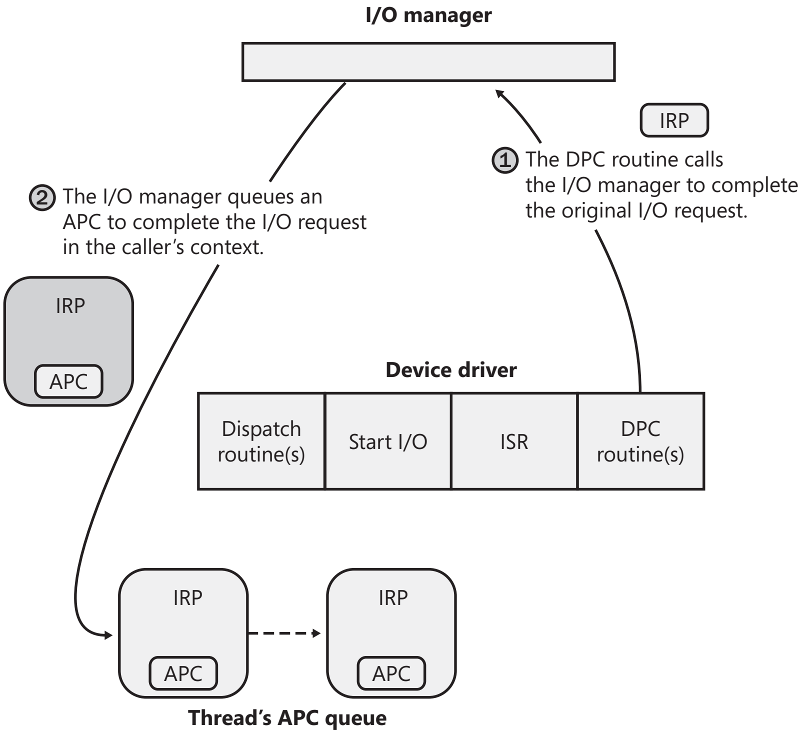

I/O manager
IRP
① The DPC routine calls the I/O manager to complete the original I/O request.
② The I/O manager queues an APC to complete the I/O request in the caller's context.
IRP
APC
Device driver
Dispatch routine(s)
Start I/O
ISR
DPC routine(s)
IRP
APC
IRP
APC
Thread's APC queue

Figure 9-13 Completing an I/O request (phase 1)

The next time that thread begins to execute at low IRQL, the pending APC is delivered. The kernel transfers control to the I/O manager's APC routine, which copies the data (if any) and the return status into the original caller's address space, frees the IRP representing the I/O operation, and sets the caller's file handle (and any caller-supplied event or I/O completion port) to the signaled state. The I/O is now considered complete. The original caller or any other threads that are waiting on the file (or other object) handle are released from their waiting state and readied for execution. Figure 9-14 illustrates the second stage of I/O completion.

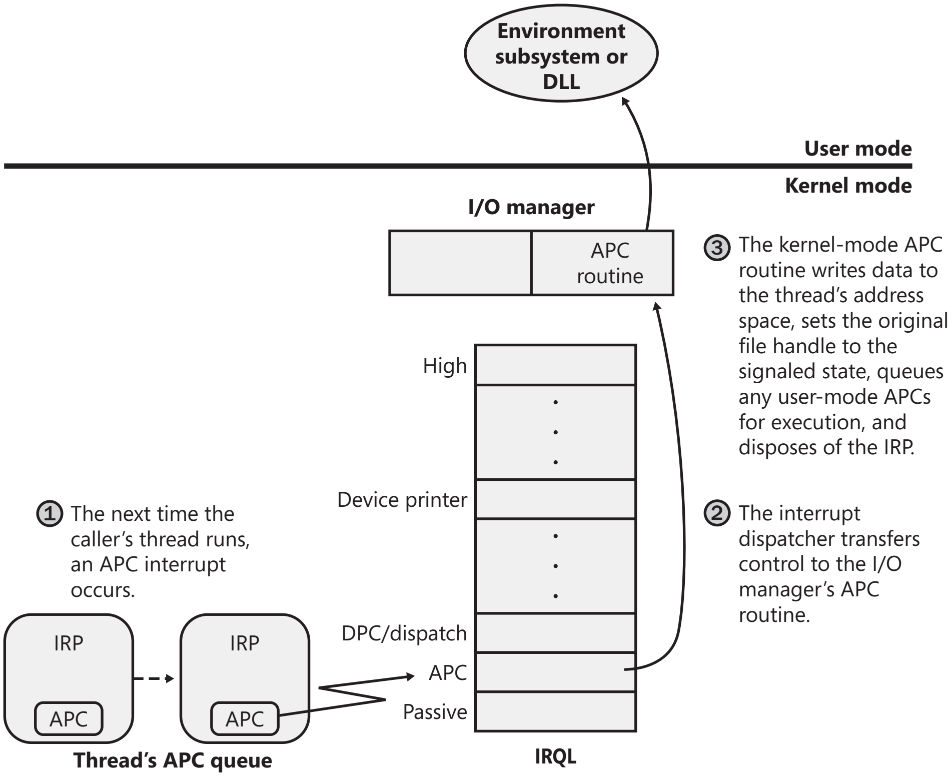

Environment subsystem or DLL
User mode
Kernel mode
I/O manager
APC routine
③ The kernel-mode APC routine writes data to the thread's address space, sets the original file handle to the signaled state, queues any user-mode APCs for execution, and disposes of the IRP.
High
Device printer
② The interrupt dispatcher transfers control to the I/O manager's APC routine.
① The next time the caller's thread runs, an APC interrupt occurs.
DPC/dispatch
APC
Passive
IRP
APC
IRP
APC
Thread's APC queue
IRQL

Figure 9-14 Completing an I/O request (phase 2)

A final note about I/O completion: the asynchronous I/O functions *ReadFileEx* and *WriteFileEx* allow a caller to supply a user-mode APC as a parameter. If the caller does so, the I/O manager queues this APC to the caller's thread APC queue as the last step of I/O completion. This feature allows a caller to specify a subroutine to be called when an I/O request is completed or canceled. User-mode APC completion routines execute in the context of the requesting thread and are delivered only when the thread enters an alertable wait state (such as calling the Windows *SleepEx*, *WaitForSingleObjectEx*, or *WaitForMultipleObjectsEx* function).

Synchronization

Drivers must synchronize their access to global driver data and hardware registers for two reasons:

- The execution of a driver can be preempted by higher-priority threads and time-slice (or quantum) expiration or can be interrupted by interrupts.
- On multiprocessor systems, Windows can run driver code simultaneously on more than one processor.

Without synchronization, corruption could occur—for example, because device driver code running at a passive IRQL when a caller initiates an I/O operation can be interrupted by a device interrupt, causing the device driver's ISR to execute while its own device driver is already running. If the device driver was modifying data that its ISR also modifies, such as device registers, heap storage, or static data, the data can become corrupted when the ISR executes. Figure 9-15 illustrates this problem.

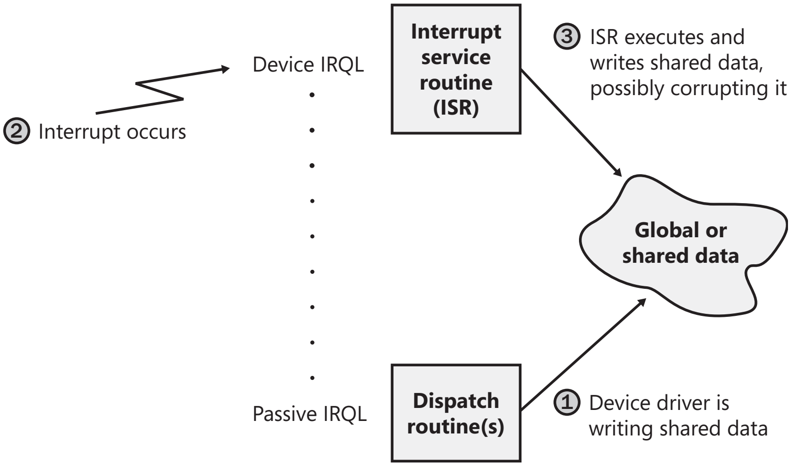

Interrupt service routine (ISR)
③ ISR executes and writes shared data, possibly corrupting it
Device IRQL
② Interrupt occurs
Global or shared data
Passive IRQL
Dispatch routine(s)
① Device driver is writing shared data

Figure 9-15 Queuing an asynchronous request to layered drivers

To avoid this situation, a device driver written for Windows must synchronize its access to any data that the device driver shares with its ISR. Before attempting to update shared data, the device driver must lock out all other threads (or CPUs, in the case of a multiprocessor system) to prevent them from updating the same data structure.

The Windows kernel provides a special synchronization routine called *KeSynchronizeExecution* that device drivers call when they access data that their ISRs also access. This kernel-synchronization routine keeps the ISR from executing while the shared data is being accessed. On a single CPU system, this routine raises the IRQL to the level associated with the ISR before updating a structure. On a multiprocessor system, however, because a driver can execute on two or more processors at once, this technique isn't enough to block other accessors. Therefore, another mechanism, a *spinlock*, is used to lock a structure for exclusive access from a particular CPU. (Spinlocks are explained in the section "Synchronization" in Chapter 3.) A driver can also use *KeAcquireInterruptSpinLock* to access an interrupt object's spin lock directly, although it's generally faster to rely on *KeSynchronizeExecution* for synchronization with an ISR.

By now, you should realize that although ISRs require special attention, any data that a device driver uses is subject to being accessed by the same device driver running on another processor. Therefore, it's critical for device driver code to synchronize its use of any global or shared data (or any accesses to the physical device itself). If the ISR uses that data, the device driver must use *KeSynchronizeExecution*; otherwise, the device driver can use standard kernel spinlocks.

I/O Requests to Layered Drivers

The preceding section showed how an I/O request to a simple device controlled by a single device driver is handled. I/O processing for file-based devices or for requests to other layered drivers happens in much the same way. The major difference is, obviously, that one or more additional layers of processing are added to the model.

Figure 9-16 shows how an asynchronous I/O request travels through layered drivers. It uses as an example a disk controlled by a file system.

Once again, the I/O manager receives the request and creates an I/O request packet to represent it. This time, however, it delivers the packet to a file system driver. The file system driver exercises great control over the I/O operation at that point. Depending on the type of request the caller made, the file system can send the same IRP to the disk driver or it can generate additional IRPs and send them separately to the disk driver.

EXPERIMENT: Viewing a Device Stack

The kernel debugger command *!devstack* shows you the device stack of layered device objects associated with a specified device object. This example shows the device stack associated with a device object, \device\keyboardclass0, which is owned by the keyboard class driver:

```
lkd> !devstack keyboardclass0
  !DevObj   !DrvObj            !DevExt   ObjectName
  8a266d28  \Driver\Ctrl2cap   8a266de0
> 8a09a030  \Driver\Kbdclass   8a09a0e8  KeyboardClass0
  8a2672b0  \Driver\nmfilter   8a267368  0000008c
  8a09ba78  \Driver\i8042prt   8a09bb30
  8a4adce0  \Driver\ACPI       8a4ab9c8  0000006b
!DevNode 8a4acee8 :
  DeviceInst is "ACPI\PNP0303\4&61f3b4b&0"
  ServiceName is "i8042prt"
```

The output highlights the entry associated with KeyboardClass0 with the “>“ prefix. The entries above that line are drivers layered above the keyboard class driver, and those below are layered beneath it. In general, IRPs flow from the top of the stack to the bottom.

The file system is most likely to reuse an IRP if the request it receives translates into a single straightforward request to a device. For example, if an application issues a read request for the first 512 bytes in a file stored on a floppy disk, the FAT file system would simply call the disk driver, asking it to read one sector from the floppy disk, beginning at the file’s starting location.

To accommodate its reuse by multiple drivers in a request to layered drivers, an IRP contains a series of *IRP stack locations* (not to be confused with the stack used by threads to store function parameters and return addresses). These data areas, one for every driver that will be called, contain the information that each driver needs to execute its part of the request—for example, function code, parameters, and driver context information. As Figure 9-16 illustrates, additional stack locations are filled in as the IRP passes from one driver to the next. You can think of an IRP as being similar to a stack in the way data is added to it and removed from it during its lifetime. However, an IRP isn't associated with any particular process, and its allocated size doesn't grow and shrink. The I/O manager allocates an IRP from one if its IRP lookaside lists or nonpaged system memory at the beginning of the I/O operation.

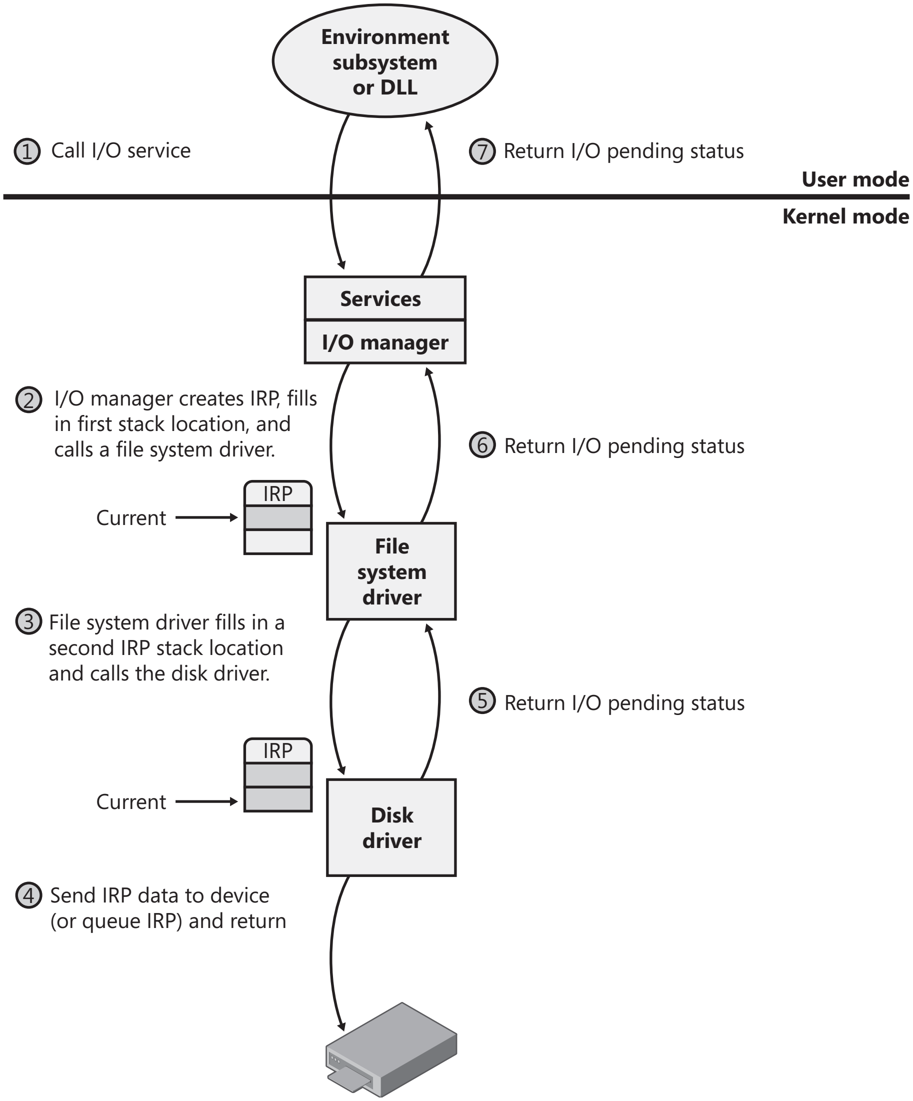

Environment subsystem or DLL
① Call I/O service
⑦ Return I/O pending status
User mode
Kernel mode
Services
I/O manager
② I/O manager creates IRP, fills in first stack location, and calls a file system driver.
⑥ Return I/O pending status
IRP
Current
File system driver
③ File system driver fills in a second IRP stack location and calls the disk driver.
⑤ Return I/O pending status
IRP
Current
Disk driver
④ Send IRP data to device (or queue IRP) and return

Figure 9-16 Queuing an asynchronous request to layered drivers

EXPERIMENT: Examining IRPs

In this experiment, you'll find an uncompleted IRP on the system, and you'll determine the IRP type, the device at which it's directed, the driver that manages the device, the thread that issued the IRP, and what process the thread belongs to.

At any point in time, there are at least a few uncompleted IRPs on a system. This is because there are many devices to which applications can issue IRPs that a driver will only complete when a particular event occurs, such as data becoming available. One example is a blocking read from a network endpoint. You can see the outstanding IRPs on a system with the *!irpfind* kernel debugger command:

```
kd> !irpfind
unable to get large pool allocation table - either wrong symbols or pool tagging
 is disabled

Searching NonPaged pool (82502000 : 8a502000) for Tag: Irp?

  Irp    [ Thread ] irpStack: (Mj,Mn)   DevObj  [Driver]
89695868 [00000000] Irp is complete (CurrentLocation 4 > StackCount 3) 0x43776f56
89712008 [8a29d7c0] irpStack: ( e, 9)  8a19e208 [ \Driver\AFD]
89716008 [8a29d7c0] irpStack: ( e, 9)  8a19e208 [ \Driver\AFD]
… 89cb3928 [8a3acbc0] irpStack: ( 3, 0)  8a09a030 [ \Driver\Kbdclass]
89cb3c88 [89cb1da8] irpStack: ( c, 2)  8a436020 [ \FileSystem\Ntfs]
89cb4640 [8a165498] irpStack: ( e, 9)  8a19e208 [ \Driver\AFD]
```

The highlighted entry in the output describes an IRP that is directed at the Kbdclass driver, so it is likely the IRP that was issued by the Windows subsystem raw input thread that reads keyboard input. Examining the IRP with the *!irp* command reveals:

```
kd> !irp 8a1716f0
Irp is active with 3 stacks 3 is current (= 0x8a1717a8)
 No Mdl System buffer = 8a458180 Thread 8a3acbc0:  Irp stack trace.
     cmd  flg cl Device   File     Completion-Context
 [  0, 0]   0  0 00000000 00000000 00000000-00000000

                        Args: 00000000 00000000 00000000 00000000
 [  0, 0]   0  0 00000000 00000000 00000000-00000000

                        Args: 00000000 00000000 00000000 00000000
>[  3, 0]   0  1 8a1eccb8 8a1262a8 00000000-00000000    pending
               \Driver\Kbdclass
                        Args: 00000078 00000000 00000000 00000000
```

The active stack location is at the bottom. (The debugger shows the active location with a ">" prefix.) It has a major function of 3, which corresponds to IRP_MJ_READ.

The next step is to see what device object the IRP is targeting by executing the *!devobj* command on the device object address in the active stack location

```
kd> !devobj 8a1eccb8
Device object (8a1eccb8) is for:
 KeyboardClass1 \Driver\Kbdclass DriverObject 8a24bd78
Current Irp 00000000 RefCount 0 Type 0000000b Flags 00002044
Dacl e1ec01e4 DevExt 8a1ecd70 DevObjExt 8a1ece50
ExtensionFlags (0000000000)
AttachedTo (Lower) 8a2e8ac8 \Driver\TermDD
Device queue is not busy.
```

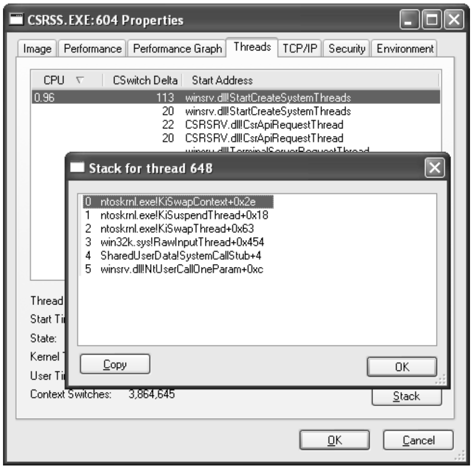
The device at which the IRP is targeted is KeyboardClass1. The presence of a device object owned by the Termdd driver attached beneath it reveals that it is the device that represents keyboard input from a Terminal Server Client, not the physical keyboard. (This output was taken from a Windows XP system.)

We can see details about the thread and process that issued the IRP by using the *!thread* and *!process* commands:

```
kd> !thread 8a3acbc0
THREAD 8a3acbc0  Cid 025c.0288  Teb: 7ffd9000 Win32Thread: e101fab0 WAIT:
rRequest) KernelMode Alertable
    8a3cdb30  SynchronizationEvent
    8a28b4e0  SynchronizationEvent
    8a3cc908  NotificationTimer
    8a294828  SynchronizationEvent
    805453e0  NotificationEvent
    8a2e1830  SynchronizationEvent
    8a45eeb0  SynchronizationTimer
IRP List:
    89cb3928: (0006,01b4) Flags: 00000970  Mdl: 00000000
    8a1716f0: (0006,01b4) Flags: 00000970  Mdl: 00000000
Not impersonating
```

```
DeviceMap                 e10087c0 Owning Process           8a3a08b8
Wait Start TickCount      6844081
Context Switch Count      2130848               LargeStack
UserTime                  00:00:00.0000
KernelTime                00:00:03.0274
Start Address 0x75b6e8ad
Stack Init bafa0000 Current baf9fa68 Base bafa0000 Limit baf9d000 Call 0
Priority 13 BasePriority 13 PriorityDecrement 0 DecrementCount 16

kd> !process 8a3a08b8 0
GetPointerFromAddress: unable to read from 80543ed4
PROCESS 8a3a08b8  SessionId: 0  Cid: 025c    Peb: 7ffdf000  ParentCid: 0220
    DirBase: 139af000  ObjectTable: e1c0b3d8  HandleCount: 581.
    Image: CSRSS.EXE
```

Locating the thread in Process Explorer (from *www.sysinternals.com*) by opening the Properties dialog box for Csrss.exe and going to the Threads tab confirms, through the names of the functions on its stack, the role of the thread as a raw input thread for the Windows subsystem:



```
CPU        CSwitch Delta   Start Address
0.96                 113   winsrv.dll!StartCreateSystemThreads
                      20   winsrv.dll!StartCreateSystemThreads
                      22   CSRSRV.dll!CsrApiRequestThread
                      20   CSRSRV.dll!CsrApiRequestThread
```

```
0   ntoskrnl.exe!KiSwapContext+0x2e
1   ntoskrnl.exe!KiSuspendThread+0x18
2   ntoskrnl.exe!KiSwapThread+0x63
3   win32k.sys!RawInputThread+0x454
4   SharedUserData!SystemCallStub+4
5   winsrv.dll!NtUserCallOneParam+0xc
```

Context Switches: 3,864,645

After the disk driver finishes a data transfer, the disk interrupts and the I/O completes, as shown in Figure 9-17.

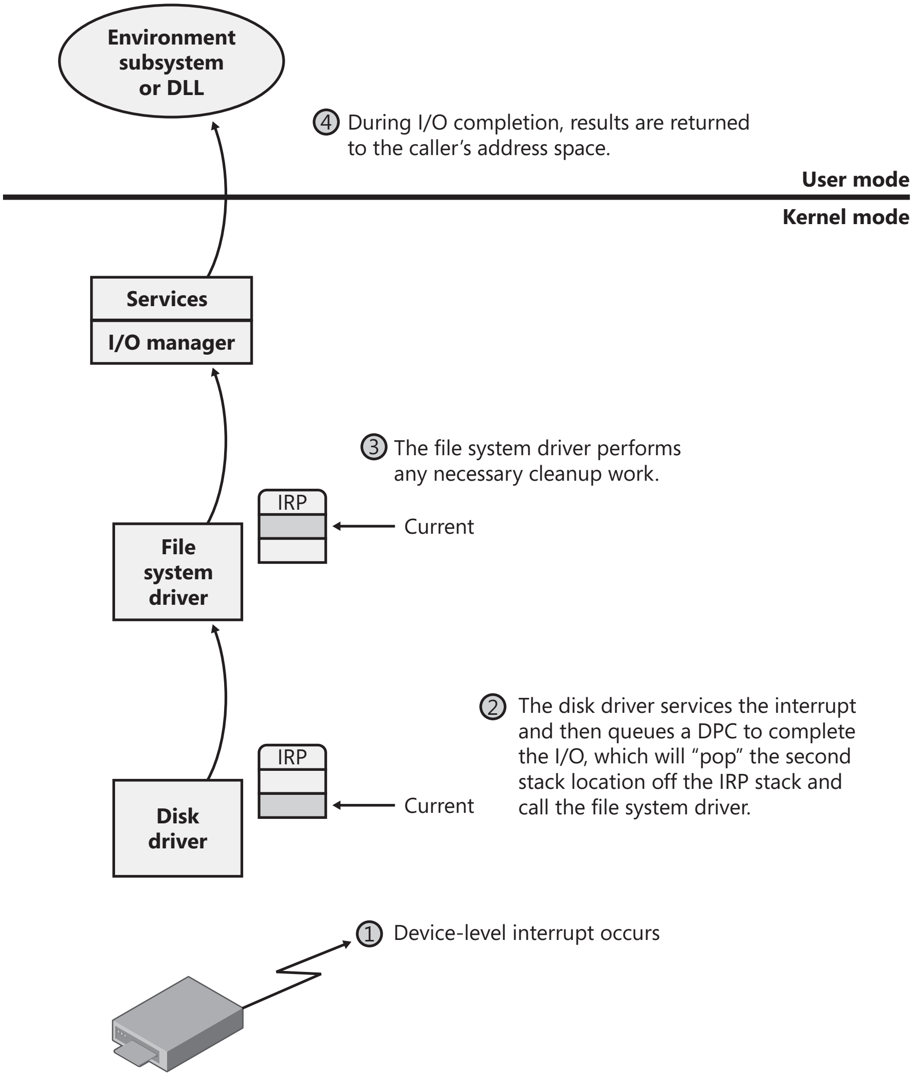

Environment subsystem or DLL
4 During I/O completion, results are returned to the caller's address space.
User mode
Kernel mode
Services
I/O manager
3 The file system driver performs any necessary cleanup work.
IRP
Current
File system driver
2 The disk driver services the interrupt and then queues a DPC to complete the I/O, which will "pop" the second stack location off the IRP stack and call the file system driver.
IRP
Current
Disk driver
1 Device-level interrupt occurs

Figure 9-17 Completing a layered I/O request

As an alternative to reusing a single IRP, a file system can establish a group of *associated* IRPs that work in parallel on a single I/O request. For example, if the data to be read from a file is dispersed across the disk, the file system driver might create several IRPs, each of which reads some portion of the request from a different sector. This queuing is illustrated in Figure 9-18.

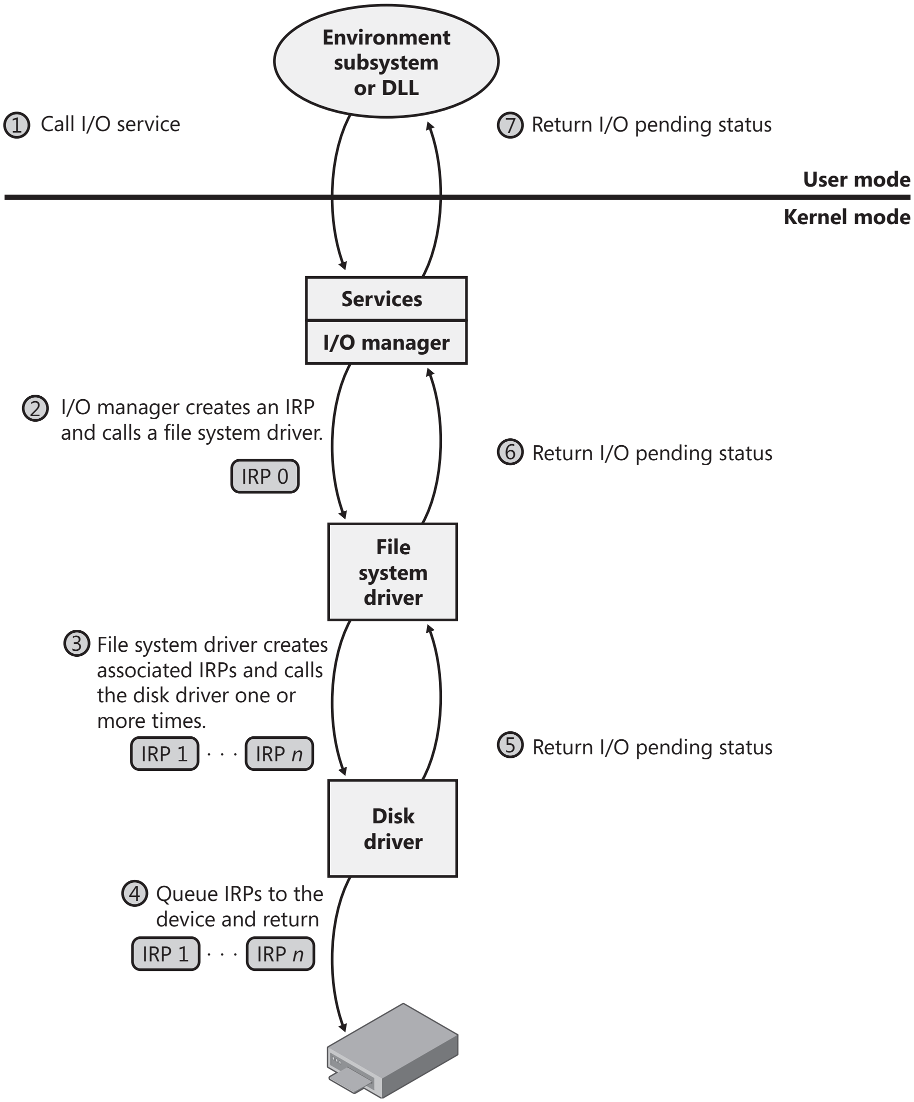

Environment subsystem or DLL
1 Call I/O service
7 Return I/O pending status
User mode
Kernel mode
Services
I/O manager
2 I/O manager creates an IRP and calls a file system driver.
IRP 0
6 Return I/O pending status
File system driver
3 File system driver creates associated IRPs and calls the disk driver one or more times.
IRP 1 · · · IRP n
5 Return I/O pending status
Disk driver
4 Queue IRPs to the device and return
IRP 1 · · · IRP n

Figure 9-18 Queuing associated IRPs

The file system driver delivers the associated IRPs to the device driver, which queues them to the device. They are processed one at a time, and the file system driver keeps track of the returned data. When all the associated IRPs complete, the I/O system completes the original IRP and returns to the caller, as shown in Figure 9-19.

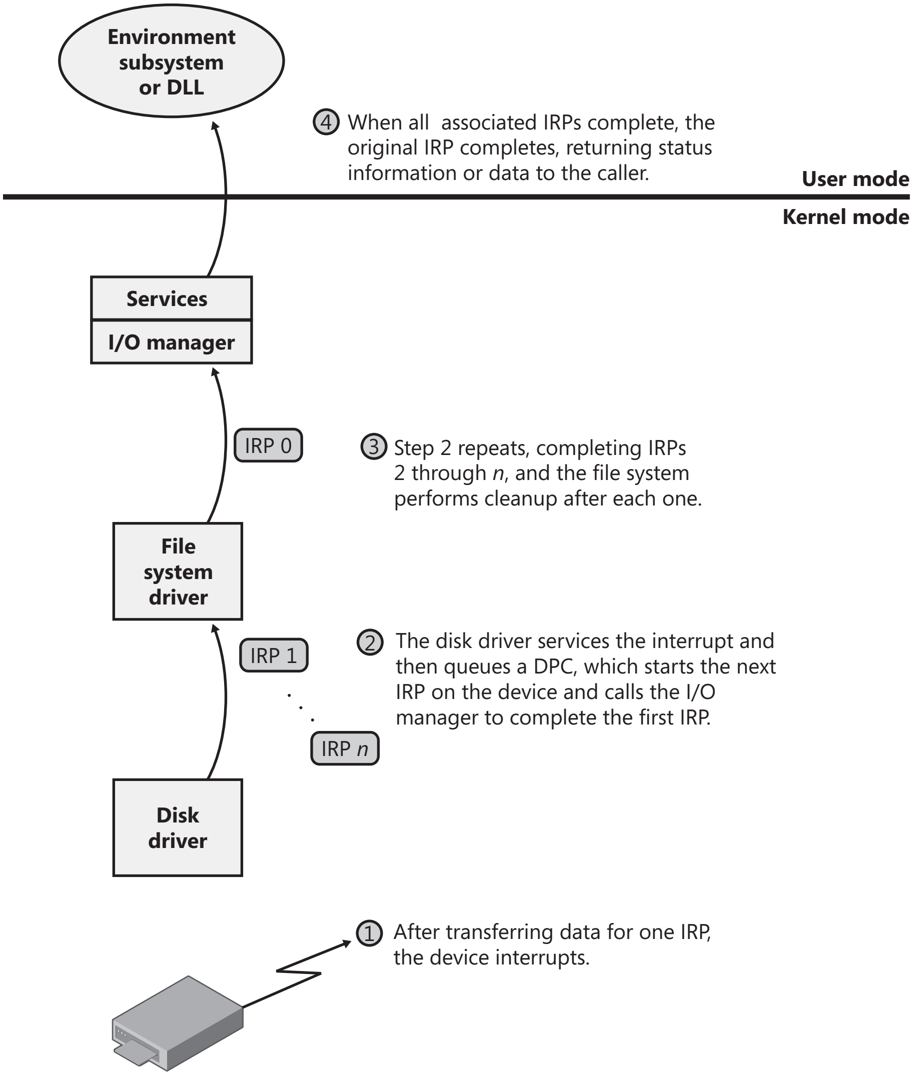

Environment subsystem or DLL
4 When all associated IRPs complete, the original IRP completes, returning status information or data to the caller.
User mode
Kernel mode
Services
I/O manager
IRP 0
3 Step 2 repeats, completing IRPs 2 through *n*, and the file system performs cleanup after each one.
File system driver
IRP 1
IRP *n*
2 The disk driver services the interrupt and then queues a DPC, which starts the next IRP on the device and calls the I/O manager to complete the first IRP.
Disk driver
1 After transferring data for one IRP, the device interrupts.

Figure 9-19 Completing associated IRPs

Note All Windows file system drivers that manage disk-based file systems are part of a stack of drivers that is at least three layers deep: the file system driver sits at the top, a volume manager in the middle, and a disk driver at the bottom. In addition, any number of filter drivers can be interspersed above and below these drivers. For clarity, the preceding example of layered I/O requests includes only a file system driver and a disk device driver. See Chapter 10, on storage management, for more information.

I/O Completion Ports

Writing a high-performance server application requires implementing an efficient threading model. Having either too few or too many server threads to process client requests can lead to performance problems. For example, if a server creates a single thread to handle all requests, clients can become starved because the server will be tied up processing one request at a time. A single thread could simultaneously process multiple requests, switching from one to another as I/O operations are started, but this architecture introduces significant complexity and can't take advantage of multiprocessor systems. At the other extreme, a server could create a big pool of threads so that virtually every client request is processed by a dedicated thread. This scenario usually leads to thread-thrashing, in which lots of threads wake up, perform some CPU processing, block while waiting for I/O, and then, after request processing is completed, block again waiting for a new request. If nothing else, having too many threads results in excessive context switching, caused by the scheduler having to divide processor time among multiple active threads.

The goal of a server is to incur as few context switches as possible by having its threads avoid unnecessary blocking, while at the same time maximizing parallelism by using multiple threads. The ideal is for there to be a thread actively servicing a client request on every processor and for those threads not to block when they complete a request if additional requests are waiting. For this optimal process to work correctly, however, the application must have a way to activate another thread when a thread processing a client request blocks on I/O (such as when it reads from a file as part of the processing).

The *IoCompletion* Object

Applications use the *IoCompletion* executive object, which is exported to Windows as a *completion port*, as the focal point for the completion of I/O associated with multiple file handles. Once a file is associated with a completion port, any asynchronous I/O operations that complete on the file result in a completion packet being queued to the completion port. A thread can wait for any outstanding I/Os to complete on multiple files simply by waiting for a completion packet to be queued to the completion port. The Windows API provides similar functionality with the *WaitForMultipleObjects* API function, but the advantage that completion ports have is that *concurrency*, or the number of threads that an application has actively servicing client requests, is controlled with the aid of the system.

When an application creates a completion port, it specifies a concurrency value. This value indicates the maximum number of threads associated with the port that should be running at any given time. As stated earlier, the ideal is to have one thread active at any given time for every processor in the system. Windows uses the concurrency value associated with a port to control how many threads an application has active. If the number of active threads associated with a port equals the concurrency value, a thread that is waiting on the completion port won't be allowed to run. Instead, it is expected that one of the active threads will finish processing its current request and check to see whether another packet is waiting at the port. If one is, the thread simply grabs the packet and goes off to process it. When this happens, there is no context switch, and the CPUs are utilized nearly to their full capacity.

Using Completion Ports

Figure 9-20 shows a high-level illustration of completion port operation. A completion port is created with a call to the Windows API function *CreateIoCompletionPort*. Threads that block on a completion port become associated with the port and are awakened in last in, first out (LIFO) order so that the thread that blocked most recently is the one that is given the next packet. Threads that block for long periods of time can have their stacks swapped out to disk, so if there are more threads associated with a port than there is work to process, the in-memory footprints of threads blocked the longest are minimized.

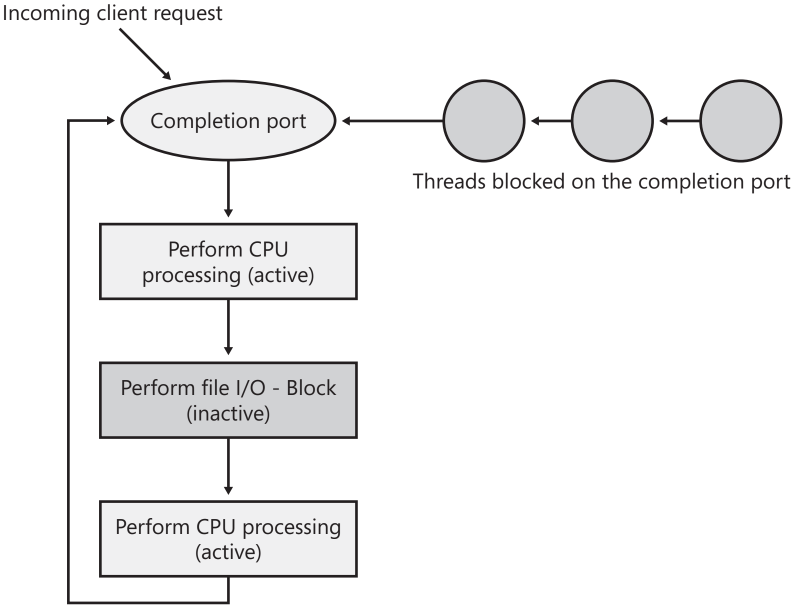

Incoming client request
Completion port
Threads blocked on the completion port
Perform CPU processing (active)
Perform file I/O - Block (inactive)
Perform CPU processing (active)

Figure 9-20 I/O completion port operation

A server application will usually receive client requests via network endpoints that are represented as file handles. Examples include Windows Sockets 2 (Winsock2) sockets or named pipes. As the server creates its communications endpoints, it associates them with a completion port and its threads wait for incoming requests by calling *GetQueuedCompletionStatus* on the port. When a thread is given a packet from the completion port, it will go off and start processing the request, becoming an active thread. A thread will block many times during its processing, such as when it needs to read or write data to a file on disk or when it synchronizes with other threads. Windows detects this activity and recognizes that the completion port has one less active thread. Therefore, when a thread becomes inactive because it blocks, a thread waiting on the completion port will be awakened if there is a packet in the queue.

Microsoft's guidelines are to set the concurrency value roughly equal to the number of processors in a system. Keep in mind that it's possible for the number of active threads for a completion port to exceed the concurrency limit. Consider a case in which the limit is specified as 1. A client request comes in, and a thread is dispatched to process the request, becoming active. A second request arrives, but a second thread waiting on the port isn't allowed to proceed because the concurrency limit has been reached. Then the first thread blocks waiting for a file I/O, so it becomes inactive. The second thread is then released, and while it's still active, the first thread's file I/O is completed, making it active again. At that point—and until one of the threads blocks—the concurrency value is 2, which is higher than the limit of 1. Most of the time, the active count will remain at or just above the concurrency limit.

The completion port API also makes it possible for a server application to queue privately defined completion packets to a completion port by using the *PostQueuedCompletionStatus* function. A server typically uses this function to inform its threads of external events, such as the need to shut down gracefully.

I/O Completion Port Operation

Windows applications create completion ports by calling the Windows API *CreateIoCompletionPort* and specifying a NULL completion port handle. This results in the execution of the *NtCreateIoCompletion* system service. The executive's *IoCompletion* object is based on the kernel synchronization object called a *queue*. Thus, the system service creates a completion port object and initializes a queue object in the port's allocated memory. (A pointer to the port also points to the queue object because the queue is at the start of the port memory.) A queue object has a concurrency value that is specified when a thread initializes one, and in this case the value that is used is the one that was passed to *CreateIoCompletionPort*. *KeInitializeQueue* is the function that *NtCreateIoCompletion* calls to initialize a port's queue object.

When an application calls *CreateIoCompletionPort* to associate a file handle with a port, the *NtSetInformationFile* system service is executed with the file handle as the primary parameter. The information class that is set is *FileCompletionInformation*, and the completion port's handle and the *CompletionKey* parameter from *CreateIoCompletionPort* are the data values. *NtSetInformationFile* dereferences the file handle to obtain the file object and allocates a completion context data structure.

Finally, *NtSetInformationFile* sets the *CompletionContext* field in the file object to point at the context structure. When an asynchronous I/O operation completes on a file object, the I/O manager checks to see whether the *CompletionContext* field in the file object is non-NULL. If it is, the I/O manager allocates a completion packet and queues it to the completion port by calling *KeInsertQueue* with the port as the queue on which to insert the packet. (Remember that the completion port object and queue object are synonymous.)

When a server thread invokes *GetQueuedCompletionStatus*, the system service *NtRemoveIoCompletion* is executed. After validating parameters and translating the completion port handle to a pointer to the port, *NtRemoveIoCompletion* calls *KeRemoveQueue*.

As you can see, *KeRemoveQueue* and *KeInsertQueue* are the engines behind completion ports. They are the functions that determine whether a thread waiting for an I/O completion packet should be activated. Internally, a queue object maintains a count of the current number of active threads and the maximum number of active threads. If the current number equals or exceeds the maximum when a thread calls *KeRemoveQueue*, the thread will be put (in LIFO order) onto a list of threads waiting for a turn to process a completion packet. The list of threads hangs off the queue object. A thread's control block data structure has a pointer in it that references the queue object of a queue that it's associated with; if the pointer is NULL, the thread isn't associated with a queue.

Windows keeps track of threads that become inactive because they block on something other than the completion port by relying on the queue pointer in a thread's control block. The scheduler routines that possibly result in a thread blocking (such as *KeWaitForSingleObject*, *KeDelayExecutionThread*, and so on) check the thread's queue pointer. If the pointer isn't NULL, the functions call *KiActivateWaiterQueue*, a queue-related function that decrements the count of active threads associated with the queue. If the resultant number is less than the maximum and at least one completion packet is in the queue, the thread at the front of the queue's thread list is awakened and given the oldest packet. Conversely, whenever a thread that is associated with a queue wakes up after blocking, the scheduler executes the function *KiUnwaitThread*, which increments the queue's active count.

Finally, the *PostQueuedCompletionStatus* Windows API function results in the execution of the *NtSetIoCompletion* system service. This function simply inserts the specified packet onto the completion port's queue by using *KeInsertQueue*.

Figure 9-21 shows an example of a completion port object in operation. Even though two threads are ready to process completion packets, the concurrency value of 1 allows only one thread associated with the completion port to be active, and so the two threads are blocked on the completion port.

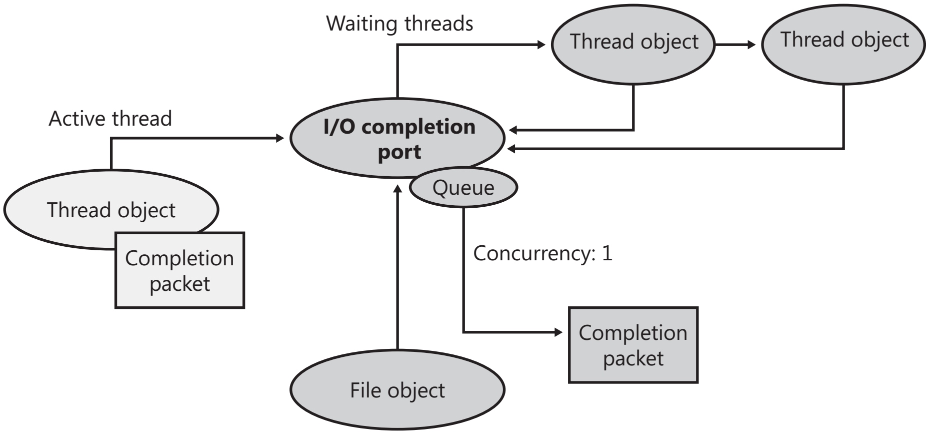
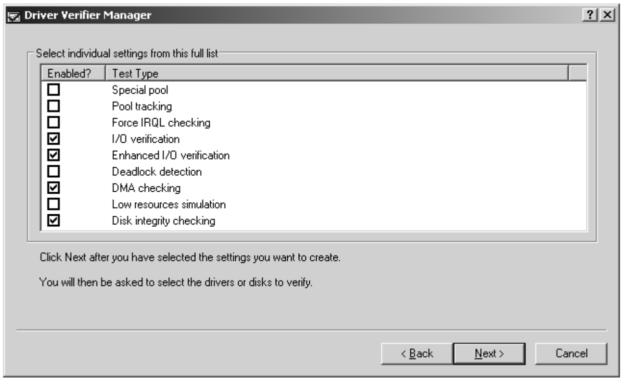


Figure 9-21 I/O completion port operation

Driver Verifier

The Driver Verifier, which was introduced in Chapter 7, includes several options that check the correctness of I/O-related operations. Figure 9-22 shows the Windows Server 2003 Driver Verifier Manager with these options selected.

Even when you don't select any options, the Verifier monitors drivers selected for verification, looking for a number of illegal operations–including calling kernel-memory pool functions at invalid IRQL, double-freeing memory, and requesting a zero-sized memory allocation.



Select individual settings from this full list

Enabled?	Test Type
□	Special pool
□	Pool tracking
□	Force IRQL checking
☑	I/O verification
☑	Enhanced I/O verification
□	Deadlock detection
☑	DMA checking
□	Low resources simulation
☑	Disk integrity checking

Click Next after you have selected the settings you want to create.

You will then be asked to select the drivers or disks to verify.

Figure 9-22 Driver Verifier I/O-related options

Additional checks enabled by checking I/O-related options include:

- **I/O Verification** When this option is selected, the I/O manager allocates IRPs for verified drivers from a special pool and their usage is tracked. In addition, the Verifier crashes the system when an IRP is completed that contains an invalid status and when an invalid device object is passed to the I/O manager. (In Windows 2000, this is called I/O Verification Level 1).
- **I/O Verification Level 2** This option exists only in Windows 2000 and results in more rigorous testing of IRP completion operations and stack usage.
- **Enhanced I/O Verification** This option was introduced in Windows XP, and it monitors all IRPs to ensure that drivers mark them correctly when completing them asynchronously, that they manage device stack locations correctly, and that they delete device objects only once. In addition, the Verifier randomly stresses drivers by sending them fake power management and WMI IRPs, changing the order that devices are enumerated, and adjusting the status of PnP and power IRPs when they complete to test for drivers that return incorrect status from their dispatch routines.
- **DMA Checking DMA – Direct Memory Access** This is a hardware-supported mechanism that allows devices to transfer data to or from physical memory without involving the CPU. The I/O manager provides a number of functions that drivers use to schedule and control DMA operations, and this option enables checks for correct use of the functions and for the buffers that the I/O manager supplies for DMA operations.

- **Disk Integrity Verification** When you enable this option, which is available only in Windows Server 2003, the Verifier monitors disk read and write operations and checksums the associated data. When disk reads complete, it checks to see whether it has a previously stored checksum and crashes the system if the new and old checksum don't match, because that would indicate corruption of the disk at the hardware level.
- **SCSI Verification** This was introduced in Windows XP and is not visible in the Driver Verifier option dialog box. However, it is enabled when you select a SCSI miniport driver for verification and enable at least one of the other options. When selected, the Verifier checks the SCSI miniport driver's usage of functions supplied by the SCSI miniport library drivers, storport.sys or scsiport.sys. Checks include ensuring that the driver does not complete a request more than once, that it doesn't pass invalid arguments, and that it doesn't take more than a certain amount of time to complete operations. (See Chapter 10 for more information on SCSI miniport drivers.)

The Driver Verifier serves primarily as a tool to help device driver developers discover bugs in their code, but it can also be a powerful tool for systems administrators experiencing crashes. Chapter 14 describes its role in crash analysis troubleshooting.

The Plug and Play (PnP) Manager

The PnP manager is the primary component involved in supporting the ability of Windows to recognize and adapt to changing hardware configurations. A user doesn't need to understand the intricacies of hardware or manual configuration to install and remove devices. For example, it's the PnP manager that enables a running Windows laptop that is placed on a docking station to automatically detect additional devices located in the docking station and make them available to the user.

Plug and Play support requires cooperation at the hardware, device driver, and operating system levels. Industry standards for the enumeration and identification of devices attached to buses are the foundation of Windows Plug and Play support. For example, the USB standard defines the way that devices on a USB bus identify themselves. With this foundation in place, Windows Plug and Play support provides the following capabilities:

- The PnP manager automatically recognizes installed devices, a process that includes enumerating devices attached to the system during a boot and detecting the addition and removal of devices as the system executes.
- Hardware resource allocation is a role the PnP manager fills by gathering the hardware resource requirements (interrupts, I/O memory, I/O registers, or bus-specific resources) of the devices attached to a system and, in a process called *resource arbitration*, optimally assigning resources so that each device meets the requirements necessary for its operation. Because hardware devices can be added to the system after boot-time resource assignment, the PnP manager must also be able to reassign resources to accommodate the needs of dynamically added devices.

- Loading appropriate drivers is another responsibility of the PnP manager. The PnP manager determines, based on the identification of a device, whether a driver capable of managing the device is installed on the system, and if one is, instructs the I/O manager to load it. If a suitable driver isn't installed, the kernel-mode PnP manager communicates with the user-mode PnP manager to install the device, possibly requesting the user's assistance in locating a suitable set of drivers.
- The PnP manager also implements application and driver mechanisms for the detection of hardware configuration changes. Applications or drivers sometimes require a specific hardware device to function, so Windows includes a means for them to request notification of the presence, addition, or removal of devices.

Level of Plug and Play Support

Windows aims to provide full support for Plug and Play, but the level of support possible depends on the attached devices and installed drivers. If a single device or driver doesn't support Plug and Play, the extent of Plug and Play support for the system can be compromised. In addition, a driver that doesn't support Plug and Play might prevent other devices from being usable by the system. Table 9-2 shows the outcome of various combinations of devices and drivers that can and can't support Plug and Play.

Table 9-2 Device and Driver Plug and Play Capability

	Type of Driver	
Type of Device	**Plug and Play**	**Non–Plug and Play**
Plug and Play	Full Plug and Play	No Plug and Play
Non–Plug and Play	Possible partial Plug and Play	No Plug and Play

A device that isn't Plug and Play–compatible is one that doesn't support automatic detection, such as a legacy ISA sound card. Because the operating system doesn't know where the hardware physically lies, certain operations—such as laptop undocking, sleep, and hibernation—are disallowed. However, if a Plug and Play driver is manually installed for the device, the driver can at least implement PnP manager–directed resource assignment for the device.

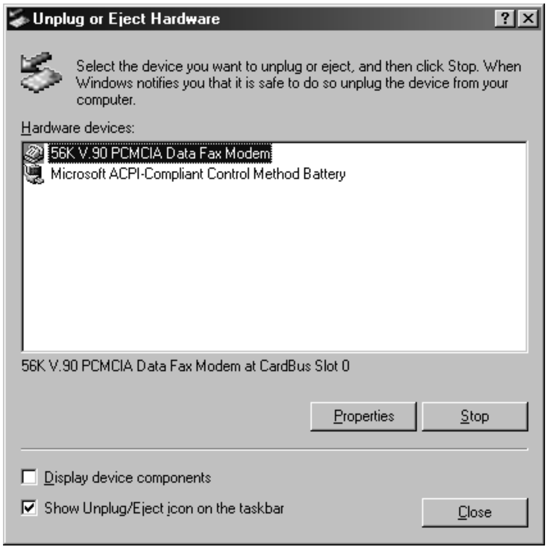
Drivers that aren't Plug and Play–compatible include legacy drivers, such as those that ran on Windows NT 4. Although these drivers might continue to function on later versions of Windows, the PnP manager can't reconfigure the resources assigned to such devices in the event that resource reallocation is necessary to accommodate the needs of a dynamically added device. For example, a device might be able to use I/O memory ranges A and B, and during the boot the PnP manager assigns it a range A. If a device that can use only A is attached to the system later, the PnP manager can't direct the first device's driver to reconfigure itself to use range B. This prevents the second device from obtaining required resources, which results in the device being unavailable for use by the system. Legacy drivers also impair a machine's ability to sleep or hibernate. (See the section "The Power Manager" more details.)

Driver Support for Plug and Play

To support Plug and Play, a driver must implement a Plug and Play dispatch routine, a power management dispatch routine (described in the section "The Power Manager" later in this chapter), and an add-device routine. Bus drivers must support different types of Plug and Play requests than function or filter drivers do, however. For example, when the PnP manager is guiding device enumeration during the system boot (described in detail later in this chapter), it asks bus drivers for a description of the devices that they find on their respective buses. The description includes data that uniquely identifies each device as well as the resource requirements of the devices. The PnP manager takes this information and loads any function or filter drivers that have been installed for the detected devices. It then calls the add-device routine of each driver for every installed device the drivers are responsible for.

Function and filter drivers prepare to begin managing their devices in their add-device routines, but they don't actually communicate with the device hardware. Instead, they wait for the PnP manager to send a *start-device* command for the device to their Plug and Play dispatch routine. Prior to sending the start-device command the PnP manager performs resource arbitration to decide what resources to assign the device. The *start-device* command includes the resource assignment that the PnP manager determines during resource arbitration. When a driver receives a *start-device* command, it can configure its device to use the specified resources. If an application tries to open a device that hasn't finished starting, it receives an error indicating that the device does not exist.

After a device has started, the PnP manager can send the driver additional Plug and Play commands, including ones related to a device's removal from the system or to resource reassignment. For example, when the user invokes the remove/eject device utility, shown in Figure 9-23 (accessible by right-clicking on the PC card icon in the taskbar and selecting Unplug Or Eject Hardware), to tell Windows to eject a PCMCIA card, the PnP manager sends a *query-remove* notification to any applications that have registered for Plug and Play notifications for the device. Applications typically register for notification on their handles, which they close during a query-remove notification. If no applications veto the query-remove request, the PnP manager sends a *query-remove* command to the driver that owns the device being ejected. At that point, the driver has a chance to deny the removal or to ensure that any pending I/O operations involving the device have completed and to begin rejecting further I/O requests aimed at the device. If the driver agrees to the remove request and no open handles to the device remain, the PnP manager next sends a *remove* command to the driver to request that the driver discontinue accessing the device and release any resources the driver has allocated on behalf of the device.



Select the device you want to unplug or eject, and then click Stop. When Windows notifies you that it is safe to do so unplug the device from your computer.

Hardware devices:

56K V.90 PCMCIA Data Fax Modem
Microsoft ACPI-Compliant Control Method Battery

56K V.90 PCMCIA Data Fax Modem at CardBus Slot 0

□ Display device components
☑ Show Unplug/Eject icon on the taskbar

Figure 9-23 PC card remove/eject utility

When the PnP manager needs to reassign a device's resources, it first asks the driver whether it can temporarily suspend further activity on the device by sending the driver a *query-stop* command. The driver either agrees to the request, if doing so wouldn't cause data loss or corruption, or denies the request. As with a *query-remove* command, if the driver agrees to the request, the driver completes pending I/O operations and won't initiate further I/O requests for the device that can't be aborted and subsequently restarted. The driver typically queues new I/O requests so that the resource reshuffling is transparent to applications currently accessing the device. The PnP manager then sends the driver a *stop* command. At that point, the PnP manager can direct the driver to assign different resources to the device and once again send the driver a *start-device* command for the device.

The various Plug and Play commands essentially guide a device through an assortment of operational states, forming a well-defined state-transition table, which is shown in simplified form in Figure 9-24. (Several possible transitions and Plug and Play commands have been omitted for clarity. Also, the state diagram depicted is that implemented by function drivers. Bus drivers implement a more complex state diagram.) A state shown in the figure that we haven't discussed is the one that results from the PnP manager's *surprise-remove* command. This command results when either a user removes a device without warning, as when the user ejects a PCMCIA card without using the remove/eject utility, or the device fails. The *surprise-remove* command tells the driver to immediately cease all interaction with the device because the device is no longer attached to the system and to cancel any pending I/O requests.

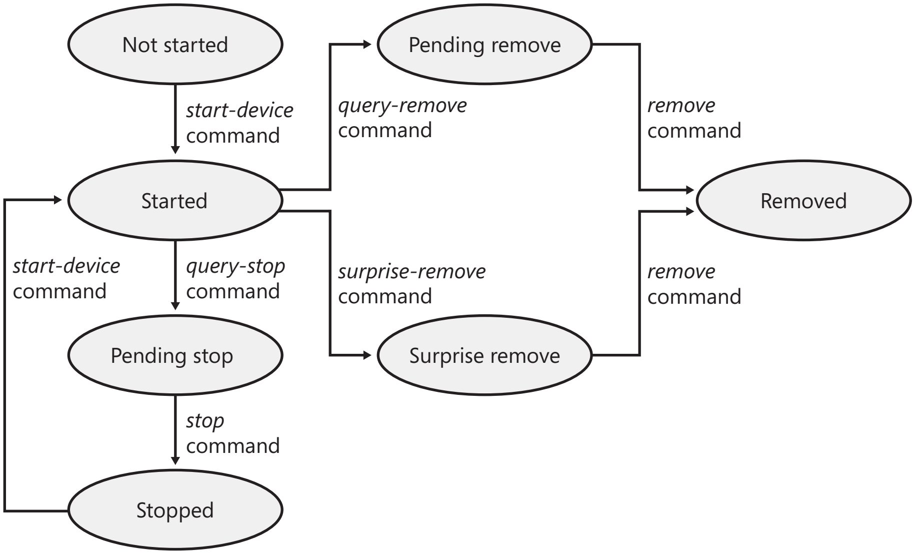


Figure 9-24 Device Plug and Play state transitions

Driver Loading, Initialization, and Installation

Driver loading and initialization on Windows consists of two types of loading: explicit loading and enumeration-based loading. Explicit loading is guided by the HKLM\SYSTEM\CurrentControlSet\Services branch of the registry, as described in the section "Service Applications" in Chapter 4. Enumeration-based loading results when the PnP manager dynamically loads drivers for the devices that a bus driver reports during bus enumeration.

The Start Value

In Chapter 4, we explained that every driver and Windows service has a registry key under the Services branch of the current control set. The key includes values that specify the type of the image (for example, Windows service, driver, and file system), the path to the driver or service's image file, and values that control the driver or service's load ordering. There are two main differences between explicit device driver loading and Windows service loading:

- Only device drivers can specify Start values of boot-start (0) or system-start (1).
- Device drivers can use the Group and Tag values to control the order of loading within a phase of the boot, but unlike services, they can't specify DependOnGroup or DependOnService values.

Chapter 5 describes the phases of the boot process and explains that a driver Start value of 0 means that the operating system loader loads the driver. A Start value of 1 means that the I/O manager loads the driver after the executive subsystems have finished initializing. The I/O manager calls driver initialization routines in the order that the drivers load within a boot phase. Like Windows services, drivers use the Group value in their registry key to specify which group they belong to; the registry value HKLM\SYSTEM\CurrentControlSet\Control\ServiceGroupOrder\List determines the order that groups are loaded within a boot phase.

A driver can further refine its load order by including a Tag value to control its order within a group. The I/O manager sorts the drivers within each group according to the Tag values defined in the drivers' registry keys. Drivers without a tag go to the end of the list in their group. You might assume that the I/O manager initializes drivers with lower-number tags before it initializes drivers with higher-number tags, but such isn't necessarily the case. The registry key HKLM\SYSTEM\CurrentControlSet\Control\GroupOrderList defines tag precedence within a group; with this key, Microsoft and device driver developers can take liberties with redefining the integer number system.

Here are the guidelines by which drivers set their Start value:

- Non–Plug and Play drivers set their Start value to reflect the boot phase they want to load in.
- Drivers, including both Plug and Play and non–Plug and Play drivers, that must be loaded by the boot loader during the system boot specify a Start value of boot-start (0). Examples include system bus drivers and the boot file system driver.
- A driver that isn't required for booting the system and that detects a device that a system bus driver can't enumerate specifies a Start value of system-start (1). An example is the serial port driver, which informs the PnP manager of the presence of standard PC serial ports that were detected by Setup and recorded in the registry.
- A non–Plug and Play driver or file system driver that doesn't have to be present when the system boots specifies a Start value of auto-start (2). An example is the Multiple Universal Naming Convention (UNC) Provider (MUP) driver, which provides support for UNC-based path names to remote resources (for example, (\\REMOTECOMPUTERNAME\SHARE).
- Plug and Play drivers that aren't required to boot the system specify a Start value of demand-start (3). Examples include network adapter drivers.

The only purpose that the Start values for Plug and Play drivers and drivers for enumerable devices have is to ensure that the operating system loader loads the driver—if the driver is required for the system to boot successfully. Beyond that, the PnP manager's device enumeration process, described next, determines the load order for Plug and Play drivers.

Device Enumeration

The PnP manager begins device enumeration with a virtual bus driver called Root, which represents the entire computer system and acts as the bus driver for non–Plug and Play drivers and for the HAL. The HAL acts as a bus driver that enumerates devices directly attached to the motherboard as well as system components such as batteries. Instead of actually enumerating, the HAL relies on the hardware description the Setup process recorded in the registry to detect the primary bus (a PCI bus in most cases) and devices such as batteries and fans.

The primary bus driver enumerates the devices on its bus, possibly finding other buses, for which the PnP manager initializes drivers. Those drivers in turn can detect other devices, including other subsidiary buses. This recursive process of enumeration, driver loading (if the driver isn't already loaded), and further enumeration proceeds until all the devices on the system have been detected and configured.

As the bus drivers report detected devices to the PnP manager, the PnP manager creates an internal tree called the *device tree* that represents the relationships between devices. Nodes in the tree are called *devnodes*, and a devnode contains information about the device objects that represent the device as well as other Plug and Play–related information stored in the devnode by the PnP manager. Figure 9-25 shows an example of a simplified device tree. This system is ACPI-compliant, so an ACPI-compliant HAL serves as the primary bus enumerator. A PCI bus serves as the system's primary bus, which USB, ISA, and SCSI buses are connected to.

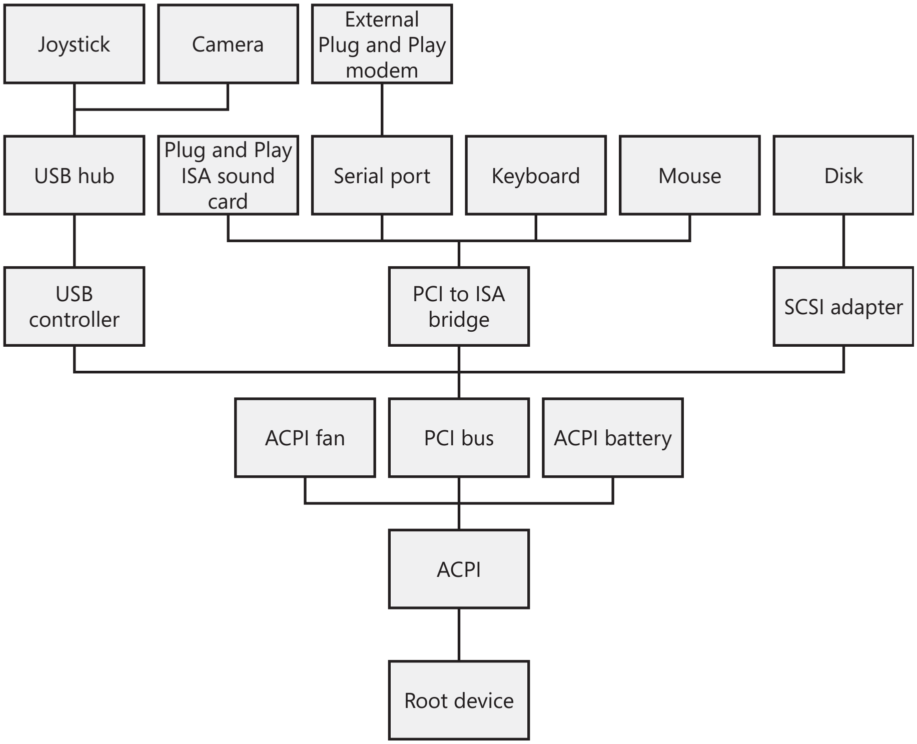
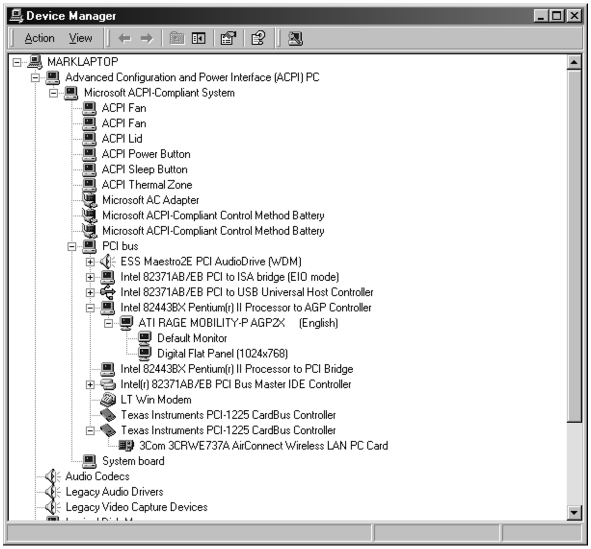


Figure 9-25 Example device tree

The Device Manager utility, which is accessible from the Computer Management snap-in in the Programs/Administrative Tools folder of the Start menu (and also from the Hardware tab of the System utility in Control Panel), shows a simple list of devices present on a system in its default configuration. You can also select the Devices By Connection option from the Device Manager's View menu to see the devices as they relate to the device tree. Figure 9-26 shows an example of the Device Manager's Devices By Connection view.



```
MARKLAPTOP
  Advanced Configuration and Power Interface (ACPI) PC
    Microsoft ACPI-Compliant System
      ACPI Fan
      ACPI Fan
      ACPI Lid
      ACPI Power Button
      ACPI Sleep Button
      ACPI Thermal Zone
      Microsoft AC Adapter
      Microsoft ACPI-Compliant Control Method Battery
      Microsoft ACPI-Compliant Control Method Battery
      PCI bus
        ESS Maestro2E PCI AudioDrive (WDM)
        Intel 82371AB/EB PCI to ISA bridge (EIO mode)
        Intel 82371AB/EB PCI to USB Universal Host Controller
        Intel 82443BX Pentium(r) II Processor to AGP Controller
          ATI RAGE MOBILITY-P AGP2X    (English)
            Default Monitor
            Digital Flat Panel (1024x768)
        Intel 82443BX Pentium(r) II Processor to PCI Bridge
        Intel(r) 82371AB/EB PCI Bus Master IDE Controller
        LT Win Modem
        Texas Instruments PCI-1225 CardBus Controller
        Texas Instruments PCI-1225 CardBus Controller
          3Com 3CRWE737A AirConnect Wireless LAN PC Card
      System board
  Audio Codecs
  Legacy Audio Drivers
  Legacy Video Capture Devices
```

Figure 9-26 Device Manager showing the device tree

Taking device enumeration into account, the load and initialization order of drivers is as follows:

1. The I/O manager invokes the driver entry routine of each boot-start driver. If a boot driver has child devices, the I/O manager enumerates those devices, reporting their presence to the PnP manager. The child devices are configured and started if their drivers are boot-start drivers. If a device has a driver that isn't a boot-start driver, the PnP manager creates a devnode for the device but doesn't start it or load its driver.
2. After the boot-start drivers are initialized, the PnP manager walks the device tree, loading the drivers for devnodes that weren't loaded in step 1 and starting their devices. As each device starts, the PnP manager enumerates its child devices, if it has any, starting those devices' drivers and performing enumeration of their children as required. The PnP manager loads the drivers for detected devices in this step *regardless of the driver's Start value*. (The one exception is if the Start value is set to disabled.) At the end of this step, all Plug and Play devices have their drivers loaded and are started, except devices that aren't enumerable and the children of those devices.
3. The PnP manager loads any drivers with a Start value of system-start that aren't yet loaded. Those drivers detect and report their nonenumerable devices. The PnP manager loads drivers for those devices until all enumerated devices are configured and started.
4. The Service Control Manager loads drivers marked as auto-start.

The device tree serves to guide both the PnP manager and the power manager as they issue Plug and Play and power IRPs to devices. In general, IRPs flow from the top of a devnode to the bottom, and in some cases a driver in one devnode creates new IRPs to send to other devnodes, always moving toward the root. The flow of Plug and Play and power IRPs is further described later in this chapter.

EXPERIMENT: Dumping the Device Tree

A more detailed way to view the device tree than using Device Manager is to use the *!devnode* kernel debugger command. Specifying *0 1* as command options dumps the internal device tree devnode structures, indenting entries to show their hierarchical relationships, as shown here:

```
lkd> !devnode 0 1
Dumping IopRootDeviceNode (= 0x8a4b7ee8)
DevNode 0x8a4b7ee8 for PDO 0x8a4b7020
  InstancePath is “HTREE\ROOT\0"
  State = DeviceNodeStarted (0x308)
  Previous State = DeviceNodeEnumerateCompletion (0x30d)
  DevNode 0x8a4b7a50 for PDO 0x8a4b7b98
    InstancePath is “Root\ACPI_HAL\0000"
    State = DeviceNodeStarted (0x308)
    Previous State = DeviceNodeEnumerateCompletion (0x30d)
    DevNode 0x8a4af448 for PDO 0x8a4eb2c8
      InstancePath is “ACPI_HAL\PNP0C08\0"
      ServiceName is “ACPI"
      State = DeviceNodeStarted (0x308)
      Previous State = DeviceNodeEnumerateCompletion (0x30d)
      DevNode 0x8a4af198 for PDO 0x8a4b1350
        InstancePath is “ACPI\GenuineIntel_-_x86_Family_6_Model_9\_0"
        ServiceName is “gv3"
        State = DeviceNodeStarted (0x308)
        Previous State = DeviceNodeEnumerateCompletion (0x30d)
      DevNode 0x8a4e8008 for PDO 0x8a4a8950
        InstancePath is “ACPI\ThermalZone\THM_"
        State = DeviceNodeStarted (0x308)
        Previous State = DeviceNodeEnumerateCompletion (0x30d)
      DevNode 0x8a4e82b8 for PDO 0x8a4eb640
        InstancePath is “ACPI\ACPI0003\2&daba3ff&0"
        ServiceName is “CmBatt"

§
```

Information shown for each devnode includes the InstancePath, which is the name of the device’s enumeration registry key stored under HKLM\SYSTEM\CurrentControlSet\Enum, and the ServiceName, which corresponds to the device’s driver registry key under HKLM\SYSTEM\CurrentControlSet\Services. To see the resources, such as interrupts, ports, and memory assigned to each devnode, specify *0 3* as the command options for the *!devnode* command.

A record of all the devices detected since the system was installed is recorded under the HKLM\SYSTEM\CurrentControlSet\Enum registry key. Subkeys are in the form <Enumerator>\<Device ID>\<Instance ID>, where the enumerator is a bus driver, the device ID is a unique identifier for a type of device, and the instance ID uniquely identifies different instances of the same hardware.

Devnodes

Figure 9-27 shows that a devnode is made up of at least two, and sometimes more, device objects:

- A *physical device object* (PDO) that the PnP manager instructs a bus driver to create when the bus driver reports the presence of a device on its bus during enumeration. The PDO represents the physical interface to the device.

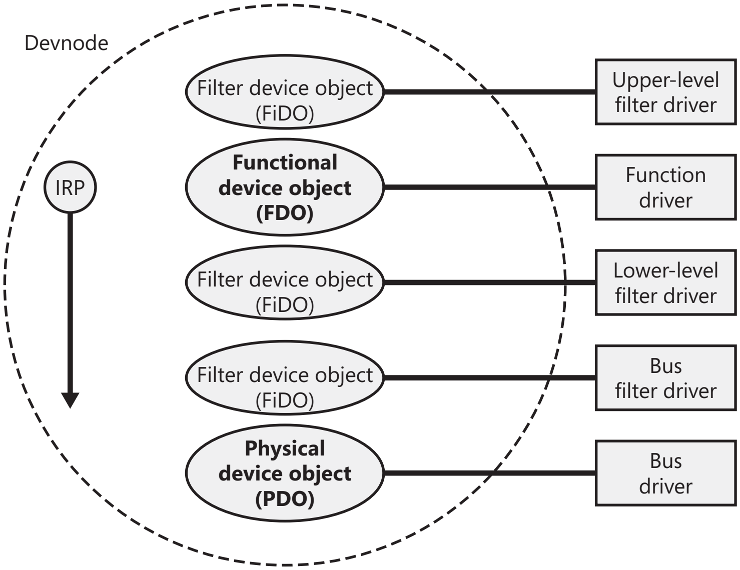

Devnode
IRP
Filter device object (FiDO)
Functional device object (FDO)
Filter device object (FiDO)
Filter device object (FiDO)
Physical device object (PDO)
Upper-level filter driver
Function driver
Lower-level filter driver
Bus filter driver
Bus driver

Figure 9-27 Devnode internals

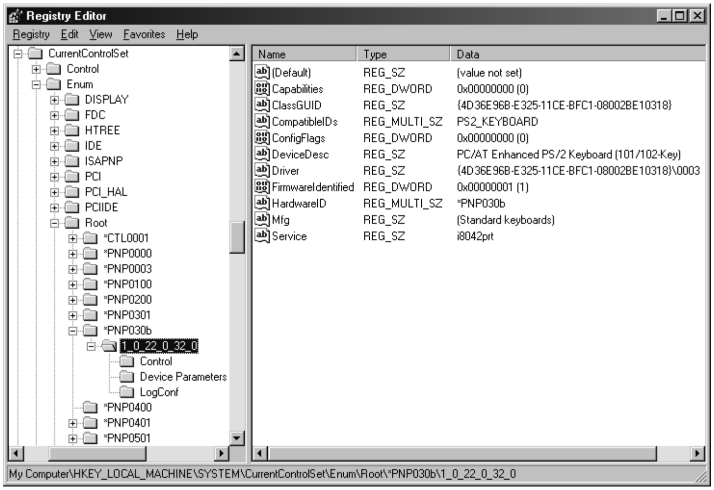
- One or more optional *filter device objects* (FiDOs) that layer between the PDO and the FDO (described next), and that are created by *bus filter drivers*.
- One or more optional FiDOs that layer between the PDO and the FDO (and that layer above any FiDOs created by bus filter drivers) that are created by *lower-level filter drivers*.
- A *functional device object* (FDO) that is created by the driver, which is called a *function driver*, that the PnP manager loads to manage a detected device. An FDO represents the logical interface to a device. A function driver can also act as a bus driver if devices are attached to the device represented by the FDO. The function driver often creates an interface (described earlier) to the FDO's corresponding PDO so that applications and other drivers can open the device and interact with it. Sometimes function drivers are divided into a separate class/port driver and miniport driver that work together to manage I/O for the FDO.
- One or more optional FiDOs that layer above the FDO and that are created by *upper-level filter drivers*.

Devnodes are built from the bottom up and rely on the I/O manager's layering functionality, so IRPs flow from the top of a devnode toward the bottom. However, any level in the devnode can choose to complete an IRP. For example, the function driver can handle a read request without passing the IRP to the bus driver. Only when the function driver requires the help of a bus driver to perform bus-specific processing does the IRP flow all the way to the bottom and then into the devnode containing the bus driver.

Devnode Driver Loading

So far, we've avoided answering two important questions: "How does the PnP manager determine what function driver to load for a particular device?" and "How do filter drivers register their presence so that they are loaded at appropriate times in the creation of a devnode?"

The answer to both these questions lies in the registry. When a bus driver performs device enumeration, it reports device identifiers for the devices it detects back to the PnP manager. The identifiers are bus-specific; for a USB bus, an identifier consists of a *vendor ID* (VID) for the hardware vendor that made the device and a *product ID* (PID) that the vendor assigned to the device. (See the DDK for more information on device ID formats.) Together these IDs form what Plug and Play calls a *device ID*. The PnP manager also queries the bus driver for an *instance ID* to help it distinguish different instances of the same hardware. The instance ID can describe either a bus-relative location (for example, the USB port) or a globally unique descriptor (for example, a serial number). The device ID and instance ID are combined to form a *device instance ID* (DIID), which the PnP manager uses to locate the device's key in the enumeration branch of the registry (HKLM\SYSTEM\CurrentControlSet\Enum). Figure 9-28 presents an example of a keyboard's enumeration subkey. The device's key contains descriptive data and includes values named Service and ClassGUID (which are obtained from a driver's INF file) that help the PnP manager locate the device's drivers.



```
CurrentControlSet
  Control
  Enum
    DISPLAY
    FDC
    HTREE
    IDE
    ISAPNP
    PCI
    PCI_HAL
    PCIIDE
    Root
      *CTL0001
      *PNP0000
      *PNP0003
      *PNP0100
      *PNP0200
      *PNP0301
      *PNP030b
        1_0_22_0_32_0
          Control
          Device Parameters
          LogConf
      *PNP0400
      *PNP0401
      *PNP0501
```

Name	Type	Data
(Default)	REG_SZ	(value not set)
Capabilities	REG_DWORD	0x00000000 (0)
ClassGUID	REG_SZ	{4D36E96B-E325-11CE-BFC1-08002BE10318}
CompatibleIDs	REG_MULTI_SZ	PS2_KEYBOARD
ConfigFlags	REG_DWORD	0x00000000 (0)
DeviceDesc	REG_SZ	PC/AT Enhanced PS/2 Keyboard (101/102-Key)
Driver	REG_SZ	{4D36E96B-E325-11CE-BFC1-08002BE10318}\0003
FirmwareIdentified	REG_DWORD	0x00000001 (1)
HardwareID	REG_MULTI_SZ	*PNP030b
Mfg	REG_SZ	(Standard keyboards)
Service	REG_SZ	i8042prt

My Computer\HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Enum\Root\*PNP030b\1_0_22_0_32_0

Figure 9-28 Keyboard enumeration key

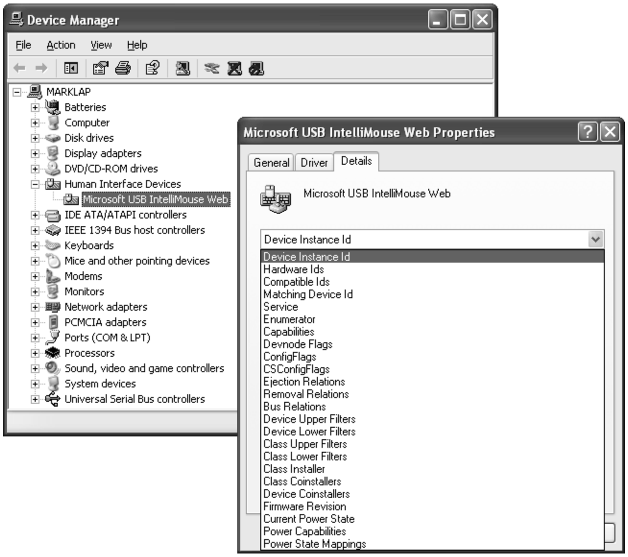
EXPERIMENT: Viewing Detailed Devnode Information in Device Manager

By default, the Device Manager applet that you can access from the Hardware tab of the System control panel application doesn't show detailed information about a device node. However, in Windows XP and Windows Server 2003 you can enable a tab called *Details* by creating and setting the devmgr_show_details environment variable to a value of 1. The tab allows you to view an assortment of fields including the devnode's device instance ID, hardware ID, service name, filters, and power capabilities.

The simplest way to launch the Device Manager with the Details tab is to open a command prompt and execute the following:

```
C:\>set devmgr_show_details=1
C:\>devmgmt.msc
```

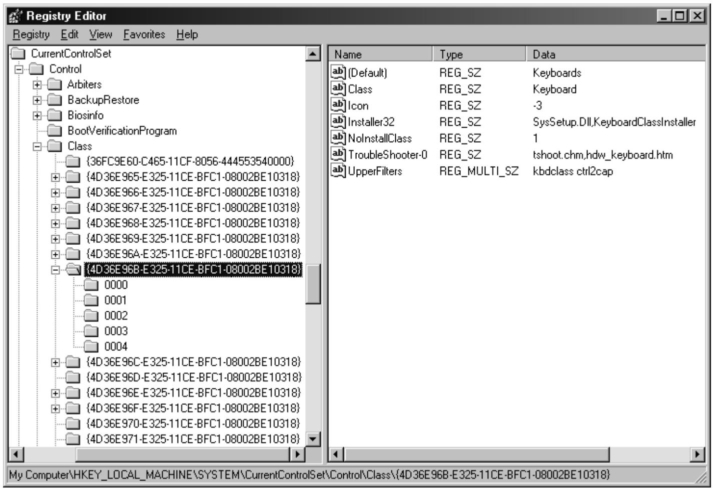
This screen shot shows the selection combo box of the Details tab expanded to reveal the types of information you can access:



Using the ClassGUID value, the PnP manager locates the device's class key under HKLM\SYSTEM\CurrentControlSet\Control\Class. The keyboard class key is shown in Figure 9-29. The enumeration key and class key supply the PnP manager the information it needs to load the drivers necessary for the device's devnode. Drivers are loaded in the following order:

1. Any lower-level filter drivers specified in the LowerFilters value of the device's enumeration key.
2. Any lower-level filter drivers specified in the LowerFilters value of the device's class key.
3. The function driver specified by the Service value in the device's enumeration key. This value is interpreted as the driver's key under HKLM\SYSTEM\CurrentControlSet\Services.
4. Any upper-level filter drivers specified in the UpperFilters value of the device's enumeration key.
5. Any upper-level filter drivers specified in the UpperFilters value of the device's class key.

In all cases, drivers are referenced by the name of their key under HKLM\SYSTEM\CurrentControlSet\Services.



```
CurrentControlSet
  Control
    Arbiters
    BackupRestore
    Biosinfo
    BootVerificationProgram
    Class
      {36FC9E60-C465-11CF-8056-444553540000}
      {4D36E965-E325-11CE-BFC1-08002BE10318}
      {4D36E966-E325-11CE-BFC1-08002BE10318}
      {4D36E967-E325-11CE-BFC1-08002BE10318}
      {4D36E968-E325-11CE-BFC1-08002BE10318}
      {4D36E969-E325-11CE-BFC1-08002BE10318}
      {4D36E96A-E325-11CE-BFC1-08002BE10318}
      {4D36E96B-E325-11CE-BFC1-08002BE10318}
        0000
        0001
        0002
        0003
        0004
      {4D36E96C-E325-11CE-BFC1-08002BE10318}
      {4D36E96D-E325-11CE-BFC1-08002BE10318}
      {4D36E96E-E325-11CE-BFC1-08002BE10318}
      {4D36E96F-E325-11CE-BFC1-08002BE10318}
      {4D36E970-E325-11CE-BFC1-08002BE10318}
      {4D36E971-E325-11CE-BFC1-08002BE10318}
```

Name	Type	Data
(Default)	REG_SZ	Keyboards
Class	REG_SZ	Keyboard
Icon	REG_SZ	-3
Installer32	REG_SZ	SysSetup.Dll,KeyboardClassInstaller
NoInstallClass	REG_SZ	1
TroubleShooter-0	REG_SZ	tshoot.chm,hdw_keyboard.htm
UpperFilters	REG_MULTI_SZ	kbdclass ctrl2cap

My Computer\HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Class\{4D36E96B-E325-11CE-BFC1-08002BE10318}

Figure 9-29 Keyboard class key

> **Note** The Windows DDK refers to a device's enumeration key as its *hardware key* and to the class key as the *software key.*

The keyboard device shown in Figure 9-28 and Figure 9-29 has no lower filter drivers. The function driver is the i8042prt driver, and there are two upper filter drivers specified in the keyboard's class key: kbdclass and ctrl2cap.

Driver Installation

If the PnP manager encounters a device for which no driver is installed, it relies on the user-mode PnP manager to guide the installation process. If the device is detected during the system boot, a devnode is defined for the device but the loading process is postponed until the user-mode PnP manager starts. (The user-mode PnP manager is implemented in \Windows\System32\Umpnpmgr.dll and runs as a service in the Services.exe process.)

The components involved in a driver's installation are shown in Figure 9-30. Shaded objects in the figure correspond to components generally supplied by the system, whereas objects that aren't shaded are included in a driver's installation files. First, a bus driver informs the PnP manager of a device it enumerates using a DIID (1). The PnP manager checks the registry for the presence of a corresponding function driver, and when it doesn't find one, it informs the user-mode PnP manager (2) of the new device by its DIID. The user-mode PnP manager first tries to perform an automatic install without user intervention. If the installation process involves the posting of dialog boxes that require user interaction and the currently logged-on user has administrator privileges, (3) the user-mode PnP manager launches the Rundll32.exe application (the same application that hosts Control Panel utilities) to execute the Hardware Installation Wizard (\Windows\System32\Newdev.dll). If the currently logged-on user doesn't have administrator privileges (or if no user is logged on) and the installation of the device requires user interaction, the user-mode PnP manager defers the installation until a privileged user logs on. The Hardware Installation Wizard uses Setup and CfgMgr (Configuration Manager) API functions to locate INF files that correspond to drivers that are compatible with the detected device. This process might involve having the user insert installation media containing a vendor's INF files, or the wizard might locate a suitable INF file in the .cab (Cabinet) file \Windows\Driver Cache\i386\Driver.cab that contains drivers that ship with Windows.

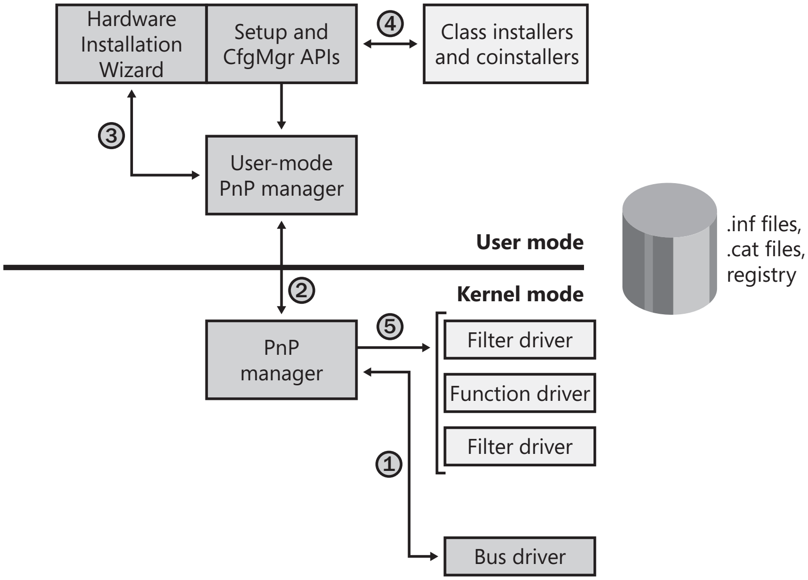


Figure 9-30 Driver installation components

To find drivers for the new device, the installation process gets a list of *hardware IDs* and *compatible IDs* from the bus driver. These IDs describe all the various ways the hardware might be identified in a driver installation file (.inf). The lists are ordered so that the most specific description of the hardware is listed first. If matches are found in multiple INFs, more precise matches are preferred over less precise matches, digitally signed INFs are preferred over unsigned ones, and newer signed INFs are preferred over older signed ones. If a match is found based on a compatible ID, the Hardware Installation Wizard can choose to prompt for media in case a more up-to-date driver came with the hardware.

The INF file locates the function driver's files and contains commands that fill in the driver's enumeration and class keys, and the INF file might direct the Hardware Installation Wizard to (4) launch class or device co-installer DLLs that perform class or device-specific installation steps, such as displaying configuration dialog boxes that let the user specify settings for a device.

EXPERIMENT: Looking at a Driver's INF File

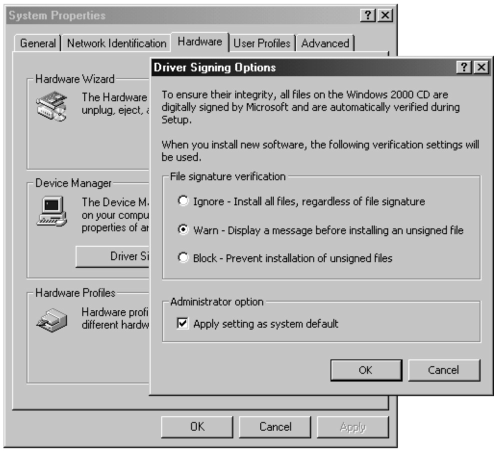
When a driver or other software that has an INF file is installed, the system copies its INF file to the \Windows\Inf directory. One file that will always be there is Keyboard.inf because it's the INF file for the keyboard class driver. View its contents by opening it in Notepad and you should see something like this:

```
; Copyright (c) 1993-1996, Microsoft Corporation

[version]
signature="$Windows NT$"
Class=Keyboard
ClassGUID={4D36E96B-E325-11CE-BFC1-08002BE10318}
Provider=%MS%
LayoutFile=layout.inf
DriverVer=07/01/2001,5.1.2600.1106

[ClassInstall32.NT]
AddReg=keyboard_class_addreg
…
```

If you search the file for ".sys", you'll come across the entry that directs the user-mode PnP manager to install the i8042prt.sys and kbdclass.sys drivers:

```
…
[STANDARD_CopyFiles]
i8042prt.sys
kbdclass.sys
…
```

Before actually installing a driver, the user-mode PnP manager checks the system's driver-signing policy. The policy is stored in the registry key HKLM\SOFTWARE\Microsoft\Driver Signing\Policy if the administrator has designated a systemwide policy and in the key HKCU\Software\Microsoft\Driver Signing\Policy if there are only per-user policies. The policy is configurable using the Driver Signing Options dialog box, accessed from the Hardware tab in the System Control Panel utility, which is shown in Figure 9-31. If the settings specify that the system should block or warn of the installation of unsigned drivers, the user-mode PnP manager checks the driver's INF file for an entry that locates a catalog (a file that ends with the .cat extension) containing the driver's digital signature.



Driver Signing Options

To ensure their integrity, all files on the Windows 2000 CD are digitally signed by Microsoft and are automatically verified during Setup.

When you install new software, the following verification settings will be used.

File signature verification

○ Ignore - Install all files, regardless of file signature

● Warn - Display a message before installing an unsigned file

○ Block - Prevent installation of unsigned files

Administrator option

☑ Apply setting as system default

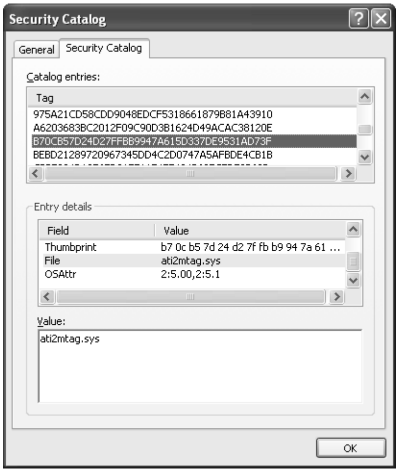
Figure 9-31 Driver-signing policy options

Microsoft's WHQL tests the drivers included with Windows and those submitted by hardware vendors. When a driver passes the WHQL tests, it is "signed" by Microsoft. This means that WHQL obtains a *hash*, or unique value representing the driver's files, including its image file, and then cryptographically signs it with Microsoft's private driver-signing key. The signed hash is stored in a catalog file and included on the Windows installation media or returned to the vendor that submitted the driver for inclusion with its driver.

EXPERIMENT: Viewing Catalog Files

When you install a component such as a driver that includes a catalog file, Windows copies the catalog file to a directory under \Windows\System32\Catroot. Navigate to that directory in Explorer and you find the subdirectory that contains .cat files. Nt5.cat and Nt5inf.cat store the signatures for Windows system files, for example.

If you open one of the catalog files, a dialog box appears with two pages. The page labeled General shows information about the signature on the catalog file, and the Security Catalog has the hashes of the components that are signed with the catalog file. This screen shot of a catalog file for ATI video drivers shows the hash for the video adapter's miniport drivers. Other hashes in the catalog are associated with the various support DLLs that ship with the driver:



Catalog entries:

Tag
975A21CD58CDD9048EDCF5318661879B81A43910
A6203683BC2012F09C90D3B1624D49ACAC38120E
B70CB57D24D27FFBB9947A615D337DE9531AD73F
BEBD21289720967345DD4C2D0747A5AFBDE4CB1B

Entry details

Field	Value
Thumbprint	b7 0c b5 7d 24 d2 7f fb b9 94 7a 61 ...
File	ati2mtag.sys
OSAttr	2:5.00,2:5.1

Value: ati2mtag.sys

As it's installing a driver, the user-mode PnP manager extracts the driver's signature from its catalog file, decrypts the signature using the public half of Microsoft's driver-signing private/public key pair, and compares the resulting hash with a hash of the driver file it's about to install. If the hashes match, the driver is verified as having passed WHQL testing. If a driver fails the signature verification, the user-mode PnP manager acts according to the settings of the system driver-signing policy, either failing the installation attempt, warning the user that the driver is unsigned, or silently installing the driver.

Note Drivers installed using setup programs that manually configure the registry and copy driver files to a system and driver files that are dynamically loaded by applications aren't checked for signatures. Only drivers installed using INF files are validated against the system's driver-signing policy.

After a driver is installed, the kernel-mode PnP manager (step 5 in Figure 9-30) starts the driver and calls its add-device routine to inform the driver of the presence of the device it was loaded for. The construction of the devnode then continues as described earlier.

Note In Windows XP and Windows Server 2003, the user-mode PnP manager also checks to see whether the driver it's about to install is on the *protected driver list* maintained by Windows Update and, if so, blocks the installation with a warning to the user. Drivers that are known to have incompatibilities or bugs are added to the list and blocked from installation. You can find out more at *www.microsoft.com/whdc/winlogo/drvsign/drv_protect.mspx*.

The Power Manager

Just as Windows Plug and Play features require support from a system's hardware, its power-management capabilities require hardware that complies with the Advanced Configuration and Power Interface (ACPI) specification (available at *www.teleport.com/~acpi/spec.htm*). As a result of this requirement, the computer's BIOS (Basic Input Output System), the code that runs when the computer turns on, must also conform to the ACPI standard. Most x86 computers manufactured since the end of 1998 are ACPI compliant.

> **Note** Some computers, especially ones more than few years old, don't comply with the ACPI standard. Instead, they often conform to the older Advanced Power Management (APM) standard, which mandates fewer power-management capabilities than ACPI. Windows provides limited power management for APM systems, but we won't go into the details of that topic here. In this book, we focus on the behavior of Windows on ACPI computers.

The ACPI standard defines various power levels for a system and for devices. The six system power states are described in Table 9-3. They are referred to as S0 (*fully on* or *working*) through S5 (*fully off*). Each state has the following characteristics:

- **Power consumption** The amount of power the computer consumes
- **Software resumption** The software state from which the computer resumes when moving to a "more on" state
- **Hardware latency** The length of time it takes to return the computer to the fully on state

Table 9-3 System Power-State Definitions

State	Power Consumption	Software Resumption	Hardware Latency
S0 (fully on)	Maximum	Not applicable	None
S1 (sleeping)	Less than S0, more than S2	System resumes where it left off (returns to S0)	Less than 2 seconds
S2 (sleeping)	Less than S1, more than S3	System resumes where it left off (returns to S0)	2 or more seconds
S3 (sleeping)	Less than S2; processor is off	System resumes where it left off (returns to S0)	Same as S2
S4 (hibernating)	Trickle current to power button and wake circuitry	System restarts from saved hibernate file and resumes where it left off prior to hibernation (returns to S0)	Long and undefined
S5 (fully off)	Trickle current to power button	System boot	Long and undefined

States S1 through S4 are sleeping states, in which the computer appears to be off because of reduced power consumption. However, the computer retains enough information, either in memory or on disk, to move to S0. For states S1 through S3, enough power is required to preserve the contents of the computer's memory so that when the transition is made to S0 (when the user or a device wakes up the computer), the power manager continues executing where it left off before the suspend. When the system moves to S4, the power manager saves the compressed contents of memory to a hibernation file named Hiberfil.sys, which is large enough to hold the uncompressed contents of memory, in the root directory of the system volume. (Compression is used to minimize disk I/O and to improve hibernation and resume-from-hibernation performance.) After it finishes saving memory, the power manager shuts off the computer. When a user subsequently turns on the computer, a normal boot process occurs, except that Ntldr checks for and detects a valid memory image stored in the hibernation file. If the hibernation file contains saved system state, Ntldr reads the contents of the file into memory, and then resumes execution at the point in memory that is recorded in the hibernation file.

The computer never directly transitions between states S1 and S4; instead, it must move to state S0 first. As illustrated in Figure 9-32, when the system is moving from any of states S1 through S5 to state S0, it's said to be *waking*, and when it's transitioning from state S0 to any of states S1 through S5, it's said to be *sleeping*.

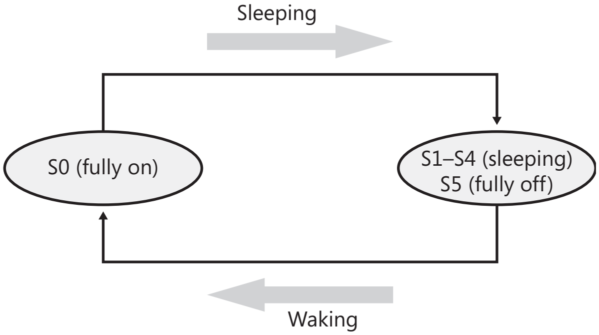


Figure 9-32 System power-state transitions

Although the system can be in one of six power states, ACPI defines devices as being in one of four power states, D0 through D3. State D0 is *fully on*, and state D3 is *fully off*. The ACPI standard leaves it to individual drivers and devices to define the meanings of states D1 and D2, except that state D1 must consume an amount of power less than or equal to that consumed in state D0, and when the device is in state D2, it must consume power less than or equal to that consumed in D1. Microsoft, in conjunction with the major hardware OEMs, has defined a series of power management reference specifications (available on the Microsoft Web site at *www.microsoft.com/whdc/resources/respec/specs/pmref*) that specify the device power states that are required for all devices in a particular class (for the major device classes: display, network, SCSI, and so on). For some devices, there's no intermediate power state between fully on and fully off, which results in these states being undefined.

Power Manager Operation

Power management policy in Windows is split between the power manager and the individual device drivers. The power manager is the owner of the system power policy. This ownership means that the power manager decides which system power state is appropriate at any given point, and when a sleep, hibernation, or shutdown is required, the power manager instructs the power-capable devices in the system to perform appropriate system power-state transitions. The power manager decides when a system power-state transition is necessary by considering a number of factors:

- System activity level
- System battery level
- Shutdown, hibernate, or sleep requests from applications
- User actions, such as pressing the power button
- Control Panel power settings

When the PnP manager performs device enumeration, part of the information it receives about a device is its power-management capabilities. A driver reports whether or not its devices support device states D1 and D2 and, optionally, the latencies, or times required, to move from states D1 through D3 to D0. To help the power manager determine when to make system power-state transitions, bus drivers also return a table that implements a mapping between each of the system power states (S0 through S5) and the device power states that a device supports. The table lists the lowest possible device power state for each system state and directly reflects the state of various power planes when the machine sleeps or hibernates. For example, a bus that supports all four device power states might return the mapping table shown in Table 9-4. Most device drivers turn their devices completely off (D3) when leaving S0 to minimize power consumption when the machine isn't in use. Some devices, however, such as network adapter cards, support the ability to wake up the system from a sleeping state. This ability, along with the lowest device power state in which the capability is present, is also reported during device enumeration.

Table 9-4 Example System-to-Device Power Mappings

System Power State	Device Power State
S0 (fully on)	D0 (fully on)
S1 (sleeping)	D2
S2 (sleeping)	D2
S3 (sleeping)	D2
S4 (hibernating)	D3 (fully off)
S5 (fully off)	D3 (fully off)

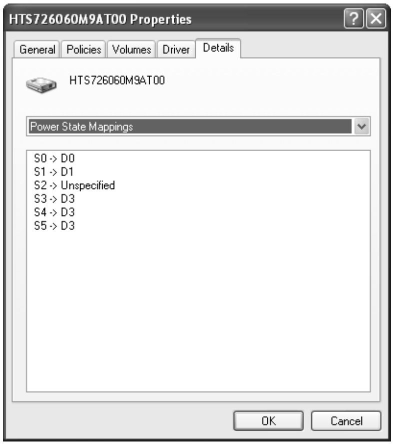
Driver Power Operation

When the power manager decides to make a transition between system power states, it sends power commands to a driver's power dispatch routine. More than one driver can be responsible for managing a device, but only one of the drivers is designated as the device power-policy owner. This driver determines, based on the system state, a device's power state. For example, if the system transitions between state S0 and S1, a driver might decide to move a device's power state from D0 to D1. Instead of directly informing the other drivers that share the management of the device of its decision, the device power-policy owner asks the power manager, via the *PoRequestPowerIrp* function, to tell the other drivers by issuing a device power command to their power dispatch routines. This behavior allows the power manager to control the number of power commands that are active on a system at any given time. For example, some devices in the system might require a significant amount of current to power up. The power manager ensures that such devices aren't powered up simultaneously.

EXPERIMENT: Viewing a Driver's Power Mappings

In Windows XP and Windows Server 2003, you can see a driver's system power state to driver power state mappings with Device Manager. Open the Properties dialog box for a device, and choose the Power State Mappings entry in the drop-down list of the Details tab to see the mappings. (By default, Device Manager doesn't show the Details tab. See the experiment "Viewing Detailed Devnode Information in Device Manager" earlier in the chapter for instructions on enabling it.)

The mappings for this disk driver show that besides fully on (D0) and fully off (D3), it supports an intermediate state, D1, for S1. This likely represents the disk spin-down power state.



HTS726060M9AT00

Power State Mappings

```
S0 -> D0
S1 -> D1
S2 -> Unspecified
S3 -> D3
S4 -> D3
S5 -> D3
```

Many power commands have corresponding query commands. For example, when the system is moving to a sleep state, the power manager will first ask the devices on the system whether the transition is acceptable. A device that is busy performing time-critical operations or interacting with device hardware might reject the command, which results in the system maintaining its current system power-state setting.

EXPERIMENT: Viewing the System Power Capabilities and Policy

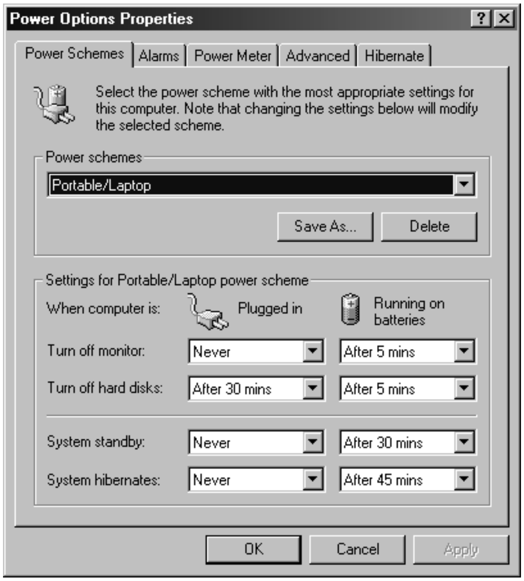
You can view a computer's system power capabilities by using the *!pocaps* kernel debugger command. Here's the output of the command when run on an ACPI-compliant laptop running Windows Professional:

```
kd> !pocaps
PopCapabilities @ 0x8046adc0
  Misc Supported Features:  PwrButton SlpButton Lid S1 S3 S4 S5
                            HiberFile FullWake
  Processor Features:       Thermal Throttle (MinThrottle = 03, Scale = 08)
  Disk Features:            SpinDown
  Battery Features:         BatteriesPresent
    Battery 0 - Capacity: 00000000  Granularity: 00000000
    Battery 1 - Capacity: 00000000  Granularity: 00000000
    Battery 2 - Capacity: 00000000  Granularity: 00000000
  Wake Caps
    Ac OnLine Wake:         Sx
    Soft Lid Wake:          Sx
    RTC Wake:               S3
    Min Device Wake:        Sx
    Default Wake:           Sx
```

The Misc Supported Features line reports that, in addition to S0 (fully on), the system supports system power states S1, S3, S4, and S5 (it doesn't implement S2) and has a valid hibernation file to which it can save system memory when it hibernates (state S4).

The Power Options Properties dialog box, shown on the following page (available by selecting Power Options in Control Panel), lets you configure various aspects of the system's power policy. The exact properties you can configure depends on the system's power capabilities, which we just examined.

Windows Professional and Home systems on an ACPI-compliant laptop (such as the system on which we captured the following screen shot) generally provides the most power-management features. On such systems, you can set the idle detection timeouts that control when the system turns off the monitor, spins down hard disks, goes to standby mode (moves to system power state S1), and hibernates (moves the system to power state S4). In addition, the Advanced tab in Power Options lets you specify the power-related behavior of the system when you press the power or sleep buttons or close a laptop's lid.



Select the power scheme with the most appropriate settings for this computer. Note that changing the settings below will modify the selected scheme.

Power schemes: Portable/Laptop

Settings for Portable/Laptop power scheme

When computer is:	Plugged in	Running on batteries
Turn off monitor:	Never	After 5 mins
Turn off hard disks:	After 30 mins	After 5 mins
System standby:	Never	After 30 mins
System hibernates:	Never	After 45 mins

The settings you configure in Power Options directly affect values in the system's power policy, which you can display with the *!popolicy* debugger command. Here's the output of the command on the same system:

```
kd> !popolicy
SYSTEM_POWER_POLICY (R.1) @ 0x80469180
  PowerButton:         Off  Flags: 00000003   Event: 00000000   Query UI
  SleepButton:       Sleep  Flags: 00000003   Event: 00000000   Query UI
  LidClose:      Hibernate  Flags: 00000001   Event: 00000000   Query
  Idle:               None  Flags: 00000001   Event: 00000000   Query
  OverThrottled:     Sleep  Flags: c0000004   Event: 00000000   Override
                     NoWakes Critical
  IdleTimeout:    00000000  IdleSensitivity:        32
  MinSleep:             S1  MaxSleep:               S3
  LidOpenWake:          S0  FastSleep:              S1
  WinLogonFlags:  00000000  S4Timeout:        00000000
  VideoTimeout:   00000000  VideoDim:               6e
  SpinTimeout:    00000708  OptForPower:            01
  FanTolerance:         64  ForcedThrottle:         64
  MinThrottle:          19
```

The first lines of the display correspond to the button behaviors specified on the Advanced tab of Power Options, and on this system the power button is interpreted as an off switch, the sleep button moves the system to a sleep state, and the closing of the laptop lid causes the system to hibernate.

The timeout values shown at the end of the output are expressed in seconds and displayed in hexadecimal notation. The values reported here directly correspond to the settings you can see configured in the Power Options screen shot. (The laptop is plugged in.) For example, the video timeout is 0, meaning the monitor never turns off, and the hard disk spin-down timeout is 0x708, which corresponds to 1800 seconds, or 30 minutes.

Driver Control of Device Power

Besides responding to power manager commands related to system power-state transitions, a driver can unilaterally control the device power state of its devices. In some cases, a driver might want to reduce the power consumption of a device it controls when the device is left inactive for a period of time. Examples include monitors that support a dimmed mode and disks that support spin-down. A driver can either detect an idle device itself or use facilities provided by the power manager. If the device uses the power manager, it registers the device with the power manager by calling the *PoRegisterDeviceForIdleDetection* function. This function informs the power manager of the timeout values to use to detect a device as idle and of the device power state that the power manager should apply when it detects the device as being idle. The driver specifies two timeouts: one to use when the user has configured the computer to conserve energy and the other to use when the user has configured the computer for optimum performance. After calling *PoRegisterDeviceForIdleDetection*, the driver must inform the power manager, by calling the *PoSetDeviceBusy* function, whenever the device is active.

Conclusion

The I/O system defines the model of I/O processing on Windows and performs functions that are common to or required by more than one driver. Its chief responsibility is to create IRPs representing I/O requests and to shepherd the packets through various drivers, returning results to the caller when an I/O is complete. The I/O manager locates various drivers and devices by using I/O system objects, including driver and device objects. Internally, the Windows I/O system operates asynchronously to achieve high performance and provides both synchronous and asynchronous I/O capabilities to user-mode applications.

Device drivers include not only traditional hardware device drivers but also file system, network, and layered filter drivers. All drivers have a common structure and communicate with one another and the I/O manager by using common mechanisms. The I/O system interfaces allow drivers to be written in a high-level language to lessen development time and to enhance their portability. Because drivers present a common structure to the operating system, they can be layered one on top of another to achieve modularity and reduce duplication between drivers. Also, all Windows device drivers should be designed to work correctly on multiprocessor systems.

Finally, the role of the PnP manager is to work with device drivers to dynamically detect hardware devices and to build an internal device tree that guides hardware device enumeration and driver installation. The power manager works with device drivers to move devices into low-power states when applicable to conserve energy and prolong battery life.

The next four chapters cover additional topics related to the I/O system: storage management, file systems (including details on the NTFS file system), the cache manager, and networking.

Chapter 10

Storage Management

Storage management defines the way that an operating system interfaces with nonvolatile storage devices and media. The term *storage* encompasses many different devices, including tape drives, optical media, CD jukeboxes, floppy diskettes, local hard disks, and storage area networks (SANs). Microsoft Windows provides specialized support for each of these classes of storage media. Because our focus in this book is on the kernel components of Windows, in this chapter we'll concentrate on just the fundamentals of the hard-disk storage subsystem in Windows. Significant portions of the Windows support for removable media and remote storage (offline archiving) are implemented in user mode.

In this chapter, we'll examine how kernel-mode device drivers interface file system drivers to disk media, discuss how disks are partitioned, describe the way volume managers abstract and manage volumes, and present the implementation of multipartition disk-management features in Windows, including replicating and dividing file system data across physical disks for reliability and for performance enhancement. We conclude by describing how file system drivers mount volumes they are responsible for managing.

Storage Terminology

To fully understand the rest of this chapter, you need to be familiar with some basic terminology:

- *Disks* are a physical storage device such as a hard disk, a 3.5-inch floppy disk, or a CD-ROM.
- A disk is divided into *sectors*, addressable blocks of fixed size. Sector sizes are determined by hardware. Most hard disk sectors are 512 bytes, and CD-ROM sectors are typically 2048 bytes.
- *Partitions* are collections of contiguous sectors on a disk. A partition table or other disk-management database stores a partition's starting sector, size, and other characteristics and is located on the same disk as the partition.
- *Simple volumes* are objects that represent sectors from a single partition that file system drivers manage as a single unit.
- *Multipartition volumes* are objects that represent sectors from multiple partitions and that file system drivers manage as a single unit. Multipartition volumes offer performance, reliability, and sizing features that simple volumes do not.

Disk Drivers

The device drivers involved in managing a particular storage device are collectively known as a storage stack. Figure 10-1 shows each type of driver that might be present in a stack and includes a brief description of their purpose. This chapter describes the behavior of device drivers below the file system layer in the stack. (The file system driver operation is described in Chapter 12.)

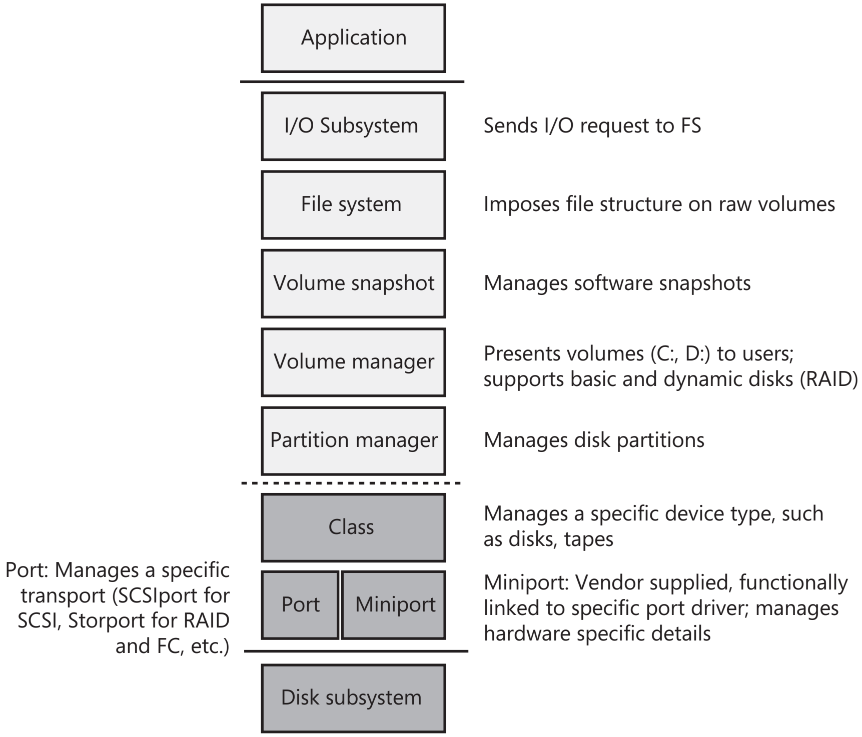

Application
I/O Subsystem
Sends I/O request to FS
File system
Imposes file structure on raw volumes
Volume snapshot
Manages software snapshots
Volume manager
Presents volumes (C:, D:) to users; supports basic and dynamic disks (RAID)
Partition manager
Manages disk partitions
Class
Manages a specific device type, such as disks, tapes
Port: Manages a specific transport (SCSIport for SCSI, Storport for RAID and FC, etc.)
Port
Miniport
Miniport: Vendor supplied, functionally linked to specific port driver; manages hardware specific details
Disk subsystem

Figure 10-1 Windows storage stack

Ntldr

As you saw in Chapter 4, Ntldr is the Windows operating system file that conducts the first portion of the Windows boot process. Although Ntldr isn't technically part of the storage stack, it is involved with storage management because it includes support for accessing disk devices before the Windows I/O system is operational. Ntldr resides on the system volume; the boot-sector code on the system volume executes Ntldr. Ntldr reads the Boot.ini file from the system volume and presents the computer's boot choices to the user. The partition names that Boot.ini designates are in the form multi(0)disk(0)rdisk(0)partition(1). These names are Advanced RISC Computing (ARC) names because they're part of a standard partition-naming scheme that Alpha firmware and other RISC processors use. Ntldr translates the name of the Boot.ini boot entry that a user selects to the appropriate boot partition and then loads the

Windows system files (starting with the registry, Ntoskrnl.exe, and the boot drivers) into memory to continue the boot process. In all cases, Ntldr uses the computer firmware to read the disk containing the system volume, but as described in Chapter 4, sometimes Ntldr relies on functions in a disk miniport driver to read from the boot volume's disk.

Disk Class, Port, and Miniport Drivers

During initialization, the Windows I/O manager starts the hard-disk storage drivers. Storage drivers in Windows follow a class/port/miniport architecture, in which Microsoft supplies a storage class driver that implements functionality common to all storage devices and a storage port driver that implements functionality common to a particular bus—such as a Small Computer System Interface (SCSI) bus or an Integrated Device Electronics (IDE) system—and OEMs supply miniport drivers that plug into the port driver to interface Windows to a particular implementation.

In the disk storage driver architecture, only class drivers conform to the standard Windows device driver interfaces. Miniport drivers use a port driver interface instead of the device driver interface, and the port driver simply implements a collection of device driver support routines that interface miniport drivers to Windows. This approach simplifies the role of miniport driver developers and, because Microsoft supplies operating system–specific port drivers, gives miniport drivers binary portability between Microsoft Windows 98, Microsoft Windows Millennium Edition, and Windows.

Windows includes Disk (\Windows\System32\Drivers\Disk.sys), a class driver that implements functionality common to disks. Windows also provides a handful of disk port drivers. For example, Scsiport.sys is the port driver for disks on SCSI buses, and Atapi.sys is a port driver for IDE-based systems. Windows Server 2003 introduces the Storport.sys port driver as a replacement for Scsiport.sys. Storport.sys was designed to realize the high performance capabilities of hardware RAID and Fibre Channel adapters. The Storport model is similar to Scsiport, making it easy for vendors to migrate existing SCSI miniport drivers to Storport. Miniport drivers that developers write to use Storport take advantage of several of Storport's performance enhancing features, including support for the parallel execution of I/O initiation and completion on multiprocessor systems, more controllable I/O request-queue architecture, and execution of more code at lower IRQL to minimize the duration of hardware interrupt masking.

The Scsiport.sys and Atapi.sys drivers both implement a version of the disk scheduling algorithm known as C-LOOK. The drivers place disk I/O requests in lists sorted by the first sector (also known as the logical block number or LBN) at which an I/O request is directed. They use the KeInsertByKeyDeviceQueue and KeRemoveByKeyDeviceQueue functions (documented in the Windows DDK) representing I/O requests as items and using a request's starting sector as the key required by the functions. When servicing requests, the drivers proceeed through the list from lowest sector to highest. When they reach the end of the list the driver starts back at the beginning, since new requests might have been inserted in the meantime. If disk requests are spread throughout a disk this approach results in the disk head continuously moving from near the start of a disk torward the end. Storport.sys does not implement disk scheduling since it is commonly used for managing I/O's directed at storage arrays where there is no clearly-defined notion of a disk start and end.

Windows ships with several miniport drivers, including one—Aha154x.sys—for Adaptec's 1540 family of SCSI controllers. On systems that have at least one ATAPI-based IDE device, Pciidex.sys and Pciide.sys together provide miniport functionality. Most Windows installations include one or more of the drivers mentioned.

iSCSI Drivers

The development of iSCSI as a disk transport protocol integrates the SCSI protocol with TCP/IP networking so that computers can communicate with block-storage devices, including disks, over IP networks. Storage area networking (SAN) is usually architected on Fibre Channel networking, but administrators can leverage iSCSI to create relatively inexpensive SANs from networking technology such as gigabit Ethernet to provide scalability, disaster protection, efficient backup, and data protection. Windows support for iSCSI comes in the form of the Microsoft iSCSI Software Initiator, which can be downloaded from the Microsoft Web site and which works on Windows 2000, Windows XP, and Windows Server 2003.

The Microsoft iSCSI Software Initiator includes several components:

- **Initiator** This optional component, which consists of an iSCSI port driver (\Windows\System32\Drivers\Iscsiprt.sys) and miniport driver (\Windows\System32\Drivers\Msiscis.sys), uses the TCP/IP driver to implement software iSCSI over standard Ethernet adapters and TCP/IP offloaded network adapters.
- **Initiator service** This service, implemented in \Windows\System32\Iscsiexe.exe, manages the discovery and security of all iSCSI initiators as well as session initiation and termination. iSCSI device discovery functionality is implemented in \Windows\System32\Iscsium.dll and conforms to the Internet Storage Name Service (iSNS) protocol.
- **Management applications** These include Iscsicli.exe, a command-line tool for managing iSCSI device connections and security, and the corresponding Control Panel application.

Some vendors produce iSCSI adapters that offload the iSCSI protocol to hardware. The Initiator service works with these adapters, which must support iSNS, so that all iSCSI devices, including those discovered by the Initiator service and those discovered by iSCSI hardware, are recognized and managed through standard Windows interfaces.

Multipath I/O (MPIO) Drivers

Most disk devices have one *path*—or series of adapters, cables, and switches—between them and a computer. Servers requiring high levels of availability use *multipathing* solutions, where more than one set of connection hardware exists between the computer and a disk so that if a path fails the system can still access the disk via an alternate path. Without support of the operating system or disk drivers, however, a disk with two paths, for example, appears as two different disks. Windows includes multipath I/O support to manage multipath disks as a single disk that relies on third-party drivers called *device-specific modules* (DSMs) to manage details of the path management, such as load balancing policies that choose which path to route requests and error detection mechanisms to inform Windows when a path fails. MPIO support is available for Windows 2000 Server, Advanced Server, and Datacenter Server as well as Windows Server 2003 in the form of the Microsoft MPIO Driver Development Kit that hardware and software vendors can license.

In a Windows MPIO storage stack, shown in Figure 10-2, the Multipath Disk Driver Replacement (\Windows\System32\Drivers\Mpdev.sys) replaces the functionality of the standard Disk.sys class driver. Mpdev.sys is responsible for claiming ownership of device objects representing multipath disks—so that it can ensure that only one device object is created to represent those disks—and for locating the appropriate DSM to manage the paths to the device. The Multipath Bus Driver (\Windows\System32\Drivers\Mpio.sys) manages connections between the computer and the device, including power management for the device. Mpdev.sys informs Mpio.sys of the presence of the devices for it to manage. Finally, the Multipath Port Filter (\Windows\System32\Drivers\Mpsfltr.sys) layers on top of the port driver for a multipath disk to manage information passed up the device stack.

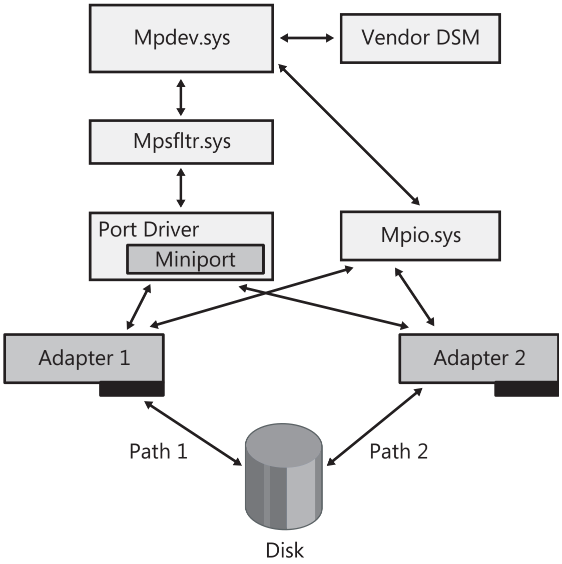
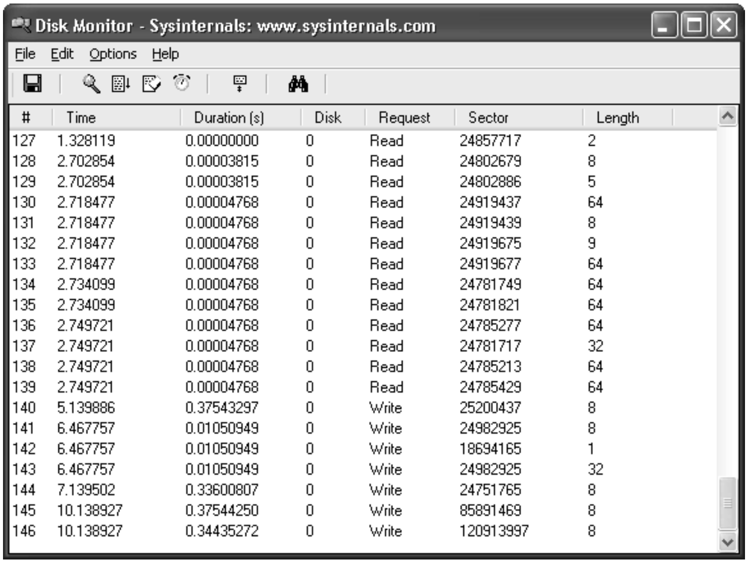


Figure 10-2 Windows MPIO storage stack

EXPERIMENT: Watching Physical Disk I/O

Diskmon from Sysinternals uses the disk class driver's Event Tracing for Windows (or ETW, which is described in Chapter 3) instrumentation to monitor I/O activity to physical disks and display it in a window. Diskmon updates once a second with new data. For each operation, Diskmon shows the time, duration, target disk number, type and offset, and length.



#	Time	Duration (s)	Disk	Request	Sector	Length
127	1.328119	0.00000000	0	Read	24857717	2
128	2.702854	0.00003815	0	Read	24802679	8
129	2.702854	0.00003815	0	Read	24802886	5
130	2.718477	0.00004768	0	Read	24919437	64
131	2.718477	0.00004768	0	Read	24919439	8
132	2.718477	0.00004768	0	Read	24919675	9
133	2.718477	0.00004768	0	Read	24919677	64
134	2.734099	0.00004768	0	Read	24781749	64
135	2.734099	0.00004768	0	Read	24781821	64
136	2.749721	0.00004768	0	Read	24785277	64
137	2.749721	0.00004768	0	Read	24781717	32
138	2.749721	0.00004768	0	Read	24785213	64
139	2.749721	0.00004768	0	Read	24785429	64
140	5.139886	0.37543297	0	Write	25200437	8
141	6.467757	0.01050949	0	Write	24982925	8
142	6.467757	0.01050949	0	Write	18694165	1
143	6.467757	0.01050949	0	Write	24982925	32
144	7.139502	0.33600807	0	Write	24751765	8
145	10.138927	0.37544250	0	Write	85891469	8
146	10.138927	0.34435272	0	Write	120913997	8

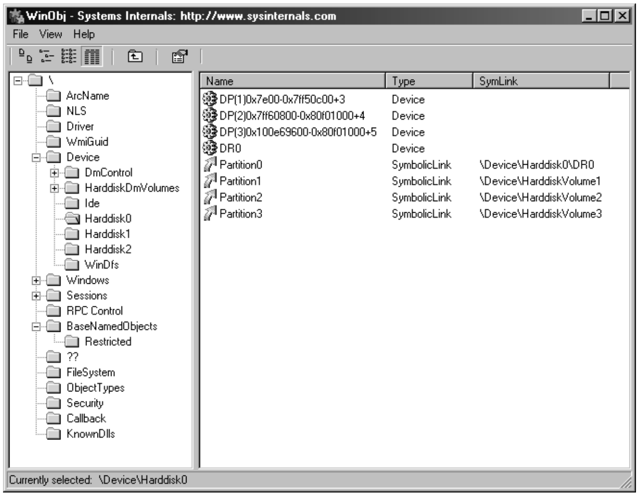
Disk Device Objects

The Windows disk class driver creates device objects that represent disks and partitions. Device objects that represent disks have names of the form \Device\ Harddisk*X*\DR*X*; the number that identifies the disk replaces both *X*s. The disk class driver uses the I/O manager's *IoReadPartitionTable* function in Windows 2000 and *IoReadParititionTableEx* function in Windows XP and Windows Server 2003 to identify partitions and create device objects that represent them. As miniport drivers present the disks that they identify early in the boot process to the disk class driver, the disk class driver invokes the *IoReadPartitionTable* or *IoReadPartitionTableEx* function for each disk. These functions invoke sector-level disk I/O that the class, port, and miniport drivers provide to read a disk's MBR-style or GPT-style partition table (described later in this chapter) and construct an internal representation of the disk's hard partitioning. The disk class driver creates device objects to represent each primary partition (including logical drives within extended partitions) that the driver obtains from *IoReadPartitionTable* or *IoReadPartitionTableEx*. The following is a sample partition object name:

\Device\Harddisk0\DP(1)0x7e000-0x7ff50c00+2

This name identifies the first partition on a system's first disk. The first two hexadecimal numbers (0x7e000 and 0x7ff50c00) designate the start and length of the partition, and the last number is an internal identifier that the class driver assigns.

To maintain compatibility with applications that use the Windows NT 4 naming conventions, the disk class driver creates symbolic links with Windows NT 4–formatted names that refer to the device objects the driver created. For example, the Disk class driver creates the link \Device\Harddisk0\Partition0 to refer to \Device\Harddisk0\DR0, and \Device\Harddisk0\Partition1 to refer to the first partition device object of the first disk. The class driver also creates the same symbolic links in Windows that represent physical drives that it created in Windows NT 4 systems. Thus, for example, the link \??\PhysicalDrive0 references \Device\Harddisk0\DR0. Figure 10-3 shows the Winobj utility from Sysinternals displaying the contents of a Harddisk directory for a basic disk. You can see the physical disk and partition device objects in the right-hand pane.



Name	Type	SymLink
DP(1)0x7e00-0x7ff50c00+3	Device	
DP(2)0x7ff60800-0x80f01000+4	Device	
DP(3)0x100e69600-0x80f01000+5	Device	
DR0	Device	
Partition0	SymbolicLink	\Device\Harddisk0\DR0
Partition1	SymbolicLink	\Device\HarddiskVolume1
Partition2	SymbolicLink	\Device\HarddiskVolume2
Partition3	SymbolicLink	\Device\HarddiskVolume3

Currently selected: \Device\Harddisk0

Figure 10-3 Winobj showing a Harddisk directory of a basic disk

As you saw in Chapter 3, the Windows API is unaware of the Windows object manager namespace. Windows reserves two different namespace subdirectories to use, one of which is the \Global?? (\?? on Windows 2000) subdirectory. (The other is the \BaseNamedObjects subdirectory, which is covered in Chapter 3.) In this subdirectory, Windows makes available device objects that Windows applications interact with—including COM and parallel ports—as well as disks. Because disk objects actually reside in other subdirectories, Windows uses symbolic links to connect names under \Global?? with objects located elsewhere in the namespace. For each physical disk on a system, the I/O manager creates a \Global??\PhysicalDrive*X* link that points to \Device\Harddisk*X*\Partition0. (Numbers, starting from 0, replace *X*.) Windows applications that directly interact with the sectors on a disk open the disk by calling the Windows *CreateFile* function and specifying the name \\.\PhysicalDrive*X* (in which *X* is the disk number) as a parameter. The Windows application layer converts the name to \Global??\PhysicalDrive*X* before handing the name to the Windows object manager.

Partition Manager

The Partition Manager, \Windows\System32\Drivers\Partmgr.sys, is responsible for notifying the Plug and Play (PnP) manager of the existence of partitions so that volume manager drivers, described later in this chapter, can receive notification of the creation and deletion of partitions.

To become aware of partitions, Partition Manager acts as the function driver for disk device objects created by disk class drivers. When the system boots, it reads the partition tables of attached disks (by relying on the kernel's *IoReadPartitionTable* function on Windows 2000 and *IoReadPartitionTableEx* on Windows XP and Windows Server 2003) and reports their existence to the PnP manager. Volume manager device drivers receive the notification of partitions for disks that they manage and define volume objects when they account for all the partitions that make up the volumes. Because Partition Manager's device object's filter disk device objects, Partition Manager monitors I/O request packets (IRPs) related to partition table modification and can therefore update its internal partition map and notify the PnP manager of partition creations and deletions.

Volume Management

Windows introduces the concept of *basic* and *dynamic* disks. Windows calls disks that rely exclusively on the MBR-style or GPT partitioning scheme *basic disks*. Dynamic disks are new to Windows 2000 and implement a more flexible partitioning scheme than that of basic disks. The fundamental difference between basic and dynamic disks is that dynamic disks support the creation of new multipartition volumes. Recall from the list of terms in the preceding section that multipartition volumes provide performance, sizing, and reliability features not supported by simple volumes. Windows manages all disks as basic disks unless you manually create dynamic disks or convert existing basic disks (with enough free space) to dynamic disks. Microsoft recommends you use basic disks unless you require the multipartition-functionality of dynamic disks.

Note Basic disks support only multipartition volumes carried forward from an upgraded Windows NT 4 installation with the exception of Windows Server 2003, which does not support multipartition volumes on basic disks. For a number of reasons, including the fact that laptops usually have only one disk and laptop disks typically don't move easily between computers, Windows uses only basic disks on laptops. In addition, only fixed disks can be dynamic, and disks located on IEEE 1394 or USB buses as well as on shared cluster server disks are always basic disks.

Windows Volume Management Evolution

The evolution of storage management in Windows begins with MS-DOS, Microsoft's first operating system. As hard disks became larger, MS-DOS needed to accommodate them. To do so, one of the first steps Microsoft took was to let MS-DOS create multiple partitions, or logical disks, on a physical disk. MS-DOS could format each partition with a different file system type (FAT12 or FAT16) and assign each partition a different drive letter. MS-DOS versions 3 and 4 were severely limited in the size and number of partitions they could create, but in MS-DOS 5 the partitioning scheme fully matured. MS-DOS 5 was able to divide a disk into any number of partitions of any size.

Windows NT borrowed the partitioning scheme that evolved in MS-DOS both to provide disk compatibility with MS-DOS and Windows 3.*x* and to let the Windows NT development team rely on proven tools for disk management. Microsoft extended the basic concepts of MS-DOS disk partitioning in Windows NT to support storage-management features that an enterprise-class operating system requires: disk spanning and fault tolerance. Starting with the first version of Windows NT, version 3.1, systems administrators have been able to create volumes that comprise multiple partitions, which allows large volumes to consist of partitions from multiple physical disks and to implement fault tolerance through software-based data redundancy.

Although this MS-DOS-style partitioning support in versions of Windows NT prior to Windows 2000 is flexible enough to support most storage-management tasks, it suffers from several drawbacks. One drawback is that most disk-configuration changes require a reboot before taking effect. In today's world of servers that must remain on line for months or even years at a time, any reboot—even a planned reboot—is a major inconvenience. Another drawback is that the Windows NT 4 registry stores multipartition disk-configuration information for MS-DOS-style partitions. This arrangement means that moving configuration information is onerous when you move disks between systems, and you can easily lose configuration information when you need to reinstall the operating system. Finally, a requirement that each volume have a unique drive letter in the A through Z range plagues users of all Microsoft operating systems prior to Windows 2000 with an upper limit on the number of possible local and remote volumes they can create.

Windows includes three types of partitioning that enable it to overcome the limitations mentioned when necessary: Master Boot Record (MBR)–style, GUID Partition Table (GPT), and Logical Disk Manager (LDM).

Basic Disks

This section describes the two types of partitioning, MBR-style and GPT, that Windows uses to define volumes on basic disks, and FtDisk, the volume manager driver that presents the volumes to file system drivers. The Windows 2000 Disk Manager recommends that you make any unpartitioned disk a dynamic disk, but Windows Server 2003, like Windows XP and Windows 2000 Professional, silently defaults to defining all disks as basic disks.

MBR-Style Partitioning

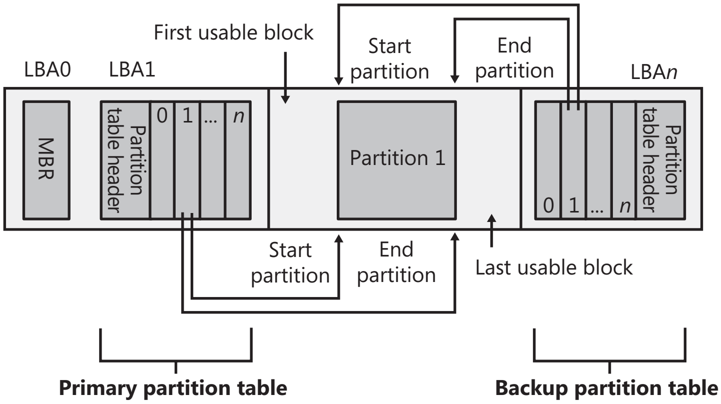
The standard BIOS implementations that x86 hardware uses dictate one requirement of the partitioning format in Windows—that the first sector of the primary disk contains the Master Boot Record (MBR). When an x86 processor boots, the computer's BIOS reads the MBR and treats part of the MBR's contents as executable code. The BIOS invokes the MBR code to initiate an operating system boot process after the BIOS performs preliminary configuration of the computer's hardware. In Microsoft operating systems, including Windows, the MBR also contains a *partition table*. A partition table consists of four entries that define the locations of as many as four *primary partitions* on a disk. The partition table also records a partition's type. Numerous predefined partition types exist, and a partition's type specifies which file system the partition includes. For example, partition types exist for FAT32 and NTFS. A special partition type, an *extended partition*, contains another MBR with its own partition table. The equivalent of a primary partition in an extended partition is called a *logical drive*. By using extended partitions, Microsoft's operating systems overcome the apparent limit of four partitions per disk. In general, the recursion that extended partitions permit can continue indefinitely, which means that no upper limit exists to the number of possible partitions on a disk. The Windows boot process makes evident the distinction between primary and logical drives. The system must mark one primary partition of the primary disk as active. The Windows code in the MBR loads the code stored in the first sector of the active partition (the system volume) into memory and then transfers control to that code. Because of the role in the boot process played by this first sector in the primary partition, Windows designates the first sector of any partition as the boot sector. Recall from Chapter 4 that every partition formatted with a file system has a boot sector that stores information about the structure of the file system on that partition.

GUID Partition Table Partitioning

As part of an initiative to provide a standardized and extensible firmware platform for operating systems to use during their boot process, Intel has designed the Extensible Firmware Interface (EFI) specification. EFI includes a mini–operating system environment implemented in firmware (typically ROM) that operating systems use early in the system boot process to load system diagnostics and their boot code. One of the first targets of EFI is IA-64, so the IA-64 versions of Windows use EFI. (You can still choose to create disks with the old-style MBR portioning.) You can find a complete description of EFI at *http://developer.intel.com/technology/efi*.

EFI defines a partitioning scheme, called the GUID (Globally Unique Identifier) Partition Table (GPT) that addresses some of the shortcomings of MBR-style partitioning. For example, the sector addresses that the GPT partition structures use are 64 bits wide instead of 32 bits. A 32-bit sector address is sufficient to access only 2 terabytes (TB) of storage, while GPT allows the addressing of disk sizes into the foreseeable future. Other advantages of the GPT scheme include the fact that it uses cyclic redundancy checksums (CRC) to ensure the integrity of the partition table, and it maintains a backup copy of the partition table. GPT takes its name from the fact that in addition to storing a 36-byte Unicode partition name for each partition, it assigns each partition a GUID.

Figure 10-4 shows a sample GPT partition layout. Like for MBR partitioning, the first sector of a GPT disk is an MBR that serves to protect the GPT partitioning in case the disk is accessed from a non-GPT aware operating system. However, the second and last sectors of the disk store the GPT partition table headers with the actual partition table following the second sector and preceding the last sector. With its extensible list of partitions, GPT partitioning doesn't require nested partitions, as MBR partitions do.



Note: LBA = Logical Block Address

Figure 10-4 Example GPT partition layout

> **Note** Because Windows doesn't support the creation of multipartition volumes on basic disks, a new basic disk partition is the equivalent of a volume. For this reason, the Disk Management MMC snap-in uses the term *partition* when you create a volume on a basic disk.

Basic Disk Volume Manager

The FtDisk driver (\Windows\System32\Drivers\Ftdisk.sys) creates disk device objects that represent volumes on basic disks and plays an integral role in managing all basic disk volumes, including simple volumes. For each volume, FtDisk creates a device object of the form \Device\HarddiskVolume*X*, in which *X* is a number (starting from 1) that identifies the volume.

FtDisk is actually a bus driver because it's responsible for enumerating basic disks to detect the presence of basic volumes and report them to the Windows PnP manager. To implement this enumeration, FtDisk leverages the PnP manager, with the aid of the partition manager (Partmgr.sys) driver to determine what basic disk partitions exist. The partition manager registers with the PnP manager so that Windows can inform the partition manager whenever the disk class driver creates a partition device object. The partition manager informs FtDisk about new partition objects through a private interface and creates filter device objects that the partition manager then attaches to the partition objects. The existence of the filter objects prompts Windows to inform the partition manager whenever a partition device object is deleted so that the partition manager can update FtDisk. The disk class driver deletes a partition device object when a partition in the Disk Management Microsoft Management Console (MMC) snap-in is deleted. As FtDisk becomes aware of partitions, it uses the basic disk configuration information to determine the correspondence of partitions to volumes and creates a volume device object when it has been informed of the presence of all the partitions in a volume's description.

Windows volume drive-letter assignment, a process described shortly, creates drive-letter symbolic links under the \Global?? (\?? on Windows 2000) object manager directory that point to the volume device objects that FtDisk creates. When the system or an application accesses a volume for the first time, Windows performs a mount operation that gives file system drivers the opportunity to recognize and claim ownership for volumes formatted with a file system type they manage. (Mount operations are described in the section "Volume Mounting" later in this chapter.)

Dynamic Disks

As we've stated, dynamic disks are the disk format in Windows necessary for creating multipartition volumes such as mirrors, stripes and RAID-5 (described later in the chapter). Dynamic disks are partitioned using Logical Disk Manager (LDM) partitioning. The LDM subsystem in Windows, which consists of user-mode and device driver components, oversees dynamic disks. Microsoft licenses LDM from VERITAS Software, which originally developed LDM technology for UNIX systems. Working closely with Microsoft, VERITAS ported its LDM to Windows to provide the operating system with more robust partitioning and multipartition volume capabilities. A major difference between LDM's partitioning and MBR and GPT partitioning is that LDM maintains one unified database that stores partitioning information for all the dynamic disks on a system—including multipartition-volume configuration.

Note LDM's UNIX version incorporates *disk groups*, in which all the dynamic disks that the system assigns to a disk group share a common database. VERITAS's commercial volume-management software for Windows also includes disk groups, but the Windows LDM implementation includes only one disk group.

The LDM Database

The LDM database resides in a 1-MB reserved space at the end of each dynamic disk. The need for this space is the reason Windows requires free space at the end of a basic disk before you can convert it to a dynamic disk. The LDM database consists of four regions, which Figure 10-5 shows: a header sector that LDM calls the Private Header, a table of contents area, a database records area, and a transactional log area. (The fifth region shown in Figure 10-5 is simply a copy of the Private Header.) The Private Header sector resides 1 MB before the end of a dynamic disk and anchors the database. As you spend time with Windows, you'll quickly notice that it uses GUIDs to identify just about everything, and disks are no exception. A GUID (globally unique identifier) is a 128-bit value that various components in Windows use to uniquely identify objects. LDM assigns each dynamic disk a GUID, and the Private Header sector notes the GUID of the dynamic disk on which it resides—hence the Private Header's designation as information that is private to the disk. The Private Header also stores the name of the disk group, which is the name of the computer concatenated with Dg0 (for example, DarylDg0 if the computer's name is Daryl), and a pointer to the beginning of the database table of contents. (As mentioned earlier, the Windows implementation of LDM includes only one disk group, so the disk group name will always end with Dg0.) For reliability, LDM keeps a copy of the Private Header in the disk's last sector.

The database table of contents is 16 sectors in size and contains information regarding the database's layout. LDM begins the database record area immediately following the table of contents with a sector that serves as the database record header. This sector stores information about the database record area, including the number of records it contains, the name and GUID of the disk group the database relates to, and a sequence number identifier that LDM uses for the next entry it creates in the database. Sectors following the database record header contain 128-byte fixed-size records that store entries that describe the disk group's partitions and volumes.

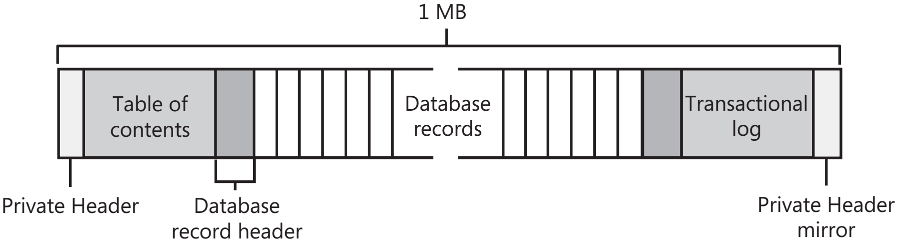


Figure 10-5 LDM database layout

A database entry can be one of four types: partition, disk, component, and volume. LDM uses the database entry types to identify three levels that describe volumes. LDM connects entries with internal object identifiers. At the lowest level, *partition entries* describe soft partitions, which are contiguous regions on a disk; identifiers stored in a partition entry link the entry to a component and disk entry. A *disk entry* represents a dynamic disk that is part of the disk group and includes the disk's GUID. A *component entry* serves as a connector between one or more partition entries and the volume entry each partition is associated with. A *volume entry* stores the GUID of the volume, the volume's total size and state, and a drive-letter hint. Disk entries that are larger than a database record span multiple records; partition, component, and volume entries rarely span multiple records.

LDM requires three entries to describe a simple volume: a partition, component, and volume entry. The following listing shows the contents of a simple LDM database that defines one 200-MB volume that consists of one partition:

```
Disk Entry      Volume Entry    Component Entry    Partition Entry
Name: Disk1     Name: Volume1   Name: Volume1-01   Name: Disk1-01
GUID: XXX-XX... ID: 0x408       ID: 0x409          ID: 0x407
Disk ID: 0x404  State: ACTIVE   Parent ID: 0x408   Parent ID: 0x409
                Size: 200MB                        Disk ID: 0x404
                GUID: XXX-XX...                    Start: 300MB
                Drive Hint: H:                     Size: 200MB
```

The partition entry describes the area on a disk that the system assigned to the volume, the component entry connects the partition entry with the volume entry, and the volume entry contains the GUID that Windows uses internally to identify the volume. Multipartition volumes require more than three entries. For example, a striped volume (which is described later in the chapter) consists of at least two partition entries, a component entry, and a volume entry. The only volume type that has more than one component entry is a mirror; mirrors have two component entries, each of which represents one-half of the mirror. LDM uses two component entries for mirrors so that when you break a mirror, LDM can split it at the component level, creating two volumes with one component entry each.

The final area of the LDM database is the transactional log area, which consists of a few sectors for storing backup database information as the information is modified. This setup safeguards the database in case of a crash or power failure because LDM can use the log to return the database to a consistent state.

EXPERIMENT: Using LDMDump to View the LDM Database

You can use LDMDump from Sysinternals to view detailed information about the contents of the LDM database. LDMDump takes a disk number as a command-line argument, and its output is usually more than a few screens in size, so you should pipe its output to a file for viewing in a text editor—for example, *ldmdump /d0 > disk.txt*. The following example shows excerpts of LDMDump output. The LDM database header displays first, followed by the LDM database records that describe a 4-GB disk with a 4-GB simple volume. The volume's database entry is listed as Volume1. At the end of the output, LDMDump lists the soft partitions and definitions of volumes it locates in the database.

```
C:\>ldump /d0
Logical Disk Manager Configuration Dump v1.03
Copyright (C) 2000-2002 Mark Russinovich

PRIVATE HEAD:
Signature          : PRIVHEAD
Version            : 2.11
Disk Id            : b78e4169-2e39-4a57-a98b-afd7131392f9
Host Id            : 1b77da20-c717-11d0-a5be-00a0c91db73c
Disk Group Id      : 2ea3c400-c92a-4172-9286-de46dda1098a
Disk Group Name    : Vmware03Dg0
Logical disk start : 3F
Logical disk size  : 7FF54B (4094 MB)
Configuration start: 7FF7E0
Configuration size : 800 (1 MB)
Number of TOCs     : 1
TOC size           : 7FE (1023 KB)
Number of Configs  : 1
Config size        : 5AC (726 KB)
Number of Logs     : 1
Log size           : DC (110 KB)

TOC 0:
Signature          : TOCBLOCK
Sequence           : 0x9
Config bitmap start: 0x11
Config bitmap size : 0x5AC
...
VBLK DATABASE:
0x000004: [000004] <Disk>
         Name        : Disk1
         Object Id   : 0x0403
         Disk Id     : b78e4169-2e39-4a57-a98b-afd7131392f9

0x000005: [000002] <DiskGroup>
         Name        : Vmware03Dg0
         Object Id   : 0x0401
         GUID        : 2ea3c400-c92a-4172-9286-de46dda1098a
0x000006: [000006] <Volume>
         Name        : Volume1
         Object Id   : 0x0406
         Volume state: ACTIVE
         Size        : 0x007FB68A (4086 MB)
         GUID        : e6ee6edc-d1ba-11d8-813e-806e6f6e6963
         Drive Hint  : C:

...
PARTITION LAYOUT:
Disk Disk1:
        Disk1-01 Offset: 0x00000000 Length: 0x007FB68A (4086 MB)
VOLUME DEFINITIONS: Volume1 Size: 0x007FB68A (4086 MB)
    Volume1-01 -
      Disk1-01   VolumeOffset: 0x00000000 Offset: 0x00000000 Length: 0x007FB68A
```

LDM and GPT or MBR-Style Partitioning

When you install Windows on a computer, one of the first things it requires you to do is to create a partition on the system's primary physical disk. Windows defines the system volume on this partition to store the files that it invokes early in the boot process. In addition, Windows Setup requires you to create a partition that serves as the home for the boot volume, onto which the setup program installs the Windows system files and creates the system directory (\Windows). The system and boot volumes can be the same volume, in which case you don't have to create a new partition for the boot volume. The nomenclature that Microsoft defines for system and boot volumes is somewhat confusing. The system volume is where Windows places boot files, including the boot loader (Ntldr) and Ntdetect, and the boot volume is where Windows stores operating system files such as Ntoskrnl.exe, the core kernel file.

Although the partitioning data of a dynamic disk resides in the LDM database, LDM implements MBR-style partitioning or GPT partitioning so that the Windows boot code can find the system and boot volumes, when the volumes are on dynamic disks. (Ntldr and the IA64 firmware, for example, know nothing about LDM partitioning.) If a disk contains the system or boot volumes, partitions in the MBR or GPT partition table describe the location of those volumes. Otherwise, one partition encompasses the entire usable area of the disk. LDM marks this partition as type "LDM," a partition type new to Windows 2000. The region encompassed by this place-holding MBR-style or GPT partition is where LDM creates partitions that the LDM database organizes. On MBR-partitioned disks the LDM database resides in hidden sectors at the end of the disk and on GPT-partitioned disks there exists an LDM metadata partition that encompasses the LDM database near the beginning of the disk.

Another reason LDM creates an MBR-style or GPT partition table is so that legacy disk-management utilities, including those that run under Windows and under other operating systems in dual-boot environments, don't mistakenly believe a dynamic disk is unpartitioned.

Because LDM partitions aren't described in the MBR-style or GPT partition table of a disk, they are called *soft partitions*; MBR-style and GPT partitions are called *hard partitions*. Figure 10-6 illustrates this dynamic disk layout on an MBR-style partitioned disk.

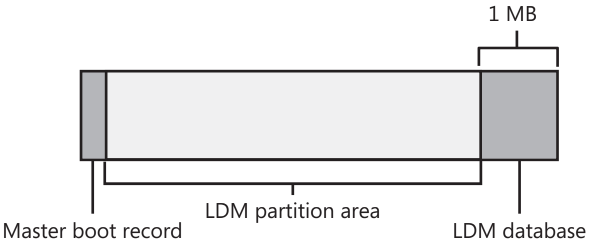
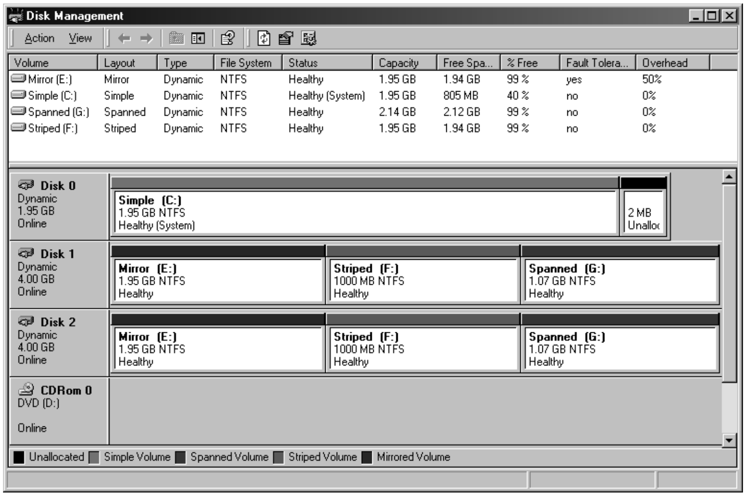


Figure 10-6 Internal dynamic disk organization

Dynamic Disk Volume Manager

The Disk Management MMC snap-in DLL (DMDiskManager, located in \Windows\System32\Dmdskmgr.dll) and shown in Figure 10-7, uses DMAdmin, the LDM Disk Administrator service (\Windows\System32\Dmadmin.exe), to create and change the contents of the LDM database. When you launch the Disk Management MMC snap-in, DMDiskManager loads into memory and starts DMAdmin, if it's not already running. DMAdmin reads the LDM database from each disk and returns the information it obtains to DMDiskManager. If DMAdmin detects a database from another computer's disk group, it notes that the volumes on the disk are foreign and lets you import them into the current computer's database if you want to use them. As you change the configuration of dynamic disks, DMDiskManager informs DMAdmin of the changes and DMAdmin updates its in-memory copy of the database. When DMAdmin commits changes, it passes the updated database to DMIO, the Dmio.sys device driver. DMIO is the dynamic disk equivalent of FtDisk, so it controls access to the on-disk database and creates device objects that represent the volumes on dynamic disks. When you exit Disk Management, DMDiskManager stops and unloads the DMAdmin service.

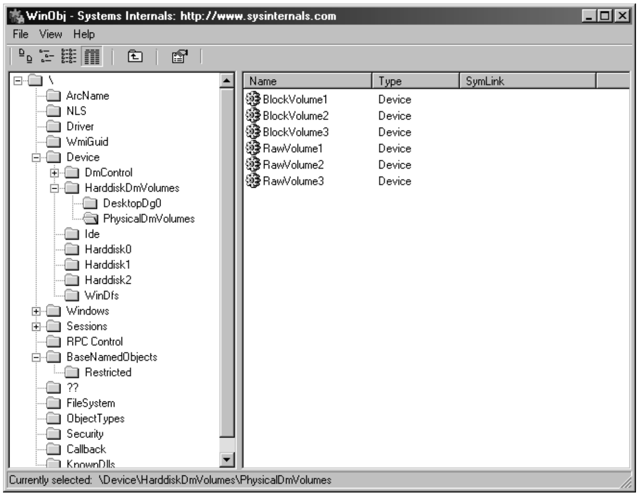


Volume	Layout	Type	File System	Status	Capacity	Free Spa...	% Free	Fault Tolera...	Overhead
Mirror (E:)	Mirror	Dynamic	NTFS	Healthy	1.95 GB	1.94 GB	99 %	yes	50%
Simple (C:)	Simple	Dynamic	NTFS	Healthy (System)	1.95 GB	805 MB	40 %	no	0%
Spanned (G:)	Spanned	Dynamic	NTFS	Healthy	2.14 GB	2.12 GB	99 %	no	0%
Striped (F:)	Striped	Dynamic	NTFS	Healthy	1.95 GB	1.94 GB	99 %	no	0%

Figure 10-7 Disk Management MMC snap-in

DMIO doesn't know how to interpret the database it oversees. DMConfig (\Windows\System32\Dmconfig.dll), a DLL that DMAdmin loads, and another device driver, DMBoot (Dmboot.sys), are responsible for interpreting the database. DMConfig knows how to both read and update the database; DMBoot knows only how to read the database. DMBoot loads during the boot process if another LDM driver, DMLoad (Dmload.sys), determines that at least one dynamic disk is present on the system. DMLoad makes this determination by asking DMIO, and if at least one dynamic disk is present, DMLoad starts DMBoot, which scans the LDM database. DMBoot informs DMIO of the composition of each volume it encounters so that DMIO can create device objects to represent the volumes. DMBoot unloads from memory immediately after it finishes its scan. Because DMIO has no database interpretation logic, it is relatively small. Its small size is beneficial because DMIO is always loaded.

Like FtDisk, DMIO is a bus driver and creates a device object for each dynamic disk volume with a name in the form \Device\HarddiskDmVolumes\PhysicalDmVolumes\BlockVolume*X*, in which *X* is an identifier that DMIO assigns to the volume. In addition, DMIO creates another device object that represents raw (unstructured) I/O in a volume named \Device\HarddiskDmVolumes\PhysicalDmVolumes\RawVolume*X*. Figure 10-8 shows the device objects that DMIO created on a system that consists of three dynamic disk volumes. DMIO also creates numerous symbolic links in the object manager namespace for each volume, starting with one link in the form \Device\HarddiskDmVolumes\ComputerNameDg0\Volume*Y* for each volume. DMIO replaces *ComputerName* with the name of the computer and replaces *Y* with a volume identifier (different from the internal identifier that DMIO assigns to the device objects). These links refer to the block-device objects under the PhysicalDmVolumes directory.



Name	Type	SymLink
BlockVolume1	Device	
BlockVolume2	Device	
BlockVolume3	Device	
RawVolume1	Device	
RawVolume2	Device	
RawVolume3	Device	

Currently selected: \Device\HarddiskDmVolumes\PhysicalDmVolumes

Figure 10-8 DMIO driver device objects

Note Another driver present in Windows 2000 is DiskPerf (Disk Performance driver, located at \Windows\System32\Drivers\Diskperf.sys). DiskPerf attaches device objects to the device objects that represent physical disks (for example, \Device\Harddisk0\Partition0) so that it can monitor I/O requests targeted at disks and generate performance-related statistics for the Performance tool to present. Statistics include bytes read and written per second, transfers per second, and the amount of time spent performing disk I/O. On Windows XP and Windows Server 2003, DiskPerf's functionality is implemented in the Partition Manager driver because it already filters physical disk device objects to implement its other functions (described earlier in this chapter).

Multipartition Volume Management

FtDisk and DMIO are responsible for presenting volumes that file system drivers manage and for mapping I/O directed at volumes to the underlying partitions that they're made of. For simple volumes, this process is straightforward: the volume manager ensures that volume-relative offsets are translated to disk-relative offsets by adding the volume-relative offset to the volume's starting disk offset.

Multipartition volumes are more complex because the partitions that make up a volume can be located on discontiguous partitions or even on different disks. Some types of multipartition volumes use data redundancy, so they require more involved volume-to-disk–offset translation. Thus, DMIO must process all I/O requests aimed at the multipartition volumes they manage by determining which partitions the I/O ultimately affects.

The following types of multipartition volumes are available in Windows:

- Spanned volumes
- Mirrored volumes
- Striped volumes
- RAID-5 volume

After describing multipartition-volume partition configuration and logical operation for each of the multipartition-volume types, we'll cover the way that the FtDisk and DMIO drivers handle IRPs that a file system driver sends to multipartition volumes. The term *volume manager* is used to represent DMIO throughout the explanation of multipartition volumes because, as mentioned earlier in the chapter, FtDisk only supports multipartition-volume type migrated from NT 4.

Spanned Volumes

A *spanned volume* is a single logical volume composed of a maximum of 32 free partitions on one or more disks. The Windows Disk Management MMC snap-in combines the partitions into a spanned volume, which can then be formatted for any of the Windows-supported file systems. Figure 10-9 shows a 100-MB spanned volume identified by drive letter D: that has been created from the last third of the first disk and the first third of the second. Spanned volumes were called *volume sets* in Windows NT 4.

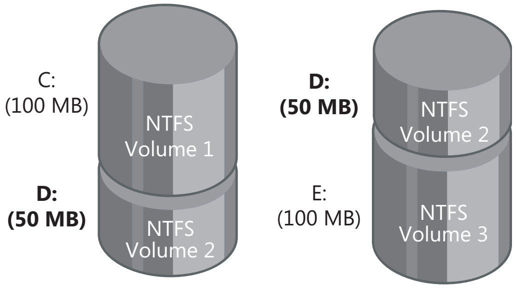


Figure 10-9 Spanned volume

A spanned volume is useful for consolidating small areas of free disk space into one larger volume or for creating a single, large volume out of two or more small disks. If the spanned volume has been formatted for NTFS, it can be extended to include additional free areas or additional disks without affecting the data already stored on the volume. This extensibility is one of the biggest benefits of describing all data on an NTFS volume as a file. NTFS can dynamically increase the size of a logical volume because the bitmap that records the allocation status of the volume is just another file—the bitmap file. The bitmap file can be extended to include any space added to the volume. Dynamically extending a FAT volume, on the other hand, would require the FAT itself to be extended, which would dislocate everything else on the disk.

A volume manager hides the physical configuration of disks from the file systems installed on Windows. NTFS, for example, views volume D: in Figure 10-9 as an ordinary 100-MB volume. NTFS consults its bitmap to determine what space in the volume is free for allocation. It then calls the volume manager to read or write data beginning at a particular byte offset on the volume. The volume manager views the physical sectors in the spanned volume as numbered sequentially from the first free area on the first disk to the last free area on the last disk. It determines which physical sector on which disk corresponds to the supplied byte offset.

Striped Volumes

A *striped volume* is a series of up to 32 partitions, one partition per disk, that gets combined into a single logical volume. Striped volumes are also known as *RAID level 0* (RAID-0) volumes. Figure 10-10 shows a striped volume consisting of three partitions, one on each of three disks. (A partition in a striped volume need not span an entire disk; the only restriction is that the partitions on each disk be the same size.)

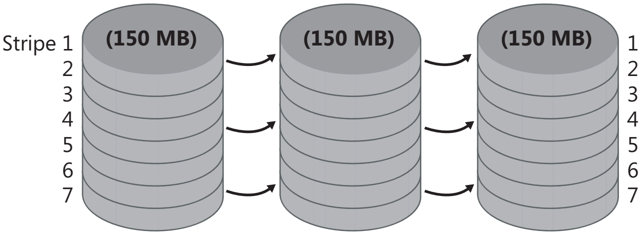


Figure 10-10 Striped volume

To a file system, this striped volume appears to be a single 450-MB volume, but a volume manager optimizes data storage and retrieval times on the striped volume by distributing the volume's data among the physical disks. The volume manager accesses the physical sectors of the disks as if they were numbered sequentially in stripes across the disks, as illustrated in Figure 10-11.

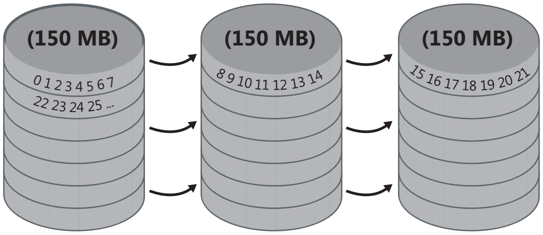


Figure 10-11 Logical numbering of physical sectors on a striped volume

Because each stripe is a relatively narrow 64 KB (a value chosen to prevent individual reads and writes from accessing two disks), the data tends to be distributed evenly among the disks. Stripes thus increase the probability that multiple pending read and write operations will be bound for different disks. And because data on all three disks can be accessed simultaneously, latency time for disk I/O is often reduced, particularly on heavily loaded systems.

Spanned volumes make managing disk volumes more convenient, and striped volumes spread the I/O load over multiple disks. These two volume-management features don't provide the ability to recover data if a disk fails, however. For data recovery, a volume manager implements three redundant storage schemes: mirrored volumes, RAID-5 volumes, and sector sparing. (Sector sparing and NTFS support for sector sparing are described in Chapter 12.) These features are created with the Windows Disk Management administrative tool.

Mirrored Volumes

In a *mirrored volume*, the contents of a partition on one disk are duplicated in an equal-sized partition on another disk. Mirrored volumes are sometimes referred to as *RAID level 1* (RAID-1). A mirrored volume is shown in Figure 10-12.

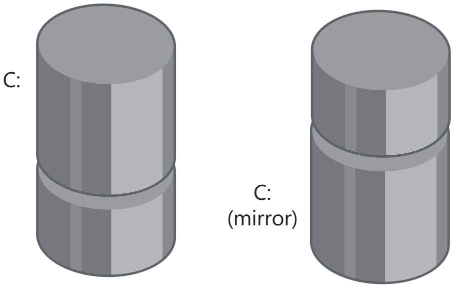


Figure 10-12 Mirrored volume

When a program writes to drive C:, the volume manager writes the same data to the same location on the mirror partition. If the first disk or any of the data on its C: partition becomes unreadable because of a hardware or software failure, the volume manager automatically accesses the data from the mirror partition. A mirror volume can be formatted for any of the Windows-supported file systems. The file system drivers remain independent and are not affected by the volume manager's mirroring activity.

Mirrored volumes can aid in read I/O throughput on heavily loaded systems. When I/O activity is high, the volume manager balances its read operations between the primary partition and the mirror partition (accounting for the number of unfinished I/O requests pending from each disk). Two read operations can proceed simultaneously and thus theoretically finish in half the time. When a file is modified, both partitions of the mirror set must be written, but disk writes are done asynchronously, so the performance of user-mode programs is generally not affected by the extra disk update.

Mirrored volumes are the only multipartition volume type supported for system and boot volumes. The reason for this is that the Windows boot code, including the MBR code and Ntldr, don't have the sophistication required to understand multipartition volumes—mirrored volumes are the exception because the boot code treats them as simple volumes, reading from the half of the mirror marked as the boot or system drive in the MBR-style partition table. Because the boot code doesn't modify the disk, it can safely ignore the other half of the mirror.

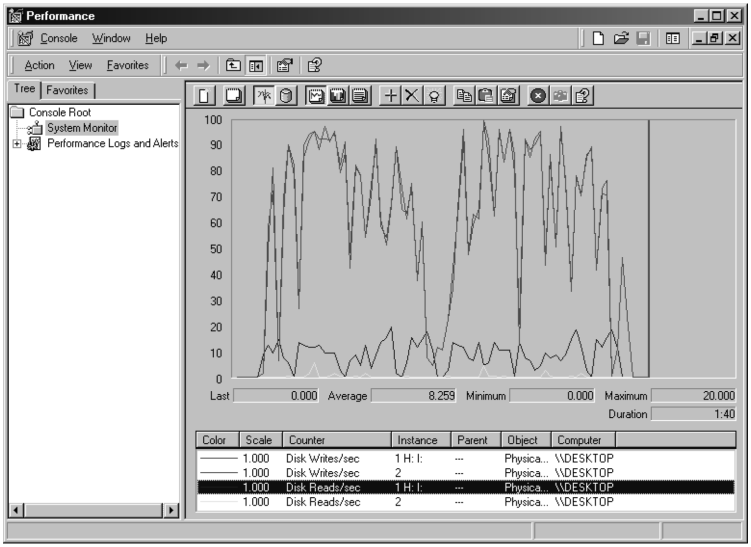
EXPERIMENT: Watching Mirrored Volume I/O Operations

Using the Windows Performance tool, you can verify that write operations directed at mirrored volumes copy to both disks that make up the mirror and that read operations, if relatively infrequent, occur primarily from one half of the volume. This experiment requires three hard disks and a Windows 2000 server or Windows Server 2003 system. If you don't have three disks or a server system, you can skip the experiment setup instructions and view the Performance tool screen shot in this experiment that demonstrates the experiment's results.

Use the Disk Management MMC snap-in to create a mirrored volume. To do this, perform the following steps:

1. Run Disk Management by starting Computer Management, expanding the Storage tree, and selecting Disk Management (or by inserting Disk Management as a snap-in in an MMC console).
2. Right-click on an unallocated space of a drive, and select Create Volume.
3. Follow the instructions in the Create Volume Wizard to create a simple volume. (Make sure there's enough room on another disk for a volume of the same size as the one you're creating.)
4. Right-click on the new volume, and select Add Mirror from the context menu.

Once you have a mirrored volume, run the Performance tool and add counters for the PhysicalDisk performance object for both disk instances that contain a partition belonging to the mirror. Select the Disk Reads/Sec and Disk Writes/Sec counters for each instance. Select a large directory from the third disk (the one that isn't part of the mirrored volume), and copy it to the mirrored volume. The Performance tool output window should look something like the following as the copy operation progresses.



Last 0.000 Average 8.259 Minimum 0.000 Maximum 20.000
Duration 1:40

Color	Scale	Counter	Instance	Parent	Object	Computer
—	1.000	Disk Writes/sec	1 H: I:	---	Physica...	\\DESKTOP
—	1.000	Disk Writes/sec	2	---	Physica...	\\DESKTOP
—	1.000	Disk Reads/sec	1 H: I:	---	Physica...	\\DESKTOP
—	1.000	Disk Reads/sec	2	---	Physica...	\\DESKTOP

The top two lines, which overlap throughout the timeline, are the Disk Writes/Sec counters for each disk. The bottom two lines are the Disk Reads/Sec lines. The screen shot reveals that the volume manager (in this case DMIO) is writing the copied file data to both halves of the volume but primarily reading from only one. This read behavior occurs because the number of outstanding I/O operations during the copy didn't warrant that the volume manager perform more aggressive read-operation load balancing.

RAID-5 Volumes

A *RAID-5 volume* is a fault tolerant variant of a regular striped volume. RAID-5 volumes implement *RAID level 5*. They are also known as *striped volumes with parity* because they are based on the striping approach taken by striped volumes. Fault tolerance is achieved by reserving the equivalent of one disk for storing parity for each stripe. Figure 10-13 is a visual representation of a RAID-5 volume.

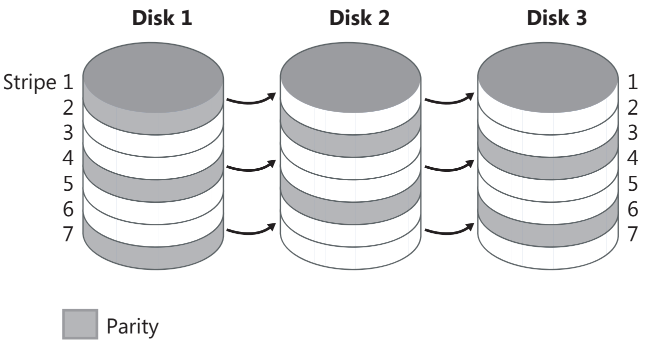


Figure 10-13 RAID-5 volume

In Figure 10-13, the parity for stripe 1 is stored on disk 1. It contains a byte-for-byte logical sum (XOR) of the first stripe on disks 2 and 3. The parity for stripe 2 is stored on disk 2, and the parity for stripe 3 is stored on disk 3. Rotating the parity across the disks in this way is an I/O optimization technique. Each time data is written to a disk, the parity bytes corresponding to the modified bytes must be recalculated and rewritten. If the parity were always written to the same disk, that disk would be busy continually and could become an I/O bottleneck.

Recovering a failed disk in a RAID-5 volume relies on a simple arithmetic principle: in an equation with n variables, if you know the value of $n - 1$ of the variables, you can determine the value of the missing variable by subtraction. For example, in the equation $x + y = z$, where z represents the parity stripe, the volume manager computes $z - y$ to determine the contents of x; to find y, it computes $z - x$. The volume manager uses similar logic to recover lost data. If a disk in a RAID-5 volume fails or if data on one disk becomes unreadable, the volume manager reconstructs the missing data by using the XOR operation (bitwise logical addition).

If disk 1 in Figure 10-13 fails, the contents of its stripes 2 and 5 are calculated by XORing the corresponding stripes of disk 3 with the parity stripes on disk 2. The contents of stripes 3 and 6 on disk 1 are similarly determined by XORing the corresponding stripes of disk 2 with the parity stripes on disk 3. At least three disks (or rather, three same-sized partitions on three disks) are required to create a RAID-5 volume.

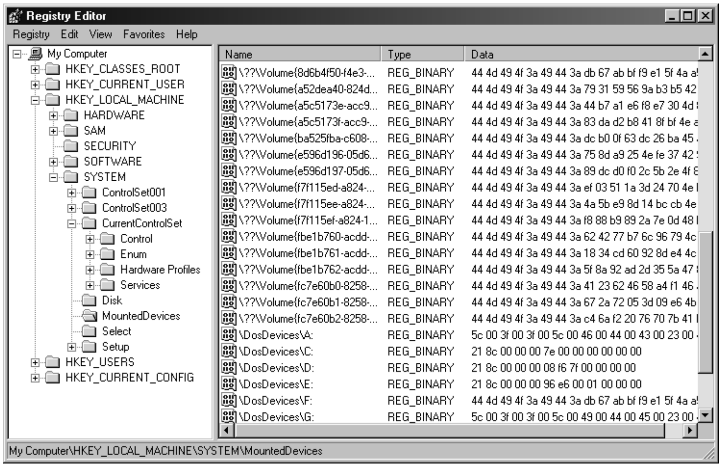
The Volume Namespace

Drive-letter assignment is an aspect of storage management that changed significantly from Windows NT 4 to Windows 2000. Even so, Windows includes support for migrating drive-letter assignments made in a Windows NT 4 installation that upgrades to Windows. Windows NT 4 drive-letter assignments are stored in HKLM\SYSTEM\Disk. After the upgrade procedure reads and stores the information in Windows-specific locations, the system no longer references the Disk key.

The Mount Manager

The Mount Manager device driver (Mountmgr.sys), assigns drive letters for dynamic disk volumes and basic disk volumes created after Windows is installed, CD-ROMs, floppies, and removable devices. Windows stores all drive-letter assignments under HKLM\SYSTEM\MountedDevices. If you look in the registry under that key, you'll see values with names such as \??\Volume{*X*} (where *X* is a GUID) and values such as \DosDevices\C:. Every volume has a volume name entry, but a volume doesn't necessarily have an assigned drive letter. Figure 10-14 shows the contents of an example Mount Manager registry key. Note that the MountedDevices key, like the Disk key in Windows NT 4, isn't included in a control set and so isn't protected by the last known good boot option. (See the section "Accepting the Boot and Last Known Good" in Chapter 4 for more information on control sets and the last known good boot option.)

The data that the registry stores in values for basic disk volume drive letters and volume names is the Windows NT 4–style disk signature and the starting offset of the first partition associated with the volume. The data that the registry stores in values for dynamic disk volumes includes the volume's DMIO-internal GUID. When the Mount Manager initializes during the boot process, it registers with the Windows Plug and Play subsystem so that it receives notification whenever either FtDisk or DMIO creates a volume. When the Mount Manager receives such a notification, it determines the new volume's GUID or disk signature and uses the GUID or signature as a guide to look in its internal database, which reflects the contents of the MountedDevices registry key. The Mount Manager then determines whether its internal database contains the drive-letter assignment. If the volume has no entry in the database the Mount Manager asks either FtDisk or DMIO (whichever created the volume) for a suggested drive-letter assignment and stores that in the database. FtDisk doesn't return suggestions and DMIO looks at the drive-letter hint in the volume's database entry.



Name	Type	Data
\??\Volume{8d6b4f50-f4e3-...	REG_BINARY	44 4d 49 4f 3a 49 44 3a db 67 ab bf f9 e1 5f 4a a
\??\Volume{a52dea40-824d...	REG_BINARY	44 4d 49 4f 3a 49 44 3a 79 31 59 56 9a b3 b5 42
\??\Volume{a5c5173e-acc9...	REG_BINARY	44 4d 49 4f 3a 49 44 3a 44 b7 a1 e6 f8 e7 30 4d
\??\Volume{a5c5173f-acc9-...	REG_BINARY	44 4d 49 4f 3a 49 44 3a 83 da d2 b8 41 8f bf 4e a
\??\Volume{ba525fba-c608-...	REG_BINARY	44 4d 49 4f 3a 49 44 3a dc b0 0f 63 dc 26 ba 45
\??\Volume{e596d196-05d6...	REG_BINARY	44 4d 49 4f 3a 49 44 3a 75 8d a9 25 4e fe 37 42
\??\Volume{e596d197-05d6...	REG_BINARY	44 4d 49 4f 3a 49 44 3a 89 dc d0 f0 2c 5b 2e 4f
\??\Volume{f7f115ed-a824-...	REG_BINARY	44 4d 49 4f 3a 49 44 3a ef 03 51 1a 3d 24 70 4e
\??\Volume{f7f115ee-a824-...	REG_BINARY	44 4d 49 4f 3a 49 44 3a 4a 5b e9 8d 14 bc cb 4e
\??\Volume{f7f115ef-a824-1...	REG_BINARY	44 4d 49 4f 3a 49 44 3a f8 88 b9 89 2a 7e 0d 48
\??\Volume{fbe1b760-acdd-...	REG_BINARY	44 4d 49 4f 3a 49 44 3a 62 42 77 b7 6c 96 79 4c
\??\Volume{fbe1b761-acdd-...	REG_BINARY	44 4d 49 4f 3a 49 44 3a 18 34 cd 60 92 8d e4 4c
\??\Volume{fbe1b762-acdd-...	REG_BINARY	44 4d 49 4f 3a 49 44 3a 5f 8a 92 ad 2d 35 5a 47
\??\Volume{fc7e60b0-8258-...	REG_BINARY	44 4d 49 4f 3a 49 44 3a 41 23 62 46 58 a4 f1 46
\??\Volume{fc7e60b1-8258-...	REG_BINARY	44 4d 49 4f 3a 49 44 3a 67 2a 72 05 3d 09 e6 4b
\??\Volume{fc7e60b2-8258-...	REG_BINARY	44 4d 49 4f 3a 49 44 3a c4 6a f2 20 76 70 7b 41
\DosDevices\A:	REG_BINARY	5c 00 3f 00 3f 00 5c 00 46 00 44 00 43 00 23 00
\DosDevices\C:	REG_BINARY	21 8c 00 00 00 7e 00 00 00 00 00 00
\DosDevices\D:	REG_BINARY	21 8c 00 00 00 08 f6 7f 00 00 00 00
\DosDevices\E:	REG_BINARY	21 8c 00 00 00 96 e6 00 01 00 00 00
\DosDevices\F:	REG_BINARY	44 4d 49 4f 3a 49 44 3a db 67 ab bf f9 e1 5f 4a a
\DosDevices\G:	REG_BINARY	5c 00 3f 00 3f 00 5c 00 49 00 44 00 45 00 23 00

My Computer\HKEY_LOCAL_MACHINE\SYSTEM\MountedDevices

Figure 10-14 Mounted devices listed in the Mount Manager's registry key

If no suggested drive-letter assignment exists for the volume, the Mount Manager uses the first unassigned drive letter (if one exists), defines a new assignment, creates a symbolic link for the assignment (for example, \Global??\D: on Windows XP and Windows Server 2003 or \??\D: on Windows 2000), and updates the MountedDevices registry key. If there are no available drive letters, no drive-letter assignment is made. At the same time, the Mount Manager creates a volume symbolic link (that is, \Global??\Volume{*X*}) that defines a new volume GUID if the volume doesn't already have one. This GUID is different from the volume GUIDs that DMIO uses internally.

Mount Points

Mount points let you link volumes through directories on NTFS volumes, which makes volumes with no drive-letter assignment accessible. For example, an NTFS directory that you've named C:\Projects could mount another volume (NTFS or FAT) that contains your project directories and files. If your project volume had a file you named \CurrentProject\Description.txt, you could access the file through the path C:\Projects\CurrentProject\Description.txt. What makes mount points possible is *reparse point* technology. (Reparse points are discussed in more detail in Chapter 12.)

A reparse point is a block of arbitrary data with some fixed header data that Windows associates with an NTFS file or directory. An application or the system defines the format and behavior of a reparse point, including the value of the unique reparse point tag that identifies reparse points belonging to the application or system and specifies the size and meaning of the data portion of a reparse point. (The data portion can be as large as 16 KB.) Reparse points store their unique tag in a fixed segment. Any application that implements a reparse point must supply a file system filter driver to watch for reparse-related return codes for file operations that execute on NTFS volumes, and the driver must take appropriate action when it detects the codes. NTFS returns a reparse status code whenever it processes a file operation and encounters a file or directory with an associated reparse point.

The Windows NTFS file system driver, the I/O manager, and the object manager all partly implement reparse point functionality. The object manager initiates pathname-parsing operations by using the I/O manager to interface with file system drivers. Therefore, the object manager must retry operations for which the I/O manager returns a reparse status code. The I/O manager implements pathname modification that mount points and other reparse points might require, and the NTFS file system driver must associate and identify reparse point data with files and directories. You can therefore think of the I/O manager as the reparse point file system filter driver for many Microsoft-defined reparse points.

An example of a reparse point application is a Hierarchical Storage Management (HSM) system, such as Windows Remote Storage Service (RSS) that Windows 2000 Server and Windows Server 2003 include, that uses reparse points to designate files that an administrator moves to offline tape storage. When a user tries to access an offline file, the HSM filter driver detects the reparse status code that NTFS returns, communicates with a user-mode service to retrieve the file from offline storage, deletes the reparse point from the file, and lets the file operation retry after the service retrieves the file. This is exactly how the RSS filter driver, Rsfilter.sys, uses reparse points.

If the I/O manager receives a reparse status code from NTFS and the file or directory for which NTFS returned the code isn't associated with one of a handful of built-in Windows reparse points, no filter driver claimed the reparse point. The I/O manager then returns an error to the object manager that propagates as a "file cannot be accessed by the system" error to the application making the file or directory access.

Mount points are reparse points that store a volume name (\Global??\Volume{*X*}) as the reparse data. When you use the Disk Management MMC snap-in to assign or remove path assignments for volumes, you're creating mount points. You can also create and display mount points by using the built-in command-line tool *Mountvol.exe* (\Windows\System32\Mountvol.exe).

The Mount Manager maintains the Mount Manager remote database on every NTFS volume in which the Mount Manager records any mount points defined for that volume. The database file, $MountMgrRemoteDatabase, resides in the NTFS root directory. Mount points move when a disk moves from one system to another and in dual-boot environments—that is, when booting between multiple Windows installations—because of the Mount Manager remote database's existence. NTFS also keeps track of mount points in the NTFS metadata file \$Extend\$Reparse. (NTFS doesn't make any of its metadata files available for viewing by applications.) NTFS stores mount-point information in the metadata file so that Windows can easily enumerate the mount points defined for a volume when a Windows application, such as Disk Management, requests mount-point definitions.

EXPERIMENT: Recursive Mount Points

This experiment uses Filemon from www.sysinternals.com to show the interesting behavior caused by a recursive mount point. A recursive mount point is a mount point that links to the same volume it's on. Performing a recursive directory listing on a recursive mount point produces file access traces that clearly demonstrate how NTFS treats mount points.

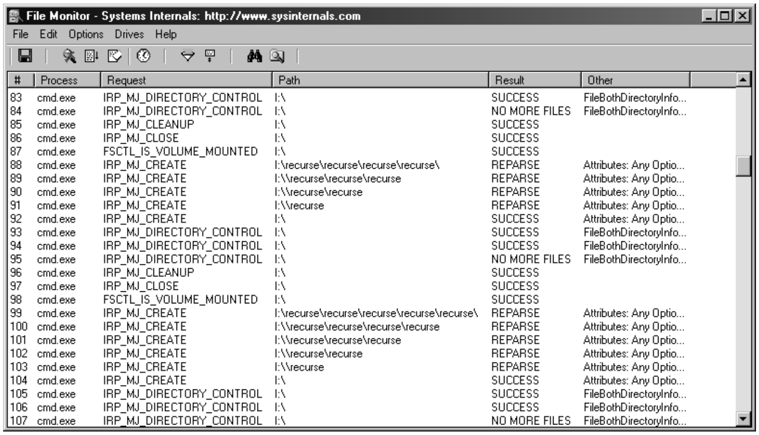
To create and view a mount point, perform the following steps:

1. Open a command prompt or Windows Explorer window, and create a directory on an NTFS drive named \Recurse.
2. In the Disk Management MMC snap-in, right-click on the volume and select Change Drive Letter And Path.
3. When the Add/Remove dialog box appears, enter the path to the directory you created (for example, *I:\Recurse*).
4. Start Filemon, and in the Drives menu, uncheck all the volumes except the one on which you created the mount point.

Now you're ready to generate a recursive mount-point trace. Open a command prompt, and execute the command *dir /s I:\Recurse*. If you look at all the file accesses that reference Recurse in the Filemon's trace of the subsequent file operations, you'll notice that the command prompt first accesses I:\Recurse, then I:\Recurse\Recurse, and so on, recursing deeper and deeper.

The application attempts to perform a directory listing at each level of the recursion, but whenever it encounters the mount point, it digs into it to try another directory listing. NTFS returns a reparse status code, which tells the object manager to back up one level and try again. Finally, when it gets back to the root directory, the application examines the file or directory that it had found deep in the mount-point recursion. The application never receives a reparse code because of the object manager–directed retry activity. The object manager processes the reparse codes as it receives them from NTFS when it performs directory lookups.

Filemon presents request types as their native IRP type, so a directory or file open appears as an *IRP_MJ_CREATE* request. A file or directory close is *IRP_MJ_CLOSE*, and a directory query is an *IRP_ MJ_DIRECTORY_CONTROL* request with *FileBothDirectoryInfo* in Filemon's Other column.


File Monitor - Systems Internals: http://www.sysinternals.com
File
Edit
Options
Drives
Help

#	Process	Request	Path	Result	Other
83	cmd.exe	IRP_MJ_DIRECTORY_CONTROL	I:\	SUCCESS	FileBothDirectoryInfo...
84	cmd.exe	IRP_MJ_DIRECTORY_CONTROL	I:\	NO MORE FILES	FileBothDirectoryInfo...
85	cmd.exe	IRP_MJ_CLEANUP	I:\	SUCCESS	
86	cmd.exe	IRP_MJ_CLOSE	I:\	SUCCESS	
87	cmd.exe	FSCTL_IS_VOLUME_MOUNTED	I:\	SUCCESS	
88	cmd.exe	IRP_MJ_CREATE	I:\recurse\recurse\recurse\recurse\	REPARSE	Attributes: Any Optio...
89	cmd.exe	IRP_MJ_CREATE	I:\\recurse\recurse\recurse	REPARSE	Attributes: Any Optio...
90	cmd.exe	IRP_MJ_CREATE	I:\\recurse\recurse	REPARSE	Attributes: Any Optio...
91	cmd.exe	IRP_MJ_CREATE	I:\\recurse	REPARSE	Attributes: Any Optio...
92	cmd.exe	IRP_MJ_CREATE	I:\	SUCCESS	Attributes: Any Optio...
93	cmd.exe	IRP_MJ_DIRECTORY_CONTROL	I:\	SUCCESS	FileBothDirectoryInfo...
94	cmd.exe	IRP_MJ_DIRECTORY_CONTROL	I:\	SUCCESS	FileBothDirectoryInfo...
95	cmd.exe	IRP_MJ_DIRECTORY_CONTROL	I:\	NO MORE FILES	FileBothDirectoryInfo...
96	cmd.exe	IRP_MJ_CLEANUP	I:\	SUCCESS	
97	cmd.exe	IRP_MJ_CLOSE	I:\	SUCCESS	
98	cmd.exe	FSCTL_IS_VOLUME_MOUNTED	I:\	SUCCESS	
99	cmd.exe	IRP_MJ_CREATE	I:\recurse\recurse\recurse\recurse\recurse\	REPARSE	Attributes: Any Optio...
100	cmd.exe	IRP_MJ_CREATE	I:\\recurse\recurse\recurse\recurse	REPARSE	Attributes: Any Optio...
101	cmd.exe	IRP_MJ_CREATE	I:\\recurse\recurse\recurse	REPARSE	Attributes: Any Optio...
102	cmd.exe	IRP_MJ_CREATE	I:\\recurse\recurse	REPARSE	Attributes: Any Optio...
103	cmd.exe	IRP_MJ_CREATE	I:\\recurse	REPARSE	Attributes: Any Optio...
104	cmd.exe	IRP_MJ_CREATE	I:\	SUCCESS	Attributes: Any Optio...
105	cmd.exe	IRP_MJ_DIRECTORY_CONTROL	I:\	SUCCESS	FileBothDirectoryInfo...
106	cmd.exe	IRP_MJ_DIRECTORY_CONTROL	I:\	SUCCESS	FileBothDirectoryInfo...
107	cmd.exe	IRP_MJ_DIRECTORY_CONTROL	I:\	NO MORE FILES	FileBothDirectoryInfo...

To prevent buffer overflows and infinite loops, both command prompt and Windows Explorer halt their recursion when the directory depth reaches 32 or the pathname exceeds 256 characters, whichever comes first.

Volume Mounting

Because Windows assigns a drive letter to a volume doesn't mean that the volume contains data that has been organized in a file system format that Windows recognizes. The volume-recognition process consists of a file system claiming ownership for a partition; the process takes place the first time the kernel, a device driver, or an application accesses a file or directory on a volume. After a file system driver signals its responsibility for a partition, the I/O manager directs all IRPs aimed at the volume to the owning driver. Mount operations in Windows consist of three components: file system driver registration, volume parameter blocks (VPBs), and mount requests.

Note In Windows Server 2003 Enterprise and Datacenter Edition, the automatic mounting of volumes as the Volume Manager reports their presence is disabled to prevent the system from aggressively mounting volumes attached to a System Area Network (SAN). You can use the Diskpart command-line utility, which Windows Server 2003 includes, to enable and disable automounting behavior, and the Mountvol utility, also included in the system, to mount volumes.

The I/O manager oversees the mount process and is aware of available file system drivers because all file system drivers register with the I/O manager when they initialize. The I/O manager provides the *IoRegisterFileSystem* function to local disk (rather than network) file system drivers for this registration. When a file system driver registers, the I/O manager stores a reference to the driver in a list that the I/O manager uses during mount operations.

Every device object contains a VPB data structure, but the I/O manager treats VPBs as meaningful only for volume device objects. A VPB serves as the link between a volume device object and the device object that a file system driver creates to represent a mounted file system instance for that volume. If a VPB's file system reference is empty, no file system has mounted the volume. The I/O manager checks a volume device object's VPB whenever an open API that specifies a filename or a directory name on a volume device object executes.

For example, if the mount manager assigns drive letter D to the second volume on a system, it creates a \Global??\D: symbolic link that resolves to the device object \Device\HarddiskVolume2. A Windows application that attempts to open the \Temp\Test.txt file on the D: drive specifies the name D:\Temp\Test.txt, which the Windows subsystem converts to \Global??\D:\Temp\Test.txt before invoking *NtCreateFile*, the kernel's file-open routine. *NtCreateFile* uses the object manager to parse the name, and the object manager encounters the \Device\ HarddiskVolume2 device object with the path \Temp\Test.txt still unresolved. At that point, the I/O manager checks to see whether \Device\HarddiskVolume2's VPB references a file system. If it doesn't, the I/O manager asks each registered file system driver via a mount request whether the driver recognizes the format of the volume in question as the driver's own.

EXPERIMENT: Looking at VPBs

You can look at the contents of a VPB by using the *!vpb* kernel debugger command. Because the VPB is pointed to by the device object for a volume, you must first locate a volume device object. To do this, you must dump a volume manager's driver object, locate a device object that represents a volume, and display the device object, which reveals its VPB field.

If your system has a dynamic disk, you can use the *!drvobj* driver object viewing command on the DMIO driver; otherwise, you need to specify the FtDisk driver. Here's an example:

```
kd> !drvobj ftdisk
Driver object (818aec50) is for:
 \Driver\Ftdisk
Driver Extension List: (id , addr)

Device Object list:
818a5290  817e96f0  817e98b0  817e9030
818a73b0  818a7810  8182d030
```

The *!drvobj* command lists the addresses of the device objects a driver owns. In this example, there are seven device objects. One of them represents the programmatic interface to the device driver, and the rest are volume device objects. Because the objects are listed in reverse order of the way that they were created and the driver creates the device driver interface object first, you know the first device object listed is that of a volume. Now execute the *!devobj* kernel debugger command on the volume device object address:

```
kd> !devobj 818a5290
Device object (818a5290) is for:
 HarddiskVolume6 \Driver\Ftdisk DriverObject 818aec50
Current Irp 00000000 RefCount 3 Type 00000007 Flags 00001050
Vpb 818a5da8 Dacl e1000384 DevExt 818a5348 DevObjExt 818a53f8 Dope 818a50a8
  DevNode 818a5ae8
ExtensionFlags (0xa8000000)  DOE_RAW_FDO, DOE_DESIGNATED_FDO
                             Unknown flags 0x08000000
AttachedDevice (Upper) 86b52b58 \Driver\VolSnap
Device queue is not busy.
```

The *!devobj* command shows the VPB field for the volume device object. (The device object shown is named HarddiskVolume6.) Now you're ready to execute the *!vpb* command:

```
kd> !vpb 818a5da8
Vpb at 0x818a5da8
Flags: 0x1 mounted
DeviceObject: 0x850dcac0
RealDevice:   0x818a5290
RefCount: 3
Volume Label:      BACKUP
```

The command reveals that the volume device object is mounted by a file system driver that has assigned the volume the name BACKUP. The *RealDevice* field in the VPB points back to the volume device object, and the *DeviceObject* field points to the mounted file system device object.

The convention followed by file system drivers for recognizing volumes mounted with their format is to examine the volume's boot record, which is stored in the first sector of the volume. Boot records for Microsoft file systems contain a field that stores a file system format type. File system drivers usually examine this field, and if it indicates a format they manage, they look at other information stored in the boot record. This information usually includes a file system name field and enough data for the file system driver to locate critical metadata files on the volume. NTFS, for example, will recognize a volume only if the type field is NTFS, the name field is "NTFS," and the critical metadata files described by the boot record are consistent.

If a file system driver signals affirmatively, the I/O manager fills in the VPB and passes the open request with the remaining path (that is, \Temp\Test.txt) to the file system driver. The file system driver completes the request by using its file system format to interpret the data that the volume stores. After a mount fills in a volume device object's VPB, the I/O manager hands subsequent open requests aimed at the volume to the mounted file system driver. If no file system driver claims a volume, Raw—a file system driver built into Ntoskrnl.exe—claims the volume and fails all requests to open files on that partition. Figure 10-15 shows a simplified example (that is, the figure omits the file system driver's interactions with the Windows cache and memory managers) of the path that I/O directed at a mounted volume follows.

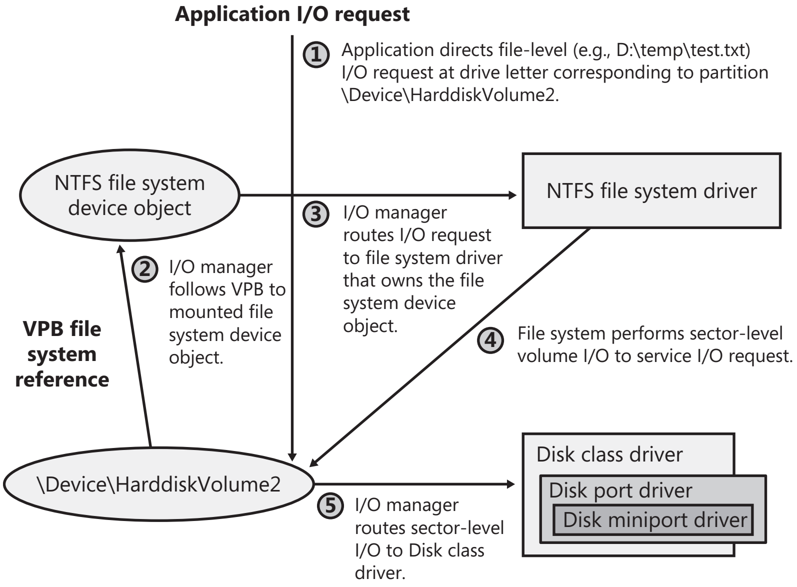

Application I/O request
1 Application directs file-level (e.g., D:\temp\test.txt) I/O request at drive letter corresponding to partition \Device\HarddiskVolume2.
NTFS file system device object
NTFS file system driver
3 I/O manager routes I/O request to file system driver that owns the file system device object.
2 I/O manager follows VPB to mounted file system device object.
VPB file system reference
4 File system performs sector-level volume I/O to service I/O request.
\Device\HarddiskVolume2
Disk class driver
Disk port driver
Disk miniport driver
5 I/O manager routes sector-level I/O to Disk class driver.

Figure 10-15 Mounted volume I/O flow

Instead of having every file system driver loaded, regardless of whether or not they have any volumes to manage, Windows tries to minimize memory usage by using a surrogate driver named File System Recognizer (\Windows\System32\Drivers\Fs_rec.sys) to perform preliminary file system recognition. File System Recognizer knows enough about each file system format that Windows supports to be able to examine a boot record and determine whether it's associated with a Windows file system driver. When the system boots, File System Recognizer registers as a file system driver, and when the I/O manager calls it during a file system mount operation for a new volume, File System Recognizer loads the appropriate file system driver if the boot record corresponds to one that isn't loaded. After loading a file system driver, File System Recognizer forwards the mount IRP to the driver and lets the file system driver claim ownership of the volume.

Aside from the boot volume, which a driver mounts while the kernel is initializing, file system drivers mount most volumes when the Chkdsk file system consistency-checking application runs during a boot sequence. The boot-time version of Chkdsk is a native application (as opposed to a Windows application) named Autochk.exe (\Windows\System32\Autochk.exe), and the Session Manager (\Windows\System32\Smss.exe) runs it because it is specified as a boot-run program in the HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\BootExecute value. Chkdsk accesses each drive letter to see whether the volume associated with the letter requires a consistency check.

One place in which mounting can occur more than once for the same disk is with removable media. Windows file system drivers respond to media changes by querying the disk's volume identifier. If they see the volume identifier change, the driver dismounts the disk and attempts to remount it.

Volume I/O Operations

File system drivers manage data stored on volumes but rely on volume managers to interact with storage drivers to transfer data to and from the disk or disks on which a volume resides. File system drivers obtain references to a volume manager's volume objects through the mount process and then send the volume manager requests via the volume objects. Applications can also send the volume manager requests, bypassing file system drivers, when they want to directly manipulate a volume's data. File-undelete programs are an example of applications that do this, and so is the DiskProbe utility that's part of the Windows resource kits.

Whenever a file system driver or application sends an I/O request to a device object that represents a volume, the Windows I/O manager routes the request (which comes in an IRP—a self-contained package) to the volume manager that created the target device object. Thus, if an application wants to read the boot sector of the second volume on the system (which is a simple volume in this example), it opens the device object \Device\HarddiskVolume2 and then sends the object a request to read 512 bytes starting at offset zero on the device. The I/O manager sends the application's request in the form of an IRP to the volume manager that owns the device object, notifying it that the IRP is directed at the HarddiskVolume2 device.

Because volumes are logical conveniences that Windows uses to represent contiguous areas on one or more physical disks, the volume manager must translate offsets that are relative to a volume to offsets that are relative to the beginning of a disk. If volume 2 consists of one partition that begins 4096 sectors into the disk, the volume manager would adjust the IRP's parameters to designate an offset with that value before passing the request to the disk class driver. The disk class driver uses a miniport driver to carry out physical disk I/O and read requested data into an application buffer designated in the IRP.

Some examples of a volume manager's operations will help clarify its role when it handles requests aimed at multipartition volumes. If a striped volume consists of two partitions, partition 1 and partition 2, that are represented by the device object \Device\HarddiskDmVolumes\

PhysicalDmVolumes\BlockVolume3, as Figure 10-16 shows, and an administrator has assigned drive letter D: to the stripe, the I/O manager defines the link \Global??\D: to reference \Device\HarddiskDmVolumes\ComputerNameDg0\Volume3, where *ComputerName* is the name of the computer. Recall from earlier that this link is also a symbolic link, and it points to the corresponding volume device object in the PhysicalDmVolumes directory (in this case, BlockVolume3). The DMIO device object intercepts file system disk I/O aimed at \Device\HarddiskDmVolumes\PhysicalDmVolumes\BlockVolume3, and the DMIO driver adjusts the request before passing it to the Disk class driver. The adjustment that DMIO makes configures the request to refer to the correct offset of the request's target stripe on either partition 1 or partition 2. If the I/O spans both partitions of the volume, DMIO must issue two subsidiary I/O requests, one aimed at each disk.

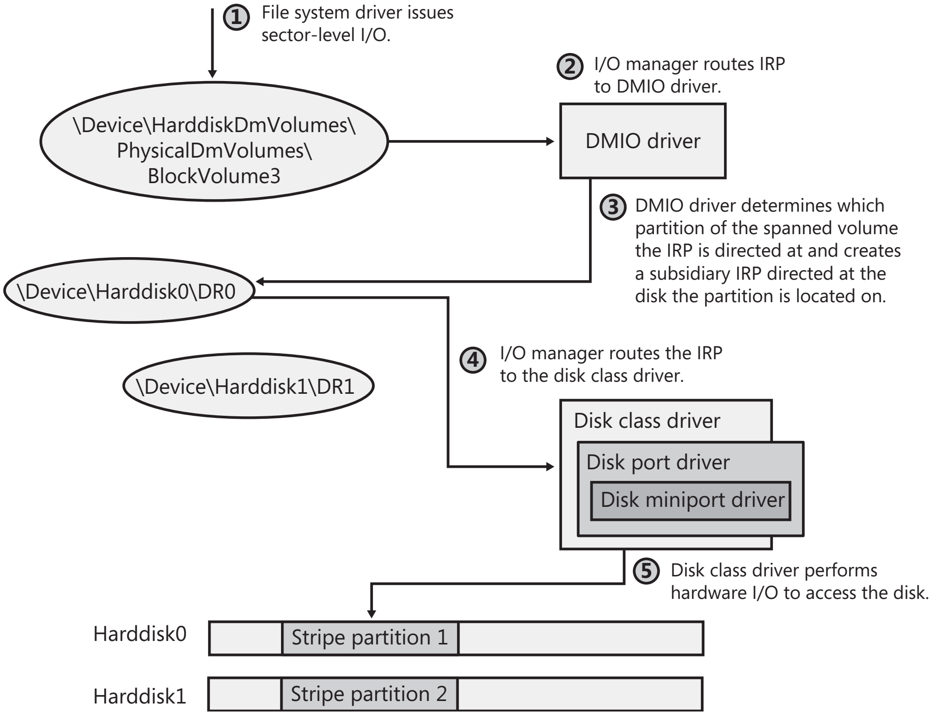

1 File system driver issues sector-level I/O.
2 I/O manager routes IRP to DMIO driver.
\Device\HarddiskDmVolumes\PhysicalDmVolumes\BlockVolume3
DMIO driver
3 DMIO driver determines which partition of the spanned volume the IRP is directed at and creates a subsidiary IRP directed at the disk the partition is located on.
\Device\Harddisk0\DR0
4 I/O manager routes the IRP to the disk class driver.
\Device\Harddisk1\DR1
Disk class driver
Disk port driver
Disk miniport driver
5 Disk class driver performs hardware I/O to access the disk.
Harddisk0
Stripe partition 1
Harddisk1
Stripe partition 2

Figure 10-16 DMIO I/O operations

In the case of writes to a mirrored volume, DMIO splits each request so that each half of the mirror receives the write operation. For mirrored reads, DMIO performs a read from half of a mirror, relying on the other half when a read operation fails.

Virtual Disk Service

A company that makes storage products such as RAID adapters, hard disks, or storage arrays has to implement custom applications for installing and managing their devices. The use of different management applications for different storage devices has obvious drawbacks from the perspective of system administration. These drawbacks include learning multiple interfaces and the inability to use standard Windows storage management tools to manage third-party storage devices.

Windows Server 2003 introduces the Virtual Disk Service (or VDS, located at \Windows\System32\Vds.exe), which provides a unified high-level storage interface so that administrators can manage storage devices from different vendors using the same user interfaces. VDS is shown in Figure 10-17. VDS exports a COM-based API that allows applications and scripts to create and format disks, and to view and manage hardware RAID adapters. For example, a utility can use the VDS API to query the list of physical disks that map to a RAID logical unit number (LUN). Windows disk management utilities, including the Disk Management MMC snap-in and the Diskpart and Diskraid (available in the Windows Server 2003 Deployment Kit) command-line tools, use VDS APIs.

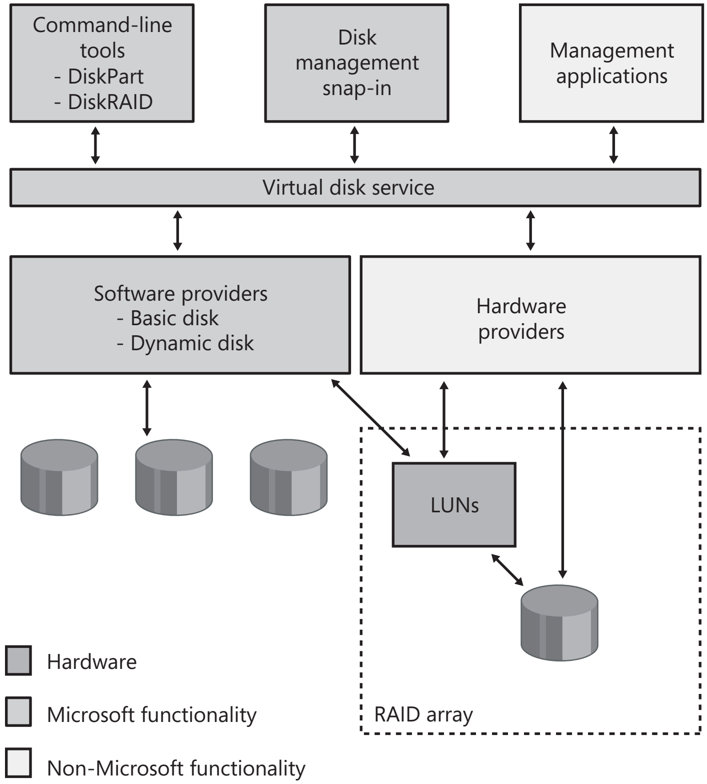


Figure 10-17 VDS service architecture

VDS supplies two interfaces, one for *software providers* and one for *hardware providers*:

- Software providers implement interfaces to high-level storage abstractions such as disks, disk partitions, and volumes. Examples of operations supported by these interfaces include creating, extending and deleting volumes, adding or breaking mirrors, formatting and assigning drive letters. VDS looks for registered software providers in HKLM\System\CurrentControlSet\Services\Vds\SoftwareProviders. Windows Server 2003 includes the VDS Dynamic Disk Provider (\Windows\System32\Vdsdyndr.dll) for interfacing to dynamic disks and the VDS Basic Provider (\Windows\System32\Vdsbas.dll) for interfacing to basic disks.
- Hardware vendors implement VDS hardware providers as DLLs that register under HKLM\System\CurrentControlSet\Services\Vds\HardwareProviders and that translate device-independent VDS commands into commands for their hardware. The hardware provider allows for management of a storage subsystem such as a hardware RAID array or adapter card and supported operations include creating, extending, deleting, masking and unmasking LUNs.

When an application initiates a connection to the VDS API and the VDS service isn't started, the Svchost process hosting the RPC service starts the VDS loader process (\Windows\System32\Vdsldr.exe), which starts the VDS service process and then exits. When the last connection to the VDS API closes, the VDS service process exits.

Volume Shadow Copy Service

A limitation of many backup utilities relates to open files. If an application has a file open for exclusive access, a backup utility can't gain access to the file's contents. Even if the backup utility can access an open file, the utility runs the risk of creating an inconsistent backup. Consider an application that updates a file at its beginning and then at its end. A backup utility saving the file during this operation might record an image of the file that reflects the start of the file before the application's modification and the end after the modification. If the file is later restored the application might deem the entire file corrupt because it might be prepared to handle the case where the beginning has been modified and not the end, but not vice versa. These two problems illustrate why most backup utilities skip open files altogether.

Windows XP introduces the Volume Shadow Copy service (\Windows\System32\Vssvc.exe), which allows the built-in backup utility to record consistent views of all files, including open ones. The Volume Shadow Copy service acts as the command center of an extensible backup core that enables independent software vendors (ISVs) to plug-in "writers" and providers. A writer is a software component that enables shadow-copy-aware application to receive freeze and thaw notifications to ensure that backup copies of their data files are internally consistent, whereas providers allow ISVs with unique storage schemes to integrate with the shadow copy service. For instance, an ISV with mirrored storage devices might define a shadow copy as the frozen half of a split mirrored volume. Figure 10-18 shows the relationship between the shadow copy service, writers, and providers.

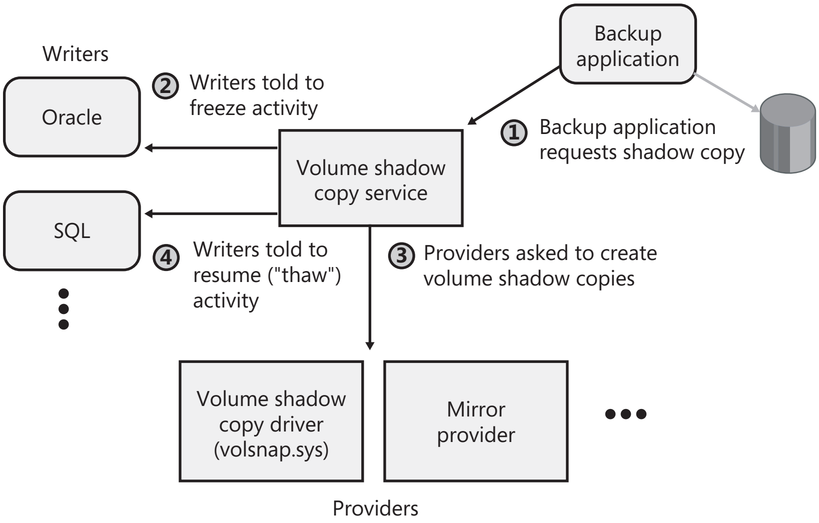

Writers
Oracle
SQL
Backup application
2 Writers told to freeze activity
1 Backup application requests shadow copy
Volume shadow copy service
4 Writers told to resume ("thaw") activity
3 Providers asked to create volume shadow copies
Volume shadow copy driver (volsnap.sys)
Mirror provider
Providers

Figure 10-18 Shadow copy service, writers, and providers

The Microsoft Shadow Copy Provider (\Windows\System32\Drivers\Volsnap.sys) is a provider that ships with Windows to implement software-based snapshots of volumes. It is a type of driver called a storage filter driver that layers between file system drivers and volume drivers (the drivers that present views of the disk sectors that represent a logical drive) so that it can see the I/O directed at a volume. When the backup utility starts a backup operation, it directs the Microsoft Shadow Copy Provider driver to create a volume shadow copy for the volumes that include files and directories being recorded. The driver freezes I/O to the volumes in question and creates a shadow volume for each. For example, if a volume's name in the Object Manager namespace is \Device\HarddiskVolume0, the shadow volume would have a name like \Device\HarddiskVolumeShadowCopy*N*, where *N* is a unique ID.

EXPERIMENT: Looking at Microsoft Shadow Copy Provider Filter Device Objects

You can see the Microsoft Shadow Copy Provider driver's device objects attached to each volume device on a Windows XP or Windows Server 2003 system in a kernel debugger. Every system has at least one volume, and the following command displays the device object of the first volume on a system:

```
0: kd> !devobj \device\harddiskvolume1
Device object (86b9daf8) is for:
 HarddiskVolume1 \Driver\Ftdisk DriverObject 86b6af38
Current Irp 00000000 RefCount 2 Type 00000007 Flags 00001050
Vpb 86b90008 Dacl e1000384 DevExt 86b9dbb0 DevObjExt 86b9dc98 Dope 86ba33e8 DevN
ode 86b71320
ExtensionFlags (0xa8000000)  DOE_RAW_FDO, DOE_DESIGNATED_FDO
                             Unknown flags 0x08000000
AttachedDevice (Upper) 86b687f8 \Driver\VolSnap Device queue is not busy.
```

The AttachedDevice field in the output for the *!devobj* command displays the address of any device object, and the name of the owning driver, attached (filtering) to the device object being examined. For volume device objects, you should see a device object belonging to the Volsnap driver, as in the example output.

Instead of opening files to backup on the original volume, the backup utility opens them on the shadow volume. A shadow volume represents a point-in-time view of a volume, so whenever the volume shadow copy driver sees a write operation directed at an original volume, it reads a copy of the sectors that will be overwritten into a paging file-backed memory section that's associated with the corresponding shadow volume. It services read operations directed at the shadow volume of modified sectors from this memory section, and it services reads to nonmodified areas by reading from the original volume. Because the backup utility won't save the paging file or the contents of the system-managed \System Volume Information directory located on every volume, the snapshot driver uses the defragmentation API to determine the location of these files and directories and does not record changes to them. By relying on the shadow copy facility, the backup utility in Windows XP and Windows Server 2003 overcomes both of the backup problems related to open files.

Figure 10-19 demonstrates the behavior of applications accessing a volume and a backup application accessing the volume's shadow volume. When an application writes to a sector after the snapshot time the Volsnap driver makes a backup copy, like it has for sectors a, b, and c of volume C: in the figure. Subsequently, when an application reads from sector c Volsnap directs the read to volume C:, but when an the backup application reads from sector c Volsnap reads the sector from the snapshot. When a read occurs for any unmodified sector, such as d, Volsnap routes the read to the volume.

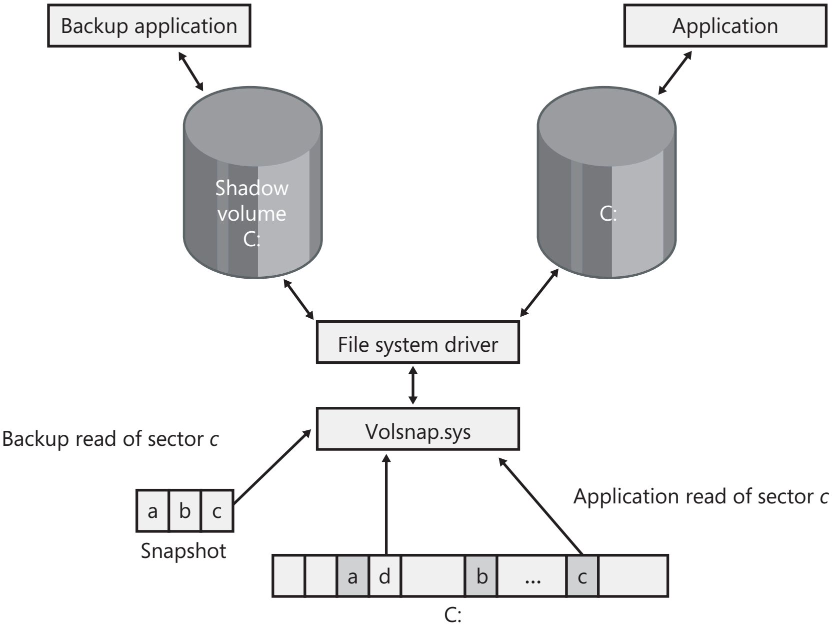
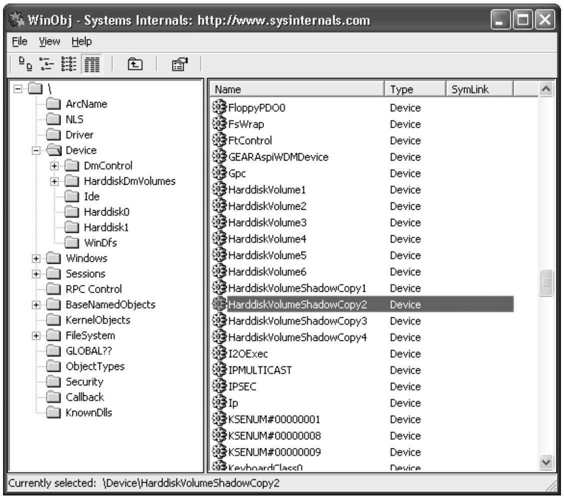


Figure 10-19 Volsnap operation

EXPERIMENT: Viewing Shadow Volume Device Objects

You can see the existence of shadow volume device objects in the object manager namespace by running the Windows backup application (in the System Tools in the Accessories folder of the start menu) and selecting enough backup data to cause the backup process to take long enough for you to run Winobj and see the objects in the \Device subdirectory.



```
\
  ArcName
  NLS
  Driver
  Device
    DmControl
    HarddiskDmVolumes
    Ide
    Harddisk0
    Harddisk1
    WinDfs
  Windows
  Sessions
  RPC Control
  BaseNamedObjects
  KernelObjects
  FileSystem
  GLOBAL??
  ObjectTypes
  Security
  Callback
  KnownDlls
```

Name	Type	SymLink
FloppyPDO0	Device	
FsWrap	Device	
FtControl	Device	
GEARAspiWDMDevice	Device	
Gpc	Device	
HarddiskVolume1	Device	
HarddiskVolume2	Device	
HarddiskVolume3	Device	
HarddiskVolume4	Device	
HarddiskVolume5	Device	
HarddiskVolume6	Device	
HarddiskVolumeShadowCopy1	Device	
HarddiskVolumeShadowCopy2	Device	
HarddiskVolumeShadowCopy3	Device	
HarddiskVolumeShadowCopy4	Device	
I2OExec	Device	
IPMULTICAST	Device	
IPSEC	Device	
Ip	Device	
KSENUM#00000001	Device	
KSENUM#00000008	Device	
KSENUM#00000009	Device	
KeyboardClass0	Device	

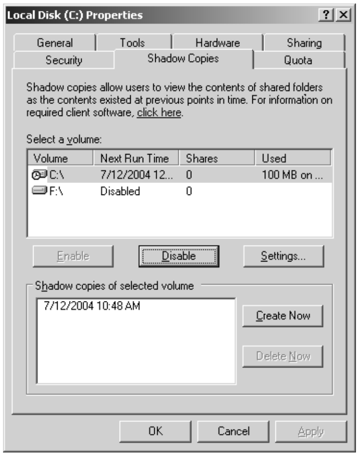
Currently selected: \Device\HarddiskVolumeShadowCopy2

File system drivers must handle two shadow copy–related I/O control requests (IOCTL) to help ensure consistent snapshots: IOCTL_VOLSNAP_FLUSH_AND_HOLD_WRITES and IOCTL_VOLSNAP_RELEASE_WRITES. Their names are self-explanatory. The shadow copy API sends the IOCTLs to the logical drives for which snapshots are being taken so that all modifications initiated before the snapshot have completed when the shadow copy is taken, making the file data recorded from a shadow copy consistent in time.

Shadow Copies for Shared Folders

Windows Server 2003 takes advantage of Volume Shadow Copy to provide a feature that lets end users access backup versions of volumes so that they can recover old versions of files and folders that they might have deleted or changed. The feature alleviates the burden on systems administrators, who would otherwise have to load backup media and access previous versions on behalf of end users.

The properties dialog box for a volume on Windows Server 2003 system includes a tab named Shadow Copies, where an administrator can enable scheduled snapshots of volumes as seen in the following screen shot. Administrators can also limit the amount of space consumed by snapshots so that the system deletes old snapshots to honor space constraints.



Shadow copies allow users to view the contents of shared folders as the contents existed at previous points in time. For information on required client software, click here.

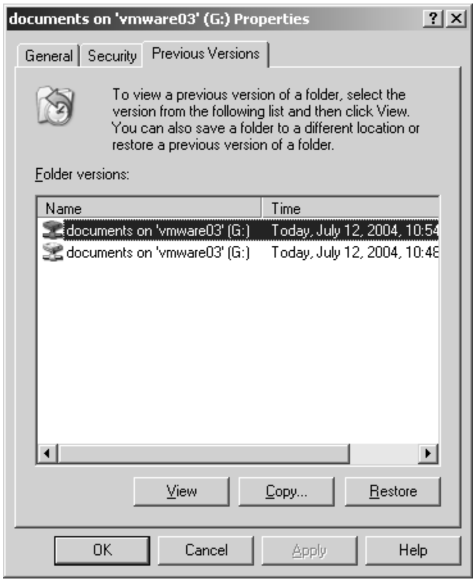
Select a volume:

Volume	Next Run Time	Shares	Used
C:\	7/12/2004 12...	0	100 MB on ...
F:\	Disabled	0	

Shadow copies of selected volume

7/12/2004 10:48 AM

Client systems that want to take advantage of Shadow Copies for Shared Folders must install an Explorer extension called the Previous Versions Client that ships with Windows Server 2003 under the \Windows\System32\Clients\Twclient directory and that Microsoft makes available as a download from its Web site (the extension comes installed with Windows XP Service Pack 2 and higher). When a client Windows system that has the Shadow Copy Explorer extension installed maps a share from a folder on a volume for which snapshots exist, a tab named Previous Versions appears on the Properties dialog box for folders and files on the share. The Previous Versions tab shows a list of snapshots that exist on the server, allowing the user to view or copy a file or folder's data as it existed in a previous snapshot.



To view a previous version of a folder, select the version from the following list and then click View. You can also save a folder to a different location or restore a previous version of a folder.

Folder versions:

Name	Time
documents on 'vmware03' (G:)	Today, July 12, 2004, 10:54
documents on 'vmware03' (G:)	Today, July 12, 2004, 10:48

Conclusion

In this chapter, we've reviewed the on-disk organization, components, and operation of Windows disk storage management. In the next chapter, we delve into the cache manager, an executive component integral to the operation of file system drivers that mount the volume types presented in this chapter.

Chapter 11

Cache Manager

The cache manager is a set of kernel-mode functions and system threads that cooperate with the memory manager to provide data caching for all Windows file system drivers (both local and network). In this chapter, we'll explain how the cache manager, including its key internal data structures and functions, works; how it is sized at system initialization time; how it interacts with other elements of the operating system; and how you can observe its activity through performance counters. We'll also describe the five flags on the Windows *CreateFile* function that affect file caching.

> **Note** None of the cache manager's internal functions are outlined in this chapter beyond the depth required to explain how the cache manager works. The programming interfaces to the cache manager are documented in the Windows Installable File System (IFS) kit. For more information about the IFS kit, see *http://www.microsoft.com/whdc/devtools/ifskit*.

Key Features of the Cache Manager

The cache manager has several key features:

- Supports all file system types (both local and network), thus removing the need for each file system to implement its own cache management code
- Uses the memory manager to control what parts of what files are in physical memory (trading off demands for physical memory between user processes and the operating system)
- Caches data on a virtual block basis (offsets within a file)—in contrast to many caching systems, which cache on a logical block basis (offsets within a disk volume)—allowing for intelligent read-ahead and high-speed access to the cache without involving file system drivers. (This method of caching, called *fast I/O*, is described later in this chapter.)
- Supports "hints" passed by applications at file open time (such as random vs. sequential access, temporary file creation, and so on)
- Supports recoverable file systems (for example, those that use transaction logging) to recover data after a system failure

Although we'll talk more throughout this chapter about how these features are used in the cache manager, in this section we'll introduce you to the concepts behind these features.

Single, Centralized System Cache

Some operating systems rely on each individual file system to cache data, a practice that results either in duplicated caching and memory management code in the operating system or in limitations on the kinds of data that can be cached. In contrast, Windows offers a centralized caching facility that caches all externally stored data, whether on local hard disks, floppy disks, network file servers, or CD-ROMs. Any data can be cached, whether it's user data streams (the contents of a file and the ongoing read and write activity to that file) or file system *metadata* (such as directory and file headers). As you'll discover in this chapter, the method Windows uses to access the cache depends on the type of data being cached.

The Memory Manager

One unusual aspect of the cache manager is that it never knows how much cached data is actually in physical memory. This statement might sound strange because the purpose of a cache is to keep a subset of frequently accessed data in physical memory as a way to improve I/O performance. The reason the cache manager doesn't know how much data is in physical memory is that it accesses data by mapping views of files into system virtual address spaces, using standard *section objects* (*file mapping objects* in Windows API terminology). (Section objects are the basic primitive of the memory manager and are explained in detail in Chapter 7.) As addresses in these mapped views are accessed, the memory manager pages in blocks that aren't in physical memory. And when memory demands dictate, the memory manager pages data out of the cache and back to the files that are open in (mapped into) the cache.

By caching on the basis of a virtual address space using mapped files, the cache manager avoids generating read or write I/O request packets (IRPs) to access the data for files it's caching. Instead, it simply copies data to or from the virtual addresses where the portion of the cached file is mapped and relies on the memory manager to fault in (or out) the data into (or out of) memory as needed. This process allows the memory manager to make global trade-offs on how much memory to give to the system cache vs. how much to give to the user processes. (The cache manager also initiates I/O, such as lazy writing, which is described later in this chapter; however, it calls the memory manager to write the pages.) Also, as you'll learn in the next section, this design makes it possible for processes that open cached files to see the same data as do processes that are mapping the same files into their user address spaces.

Cache Coherency

One important function of a cache manager is to ensure that any process accessing cached data will get the most recent version of that data. A problem can arise when one process opens a file (and hence the file is cached) while another process maps the file into its address space directly (using the Windows *MapViewOfFile* function). This potential problem doesn't occur under Windows because both the cache manager and the user applications that map files into their address spaces use the same memory management file mapping services. Because the memory manager guarantees that it has only one representation of each unique mapped file (regardless of the number of section objects or mapped views), it maps all views of a file (even if they overlap) to a single set of pages in physical memory, as shown in Figure 11-1. (For more information on how the memory manager works with mapped files, see Chapter 7.)

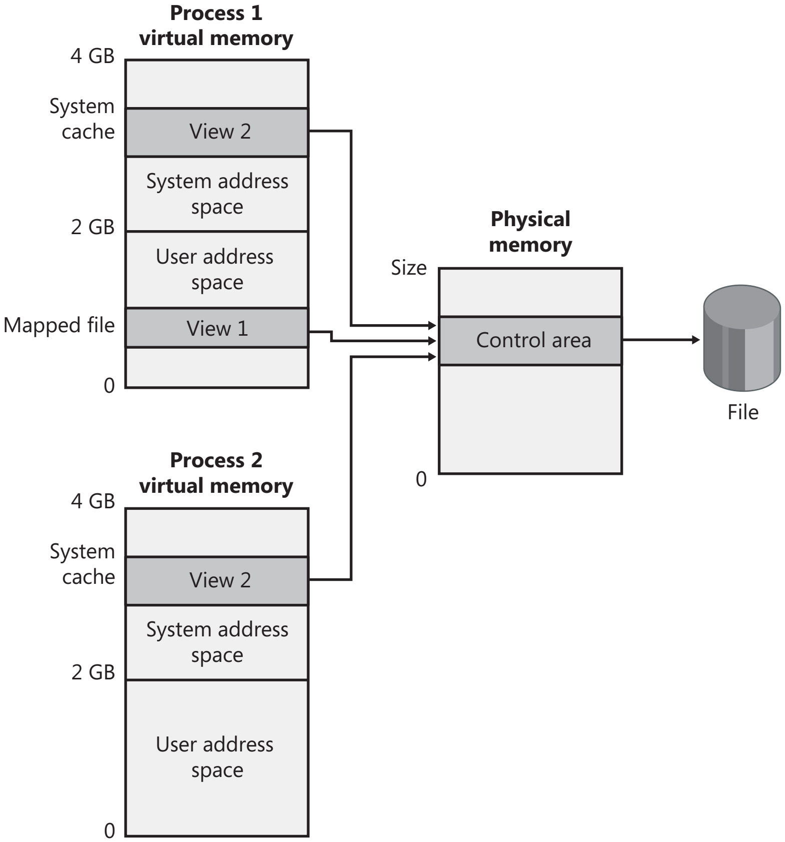


Figure 11-1 Coherent caching scheme

So, for example, if Process 1 has a view (View 1) of the file mapped into its user address space and Process 2 is accessing the same view via the system cache, Process 2 will see any changes that Process 1 makes as they're made, not as they're flushed. The memory manager won't flush *all* user-mapped pages—only those that it knows have been written to (because they have the modified bit set). Therefore, any process accessing a file under Windows always sees the most up-to-date version of that file, even if some processes have the file open through the I/O system and others have the file mapped into their address space using the Windows file mapping functions.

Note Cache coherency is somewhat more difficult for network redirectors than for local file systems because network redirectors must implement additional flushing and purge operations to ensure cache coherency when accessing network data. See Chapter 12 for a description of opportunistic locking, the Windows distributed cache coherency mechanism.

Virtual Block Caching

Many operating system cache managers (including Novell NetWare, OpenVMS, and older UNIX systems) cache data on the basis of *logical blocks*. With this method, the cache manager keeps track of which blocks of a disk partition are in the cache. The Windows cache manager, in contrast, uses a method known as *virtual block caching*, in which the cache manager keeps track of which parts of which *files* are in the cache. The cache manager is able to monitor these file portions by mapping 256-KB views of files into system virtual address spaces, using special system cache routines located in the memory manager. This approach has the following key benefits:

- It opens up the possibility of doing intelligent read-ahead; because the cache tracks which parts of which files are in the cache, it can predict where the caller might be going next.
- It allows the I/O system to bypass going to the file system for requests for data that is already in the cache (fast I/O). Because the cache manager knows which parts of which files are in the cache, it can return the address of cached data to satisfy an I/O request without having to call the file system.

Details of how intelligent read-ahead and fast I/O work are provided later in this chapter.

Stream-Based Caching

The cache manager is also designed to do *stream caching*, as opposed to file caching. A *stream* is a sequence of bytes within a file. Some file systems, such as NTFS, allow a file to contain more than one stream; the cache manager accommodates such file systems by caching each stream independently. NTFS can exploit this feature by organizing its master file table (described in Chapter 12) into streams and by caching these streams as well. In fact, although the cache manager might be said to cache files, it actually caches streams (all files have at least one stream of data) identified by both a filename and, if more than one stream exists in the file, a stream name.

Recoverable File System Support

Recoverable file systems such as NTFS are designed to reconstruct the disk volume structure after a system failure. This capability means that I/O operations in progress at the time of a system failure must be either entirely completed or entirely backed out from the disk when the system is restarted. Half-completed I/O operations can corrupt a disk volume and even render an entire volume inaccessible. To avoid this problem, a recoverable file system maintains a log file in which it records every update it intends to make to the file system structure (the file system's metadata) before it writes the change to the volume. If the system fails, interrupting volume modifications in progress, the recoverable file system uses information stored in the log to reissue the volume updates.

> **Note** The term *metadata* applies only to changes in the file system structure: file and directory creation, renaming, and deletion.

To guarantee a successful volume recovery, every log file record documenting a volume update must be completely written to disk before the update itself is applied to the volume. Because disk writes are cached, the cache manager and the file system must work together to ensure that the following actions occur, in sequence:

1. The file system writes a log file record documenting the volume update it intends to make.
2. The file system calls the cache manager to flush the log file record to disk.
3. The file system writes the volume update to the cache—that is, it modifies its cached metadata.
4. The cache manager flushes the altered metadata to disk, updating the volume structure. (Actually, log file records are batched before being flushed to disk, as are volume modifications.)

When a file system writes data to the cache, it can supply a *logical sequence number* (LSN) that identifies the record in its log file, which corresponds to the cache update. The cache manager keeps track of these numbers, recording the lowest and highest LSNs (representing the oldest and newest log file records) associated with each page in the cache. In addition, data streams that are protected by transaction log records are marked as "no write" by NTFS so that the mapped page writer won't inadvertently write out these pages before the corresponding log records are written. (When the mapped page writer sees a page marked this way, it moves the page to a special list that the cache manager then flushes at the appropriate time, such as when lazy writer activity takes place.)

When it prepares to flush a group of dirty pages to disk, the cache manager determines the highest LSN associated with the pages to be flushed and reports that number to the file system. The file system can then call the cache manager back, directing it to flush log file data up to the point represented by the reported LSN. *After* the cache manager flushes the log file up to that LSN, it flushes the corresponding volume structure updates to disk, thus ensuring that it records what it's going to do before actually doing it. These interactions between the file system and the cache manager guarantee the recoverability of the disk volume after a system failure.

Cache Virtual Memory Management

Because the Windows system cache manager caches data on a virtual basis, it is given a region of system virtual address spaces to manage (instead of a region of physical memory). The cache manager then divides each address space region into 256-KB slots called *views*, as shown in Figure 11-2. (For a detailed description of the layout of system space, see Chapter 7.)

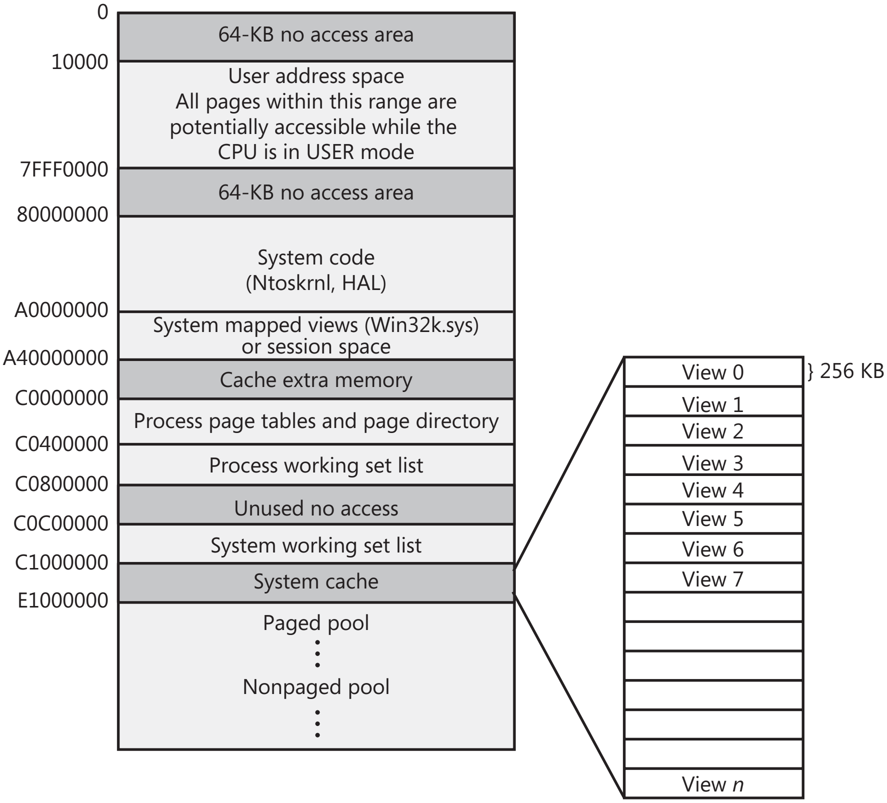


Figure 11-2 System cache address space

At a file's first I/O (read or write) operation, the cache manager maps a 256-KB view of the 256-KB-aligned region of the file that contains the requested data into a free slot in the system cache address space. For example, if 10 bytes starting at an offset of 300,000 bytes were read into a file, the view that would be mapped would begin at offset 262144 (the second 256-KB-aligned region of the file) and extend for 256 KB.

The cache manager maps views of files into slots in the cache's address space on a round-robin basis, mapping the first requested view into the first 256-KB slot, the second view into the second 256-KB slot, and so forth, as shown in Figure 11-3. In this example, File B was mapped first, File A second, and File C third, so File B's mapped chunk occupies the first slot in the cache. Notice that only the first 256-KB portion of File B has been mapped, which is due to the fact that only part of the file has been accessed and because although File C is only 100 KB (and thus smaller than one of the views in the system cache), it requires its own 256-KB slot in the cache.

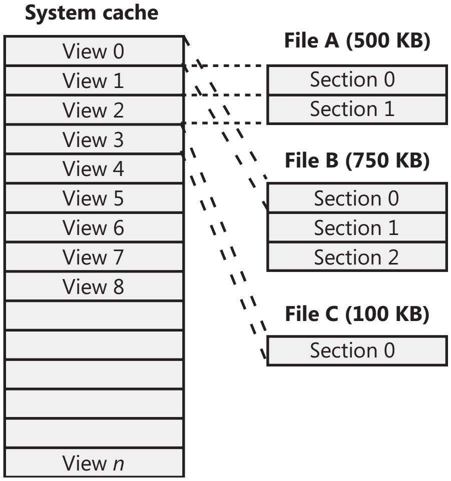


Figure 11-3 Files of varying sizes mapped into the system cache

The cache manager guarantees that a view is mapped as long as it's active (although views can remain mapped after they become inactive). A view is marked active, however, only during a read or write operation to or from the file. Unless a process opens a file by specifying the FILE_ FLAG_RANDOM_ACCESS flag in the call to *CreateFile*, the cache manager unmaps inactive views of a file as it maps new views for the file. Pages for unmapped views are sent to the standby or modified lists (depending on whether they have been changed), and because the memory manager exports a special interface for the cache manager, the cache manager can direct the pages to be placed at the end or front of these lists. Pages that correspond to views of files opened with the FILE_FLAG_SEQUENTIAL_SCAN flag are moved to the front of the lists, whereas all others are moved to the end. This scheme encourages the reuse of pages belonging to sequentially read files and specifically prevents a large file copy operation from affecting more than a small part of physical memory.

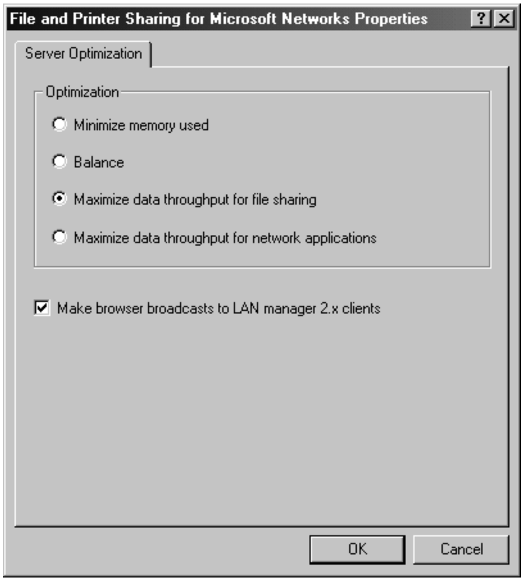
If the cache manager needs to map a view of a file and there are no more free slots in the cache, it will unmap the least recently mapped inactive view and use that slot. If no views are available, an I/O error is returned, indicating that insufficient system resources are available to perform the operation. Given that views are marked active only during a read or write operation, however, this scenario is extremely unlikely because thousands of files would have to be accessed simultaneously for this situation to occur.

Cache Size

In the following sections, we'll explain how Windows computes the size of the system cache, both virtually and physically. As with most calculations related to memory management, the size of the system cache depends on a number of factors, including memory size and which version of Windows is running.

LargeSystemCache

As you'll see, the registry value HKLM\SYSTEM\CurrentControlSet\Control\SessionManager\Memory Management\LargeSystemCache influences both the virtual and physical sizes of the cache. The value is 0 by default on Windows 2000 Professional and Windows XP and 1 on Windows Server systems. You can adjust this value through the GUI on Windows 2000 Server systems by modifying the properties of the file server service. (Bring up the properties for a network connection, and double-click File And Printer Sharing For Microsoft Networks.) Even though this service exists on Windows 2000 Professional, its parameters can't be adjusted. Figure 11-4 shows the dialog box you use on Windows 2000 Server systems when modifying the amount of memory allocated for local and network applications in the Server network service.



○ Minimize memory used
○ Balance
● Maximize data throughput for file sharing
○ Maximize data throughput for network applications

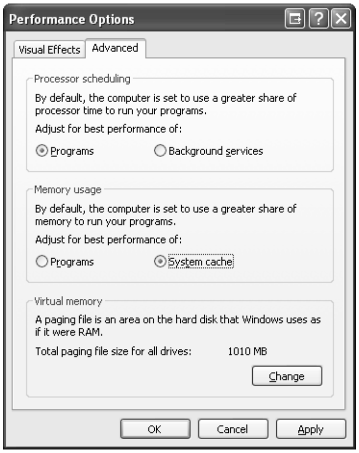
☑ Make browser broadcasts to LAN manager 2.x clients

Figure 11-4 File And Printer Sharing For Microsoft Networks Properties dialog box, which is used to modify the properties of the Windows 2000 Server network service

The setting shown in Figure 11-4, Maximize Data Throughput For File Sharing, is the default for Server systems running with Terminal Services installed—the LargeSystemCache value is 1.

Choosing any of the other settings will set the LargeSystemCache value to 0. Although each of the four settings in the Optimization section of the File And Printer Sharing For Microsoft Networks Properties dialog box affect the behavior of the system cache, they also modify the behavior of the file server service.

On Windows XP and Windows Server 2003, you can modify LargeSystemCache in the Performance Options dialog box that you access by clicking the Settings button on the Advanced page of the System control panel applet, as shown in Figure 11-5. Selecting System Cache in the Memory Usage area of the Advanced tab sets the LargeSystemCache value to 1, while choosing Programs sets it to 0.



By default, the computer is set to use a greater share of processor time to run your programs.

Adjust for best performance of:

● Programs ○ Background services

By default, the computer is set to use a greater share of memory to run your programs.

Adjust for best performance of:

○ Programs ● System cache

A paging file is an area on the hard disk that Windows uses as if it were RAM.

Total paging file size for all drives: 1010 MB

Figure 11-5 LargeSystemCache setting on Windows XP and Windows Server 2003

Cache Virtual Size

The virtual size of the system cache is a function of the amount of physical memory installed. The default size is 64 MB. If the system has more than 4032 pages (16 MB) of physical memory, the cache size is set to 128 MB plus 64 MB for each additional 4 MB of physical memory. Using this algorithm, the virtual size of the system cache for a computer with 64 MB of physical memory will be:

*128 MB + (64 MB - 16 MB) / 4 MB * 64 MB = 896 MB*

Table 11-1 shows the minimum and maximum virtual size of the system cache on different systems, along with the start and end addresses. If it's running on an x86 system and the system calculates a cache virtual size that is greater than 512 MB, the virtual size is limited to 512 MB unless the LargeSystemCache registry value is 1, in which case the cache is assigned up to 960 MB of virtual memory that includes an additional address range known as *cache extra memory*. The benefit of assigning more virtual memory to the cache is that, in general, it results in fewer view unmaps and remap operations as different files and different views of files are accessed.

Table 11-1 Size and Location of System Data Cache

Platform	Address Range	Minimum/Maximum Virtual Size
x86 2-GB system space	0xC1000000–E0FFFFFF, 0xA4000000–BFFFFFFF	64–960 MB
x86 1-GB system space	0xC1000000–DBFFFFFF	64–432 MB
x86 1-GB system space with Terminal Services	0xC1000000–DCFFFFFF	64–448 MB
x64	0xFFFFF98000000000–FFFFFA8000000000	64 MB–1024 GB
IA64	0xE000000600000000–E000010600000000	64 MB–1024 GB

Table 11-2 lists the system variables that contain the virtual size and address of the system cache.

Table 11-2 System Variables for the Virtual Size and Address of the System Cache

System Variable	Description
MmSystemCacheStart	Starting virtual address of cache
MmSystemCacheEnd	Ending virtual address of cache
MiSystemCacheStartExtra	Starting virtual address of cache extra memory if cache size is greater than 512 MB (x86 systems only)
MiSystemCacheEndExtra	Ending virtual address of cache extra memory if cache size is greater than 512 MB (x86 systems only)
MmSizeOfSystemCacheInPages	Virtual size of cache in pages

EXPERIMENT: Viewing the Cache Virtual Size

The cache's virtual size isn't revealed through any performance counters, so the only way to know its value is to examine the kernel variable that stores it, *MmSizeOfSystemCacheInPages*:

```
kd> dd MmSizeOfSystemCacheInPages l 1
80539a8c  00020000
```

In this example, taken from an x86 system running Windows XP with LargeSystemCache having a value of 0, the cache is 0x20000 virtual pages in size. Because a page on x86 is 4 KB, the virtual cache is assigned 512 MB of the 2 GB of the system's virtual address space.

Cache Working Set Size

As mentioned earlier, one of the key differences in the design of the cache manager from that of other operating systems is the delegation of physical memory management to the global memory manager. Because of this, the existing code that handles working set expansion and trimming as well as managing the modified and standby list is also used to control the size of the system cache, dynamically balancing demands for physical memory between processes and the operating system.

The system cache doesn't have its own working set but rather shares a single system set that includes cache data, paged pool, pageable Ntoskrnl code, and pageable driver code. As explained in the section "System Working Set" in Chapter 7, this single working set is called internally the *system cache working set* even though the system cache is just one of the components that contribute to it. For the purposes of this book, we'll refer to this working set simply as the *system working set.* Also explained in Chapter 7 is the fact that if the LargeSystemCache registry value is 1, the memory manager favors the system working set over that of processes running on the system.

You can examine the physical size of the system cache compared to that of the total system working set as well as page fault information on the system working set by examining the performance counters or system variables listed in Table 11-3.

Table 11-3 System Variables for the Physical Size of the System Cache and Page Fault Information

Performance Counter (in bytes)	System Variable (in pages)	Description
Memory: System Cache Resident Bytes	*MmSystemCachePage*	Physical memory consumed by the system cache.
Memory: Cache Bytes	*MmSystemCacheWs.WorkingSetSize*	Total size of the system working set (including the cache, paged pool, pageable code, and system mapped views). This is *not* the size of the cache (as the name implies)!
Memory: Cache Bytes Peak	*MmSystemCacheWs.Peak*	Peak system working set size.
Memory: Cache Faults/Sec	*MmSystemCacheWs.PageFaultCount*	Page faults in the system working set (not just the cache).

EXPERIMENT: Looking at the Cache's Working Set

The *!filecache* debugger command dumps information about the physical memory the cache is using, the current and peak working set sizes, the number of valid pages associated with views, and the names of files mapped into views, where applicable, as you can see in the following output. (File system drivers cache metadata, such as directory structures and volume bitmaps, by using unnamed file streams.)

```
lkd> !filecache
***** Dump file cache******
  Reading and sorting VACBs ...
  Removed 657 nonactive VACBs, processing 1389 active VACBs ...
File Cache Information
  Current size 132308 kb
  Peak size    145764 kb
  1389 Control Areas
Skipping view @ c1040000 - no VACB, but PTE is valid!
  Loading file cache database (100% of 131072 PTEs)
  SkippedPageTableReads = 0
  File cache has 21510 valid pages


  Usage Summary (in Kb):
Control Valid Standby/Dirty Shared Locked Name
89be09c8   272      0     0     0  default_message_database.db
8a267ad8     4      0     0     0  $Directory
8a37d140  3504   1196     0     0  $Mft
8a35c928  4320   1676     0     0  outlook.pst
8a2acbb8     4      0     0     0  $Directory
8a42c908   564      0     0     0  $BitMap
8a42bd28    12      0     0     0  $Mft
8a29ed88     4      0     0     0  $Directory
8a467008    48      0     0     0  $Directory
8a42b828   124      0     0     0    No Name for File
8a467c58     8      0     0     0  $Directory
8a357f18   136     20     0     0  archive.pst
8a4a7248    52      0     0     0  $Directory
8a42a928   388      0     0     0  $Directory
8a457a10     4      0     0     0  $Directory
8a3f17a8     4      0     0     0  WIASERVC.LOG
89ede0e8     4      0     0     0  $Directory
8a491708 11900    136     0     0  $Mft
8a429fa0     4      0     0     0  $Directory
8a4318d8  2304  26624     0     0  $LogFile
8a438bc0     4      0     0     0  $Directory
8a4286a8     4      0     0     0  $Directory
8a491c50     8      0     0     0  $Directory
8a455de0     4      0     0     0  $Directory
8a3d0f00     4      0     0     0  $Directory
8a44d7a0  3420      0     0     0  $Mft
8a35ba10  1312      0    16     0  SHDOCVW.DLL
8a324130     4      0     0     0  $Directory
8a42c110     4      0     0     0  $Directory
8a41dd80     4      0     0     0  $Directory
88dfff38     4      0     0     0  $Directory
8a4a7640     4      0     0     0  $Directory
8a464d00     4      0     0     0  $Directory
```

Cache Physical Size

While the system working set includes the amount of physical memory that is mapped into views in the cache's virtual address space, it does not necessarily reflect the total amount of file data that is cached in physical memory. There can be a discrepancy between the two values because additional file data might be in the memory manager's standby or modified page lists.

Recall from Chapter 7 that during the course of working set trimming or page replacement the memory manager can move dirty pages from a working set to either the standby list or modified page list, depending on whether the page contains data that needs to be written to the paging file or other file before the page can be reused. If the memory manager didn't implement these lists, any time a process accessed data previously removed from its working set the memory manager would have to hard-fault it in from disk. Instead, if the accessed data is present on either of these lists, the memory manager simply soft-faults the page back into the process's working set. Thus, the lists serve as in-memory caches of data that's stored in the paging file, executable images, or data files. Thus, the total amount of file data cached on a system includes not only the system working set, but the combined sizes of the standby and modified page lists as well.

An example illustrates how the cache manager can cause much more file data than that containable in the system working set to be cached in physical memory. Consider a system that acts as a dedicated file server. The system has 8 GB of physical memory and the cache virtual size is 960 MB (the maximum on x86 systems). The maximum amount of file data that can be directly mapped in the cache's virtual memory is therefore 960 MB. A client application accesses file data from across the network and server, the file server driver (\Windows\System32\Drivers\Srv.sys, described in Chapter 12), uses cache manager interfaces to read and write file data on behalf of the client. If the client reads through several thousand files of 1 MB each, the cache manager will have to start reusing views when mapping the 961st file. For each file read thereafter, the cache manager unmaps views and remaps them for new files. When the cache manager unmaps a view, the memory manager doesn't discard the file data in the cache's working set that correspond to the view, it moves the data to the standby list. In the absence of any other demand for physical memory, the standby list can consume almost all the physical memory that remains outside the system working set. In other words, virtually the entire 8 GB of the server's physical memory will be used to cache file data, as shown in Figure 11-6.

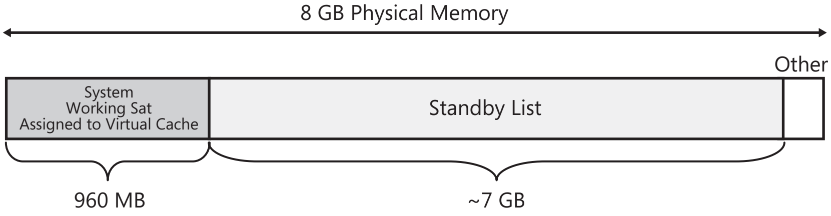
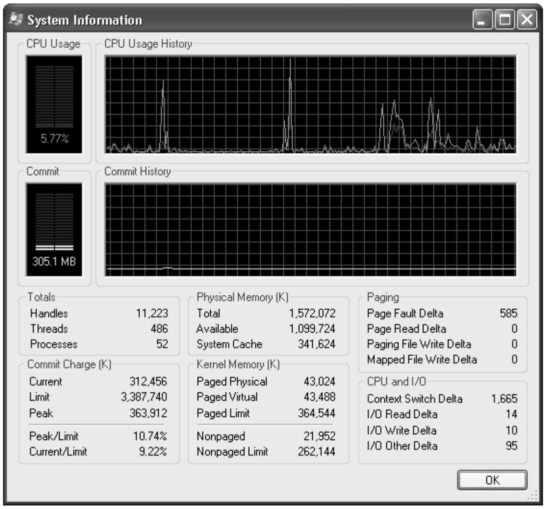


Figure 11-6 Example of where most of physical memory is file cache

Because the total amount of file data cached includes the system working set, modified page list, and standby list—the sizes of which are all controlled by the memory manager—it is in a sense the real cache manager. The cache manager subsystem simply provides convenient interfaces for accessing file data through the memory manager, and it plays an important role with its read-ahead and write-behind policies in influencing what data the memory manager keeps present in physical memory.

To try and accurately reflect the total amount of file data that's cache on a system, both Task Manager and Process Explorer show a value named System Cache in their performance view that reflects the combined sizes of the system working set, standby list, and modified page list. Figure 11-7 shows the system information view in Process Explorer and the System Cache value in the Physical Memory area in the middle of the figure.



Totals

Handles	11,223
Threads	486
Processes	52

Commit Charge (K)

Current	312,456
Limit	3,387,740
Peak	363,912
Peak/Limit	10.74%
Current/Limit	9.22%

Physical Memory (K)

Total	1,572,072
Available	1,099,724
System Cache	341,624

Kernel Memory (K)

Paged Physical	43,024
Paged Virtual	43,488
Paged Limit	364,544
Nonpaged	21,952
Nonpaged Limit	262,144

Paging

Page Fault Delta	585
Page Read Delta	0
Paging File Write Delta	0
Mapped File Write Delta	0

CPU and I/O

Context Switch Delta	1,665
I/O Read Delta	14
I/O Write Delta	10
I/O Other Delta	95

Figure 11-7 Process Explorer's System Information dialog box

Cache Data Structures

The cache manager uses the following data structures to keep track of cached files:

- Each 256-KB slot in the system cache is described by a VACB.
- Each separately opened cached file has a private cache map, which contains information used to control read-ahead (discussed later in the chapter).
- Each cached file has a single shared cache map structure, which points to slots in the system cache that contain mapped views of the file.

These structures and their relationships are described in the next sections.

Systemwide Cache Data Structures

The cache manager keeps track of the state of the views in the system cache by using an array of data structures called *virtual address control blocks* (VACBs). During system initialization, the cache manager allocates a single chunk of nonpaged pool to contain all the VACBs required to describe the system cache. It stores the address of the VACB array in the variable *CcVacbs*. Each VACB represents one 256-KB view in the system cache, as shown in Figure 11-8. The structure of a VACB is shown in Figure 11-9.

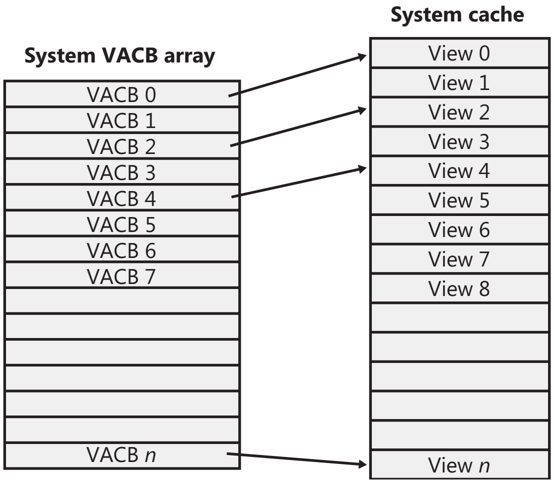
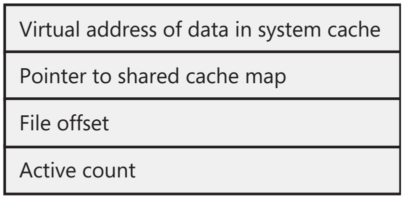


Figure 11-8 System VACB array



Virtual address of data in system cache

Pointer to shared cache map

File offset

Active count

Figure 11-9 VACB structure

As you can see in Figure 11-9, the first field in a VACB is the virtual address of the data in the system cache. The second field is a pointer to the shared cache map structure, which identifies which file is cached. The third field identifies the offset within the file at which the view begins (always based on a 256-KB granularity). Finally, the VACB contains the number of references to the view—that is, how many active reads or writes are accessing the view. During an I/O operation on a file, the file's VACB reference count is incremented and then it's decremented when the I/O operation is over. When the reference count is non-zero the VACB is *active*. For access to file system metadata, the active count represents how many file system drivers have the pages in that view locked into memory.

Per-File Cache Data Structures

Each open handle to a file has a corresponding file object. (File objects are explained in detail in Chapter 9.) If the file is cached, the file object points to a *private cache map* structure that contains the location of the last two reads so that the cache manager can perform intelligent read-ahead (described in the section "Intelligent Read-Ahead"). In addition, all the private cache maps for open instances of a file are linked together.

Each cached file (as opposed to file object) has a *shared cache map* structure that describes the state of the cached file, including its size and (for security reasons) its valid data length. (The function of the valid data length field is explained in the section "Write-Back Caching and Lazy Writing".) The shared cache map also points to the *section object* (maintained by the memory manager, and which describes the file's mapping into virtual memory), the list of private cache maps associated with that file, and any VACBs that describe currently mapped views of the file in the system cache. (See Chapter 7 for more about section object pointers.) The relationships among these per-file cache data structures are illustrated in Figure 11-10.

When asked to read from a particular file, the cache manager must determine the answers to two questions:

1. Is the file in the cache?
2. If so, which VACB, if any, refers to the requested location?

In other words, the cache manager must find out whether a view of the file at the desired address is mapped into the system cache. If no VACB contains the desired file offset, the requested data isn't currently mapped into the system cache.

To keep track of which views for a given file are mapped into the system cache, the cache manager maintains an array of pointers to VACBs, which is known as the *VACB index array*. The first entry in the VACB index array refers to the first 256 KB of the file, the second entry to the second 256 KB, and so on. The diagram in Figure 11-11 shows four different sections from three different files that are currently mapped into the system cache.

When a process accesses a particular file in a given location, the cache manager looks in the appropriate entry in the file's VACB index array to see whether the requested data has been mapped into the cache. If the array entry is nonzero (and hence contains a pointer to a VACB), the area of the file being referenced is in the cache. The VACB, in turn, points to the location in the system cache where the view of the file is mapped. If the entry is zero, the cache manager must find a free slot in the system cache (and therefore a free VACB) to map the required view.

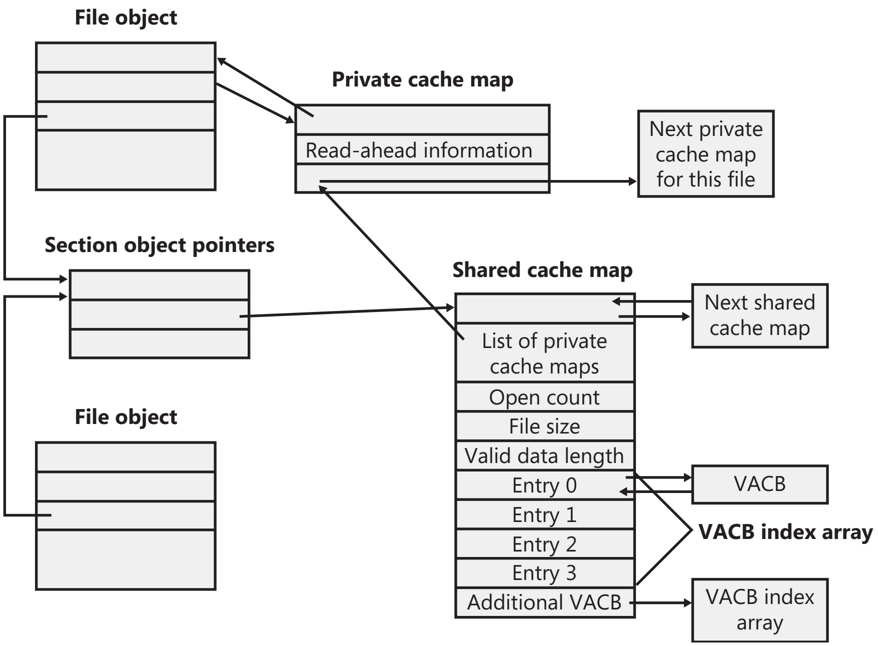


Figure 11-10 Per-file cache data structures

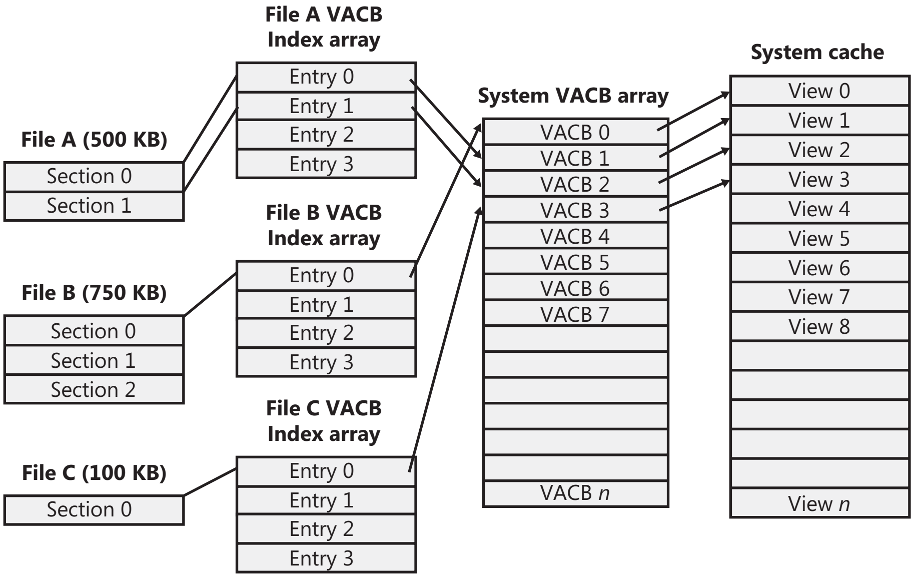


Figure 11-11 VACB index arrays

As a size optimization, the shared cache map contains a VACB index array that is 4 entries in size. Because each VACB describes 256 KB, the entries in this small fixed-size index array can point to VACB array entries that together describe a file of up to 1 MB. If a file is larger than 1 MB, a separate VACB index array is allocated from nonpaged pool, based on the size of the file divided by 256 KB and rounded up in the case of a remainder. The shared cache map then points to this separate structure.

As a further optimization, the VACB index array allocated from nonpaged pool becomes a sparse multilevel index array if the file is larger than 32 MB, where each index array consists of 128 entries. You can calculate the number of levels required for a file with the following formula:

(Number of bits required to represent file size – 18) / 7

Round the result of the equation up to the next whole number. The value 18 in the equation comes from the fact that a VACB represents 256 KB, and 256 KB is 2^{18}. The value 7 comes from the fact that each level in the array has 128 entries and 2^7 is 128. Thus, a file that has a size that is the maximum that can be described with 63 bits (the largest size the cache manager supports) would require only seven levels. The array is sparse because the only branches that the cache manager allocates are ones for which there are active views at the lowest-level index array. Figure 11-12 shows an example of a multilevel VACB array for a sparse file that is large enough to require three levels.

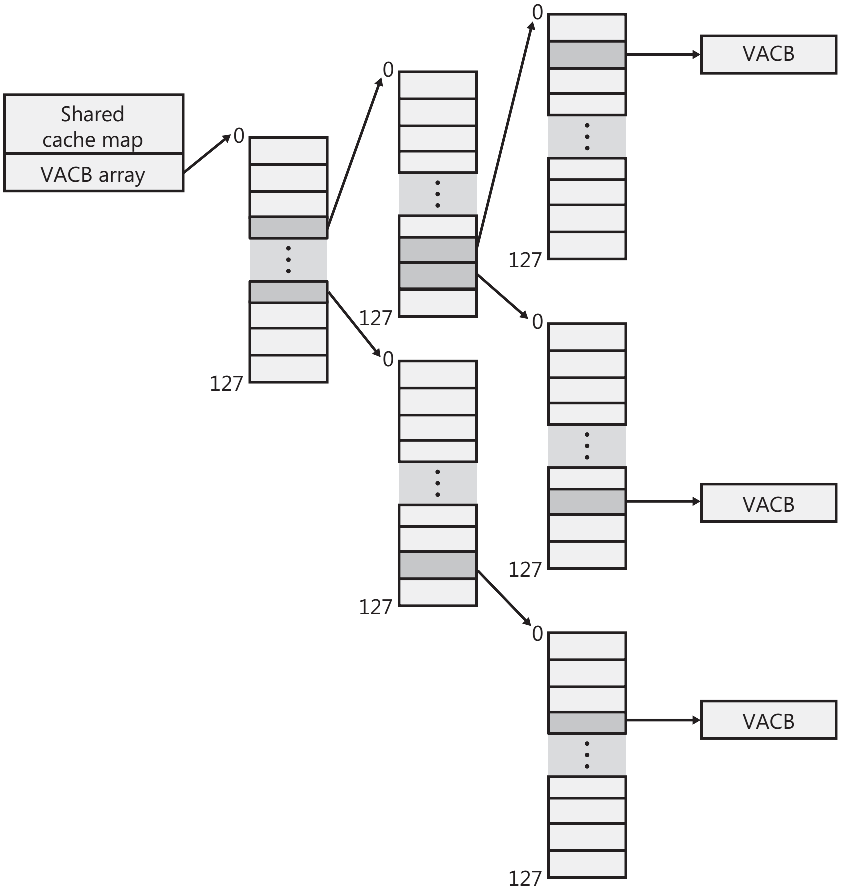


Figure 11-12 Multilevel VACB arrays

This scheme is required to efficiently handle sparse files that might have extremely large file sizes with only a small fraction of valid data because only enough of the array is allocated to handle the currently mapped views of a file. For example, a 32-GB sparse file for which only 256 KB is mapped into the cache's virtual address space would require a VACB array with three allocated index arrays because only one branch of the array has a mapping and a 32-GB (2^{35} bytes) file requires a three-level array. If the cache manager didn't use the multilevel VACB array optimization for this file, it would have to allocate a VACB array with 128,000 entries, or the equivalent of 1000 index arrays.

EXPERIMENT: Looking at Shared and Private Cache Maps

You can use the kernel debugger's *dt* command to look at the shared and private cache map data structure definitions and examine the structures on a live system. First, execute the *!filecache* command and locate an entry in the VACB output with a file name you recognize. In this example, the file is a Microsoft Outlook data file:

```
8653c828  120  160   0   0 debugger.chm
```

The first address is that of a control area data structure, which the memory manager uses to keep track of an address range. (See Chapter 7 for more information.) The control area stores the pointer to the file object that corresponds to the view in the cache. A file object identifies an instance of an open file, in this example the Debugging Tools for Windows help file. Execute the following command using the address of the control area of the entry you identified to see the control area structure:

```
lkd> !ca 8653c828

ControlArea @8653c828
  Segment:    e1bce428    Flink             0   Blink               0
  Section Ref        1    Pfn Ref          46   Mapped Views        4
  User Ref           0    WaitForDel        0   Flush Count         0
  File Object 85d927a8    ModWriteCount     0   System Views        0

  Flags (8008080) File WasPurged Accessed

  File: \Program Files\Debugging Tools for Windows\debugger.chm
```

Next look at the file object referenced by the control area with this command:

```
lkd> dt _File_object 85d927a8
nt!_FILE_OBJECT
   +0x000 Type             : 5
   +0x002 Size             : 112
   +0x004 DeviceObject     : 0x86b78e30
   +0x008 Vpb              : 0x86b90d80
   +0x00c FsContext        : 0xe3b629e0
   +0x010 FsContext2       : 0xe3b62b38
   +0x014 SectionObjectPointer : 0x85e7a334
   +0x018 PrivateCacheMap  : 0x862f5a50
   +0x01c FinalStatus      : 0
   +0x020 RelatedFileObject : (null)
   +0x024 LockOperation    : 0 ''
…
```

The private cache map is at offset 0x18:

```
lkd> dt _private_cache_map 0x862f5a50
nt!_PRIVATE_CACHE_MAP
   +0x000 NodeTypeCode     : 766
   +0x000 Flags            : _PRIVATE_CACHE_MAP_FLAGS
   +0x000 UlongFlags       : 0x2fe
   +0x004 ReadAheadMask    : 0xffff
   +0x008 FileObject       : 0x85d927a8
   +0x010 FileOffset1      : _LARGE_INTEGER 0x10b28
   +0x018 BeyondLastByte1  : _LARGE_INTEGER 0x12206
   +0x020 FileOffset2      : _LARGE_INTEGER 0x10b28
   +0x028 BeyondLastByte2  : _LARGE_INTEGER 0x12206
   +0x030 ReadAheadOffset  : [2] _LARGE_INTEGER 0x0
   +0x040 ReadAheadLength  : [2] 0
   +0x048 ReadAheadSpinLock : 0
   +0x04c PrivateLinks     : _LIST_ENTRY [ 0x862f5a10 - 0x862f5a10 ]
```

Finally, you can locate the shared cache map in the SectionObjectPointers structure of the file object and then view its contents:

```
lkd> dt _Section_object_pointers 0x85e7a334
nt!_SECTION_OBJECT_POINTERS
   +0x000 DataSectionObject : 0x8653c828
   +0x004 SharedCacheMap   : 0x862f5978
   +0x008 ImageSectionObject : (null)
lkd> dt _shared_cache_map 0x862f5978  nt!_SHARED_CACHE_MAP
   +0x000 NodeTypeCode     : 767
   +0x002 NodeByteSize     : 304
   +0x004 OpenCount        : 1
   +0x008 FileSize         : _LARGE_INTEGER 0x19970a
   +0x010 BcbList          : _LIST_ENTRY [ 0x862f5988 - 0x862f5988 ]
   +0x018 SectionSize      : _LARGE_INTEGER 0x1c0000
   +0x020 ValidDataLength  : _LARGE_INTEGER 0x19970a
   +0x028 ValidDataGoal    : _LARGE_INTEGER 0x19970a
   +0x030 InitialVacbs     : [4] (null)
   +0x040 Vacbs            : 0x85d42c00  -> 0x86bba610
   +0x044 FileObject       : 0x85d927a8
   +0x048 ActiveVacb       : 0x86bba610
…
```

File System Interfaces

The first time a file's data is accessed for a read or write operation, the file system driver is responsible for determining whether some part of the file is mapped in the system cache. If it's not, the file system driver must call the *CcInitializeCacheMap* function to set up the per-file data structures described in the preceding section.

Once a file is set up for cached access, the file system driver calls one of several functions to access the data in the file. There are three primary methods for accessing cached data, each intended for a specific situation:

- The copy method copies user data between cache buffers in system space and a process buffer in user space.
- The mapping and pinning method uses virtual addresses to read and write data directly to cache buffers.
- The physical memory access method uses physical addresses to read and write data directly to cache buffers.

File system drivers must provide two versions of the file read operation—cached and noncached—to prevent an infinite loop when the memory manager processes a page fault. When the memory manager resolves a page fault by calling the file system to retrieve data from the file (via the device driver, of course), it must specify this noncached read operation by setting the “no cache” flag in the IRP.

Figure 11-13 illustrates the typical interactions between the cache manager, memory manager, and file system drivers in response to user read or write file I/O. The cache manager is invoked by a file system through the copy interfaces (the *CcCopyRead* and *CcCopyWrite* paths). To process a *CcFastCopyRead* or *CcCopyRead* read, for example, the cache manager creates a view in the cache to map a portion of the file being read and reads the file data into the user buffer by copying from the view. The copy operation generates page faults as it accesses each previously invalid page in the view, and in response the memory manager initiates noncached I/O into the file system driver to retrieve the data corresponding to the part of the file mapped to the page that faulted.

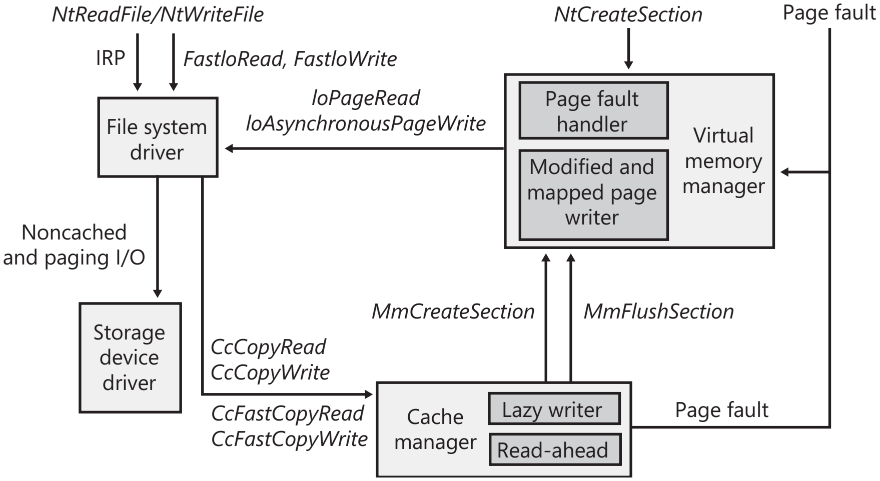


Figure 11-13 File system interaction with cache and memory managers

The next three sections explain these cache access mechanisms, their purpose, and how they’re used.

Copying to and from the Cache

Because the system cache is in system space, it is mapped into the address space of every process. As with all system space pages, however, cache pages aren't accessible from user mode because that would be a potential security hole. (For example, a process might not have the rights to read a file whose data is currently contained in some part of the system cache.) Thus, user application file reads and writes to cached files must be serviced by kernel-mode routines that copy data between the cache's buffers in system space and the application's buffers residing in the process address space. The functions that file system drivers can use to perform this operation are listed in Table 11-4.

Table 11-4 Kernel-Mode Functions for Copying to and from the Cache

Function	Description
CcCopyRead	Copies a specified byte range from the system cache to a user buffer
CcFastCopyRead	Faster variation of *CcCopyRead*, but limited to 32-bit file offsets and synchronous reads
CcCopyWrite	Copies a specified byte range from a user buffer to the system cache
CcFastCopyWrite	Faster variation of *CcCopyWrite*, but limited to 32-bit file offsets and synchronous, non-write-through writes (used by NTFS, not FAT)

You can examine read activity from the cache via the performance counters or system variables listed in Table 11-5.

Table 11-5 System Variables for Examining Read Activity from the Cache

Performance Counter (frequency)	System Variable (count)	Description
Cache: Copy Read Hits %	(*CcCopyReadWait* + *CcCopyReadNoWait*) / (*CcCopyReadWait* + (*CcCopyReadWaitMiss* + *CcCopyReadNoWait*) + *CcCopyReadNoWaitMiss*)	Percentage of copy reads to parts of files that were in the cache (A copy read can still generate paging I/O—the Memory: Cache Faults/Sec counter reports page fault activity for the system working set but includes both hard and soft page faults, so the counter still doesn't indicate actual paging I/O caused by cache faults.)
Cache: Copy Reads/Sec	*CcCopyReadWait* + *CcCopyReadNoWait*	Total copy reads from the cache
Cache: Sync Copy Reads/Sec	*CcCopyReadWait*	Synchronous copy reads from the cache
Cache: Async Copy Reads/Sec	*CcCopyReadNoWait*	Asynchronous copy reads from the cache

Caching with the Mapping and Pinning Interfaces

Just as user applications read and write data in files on a disk, file system drivers need to read and write the data that describes the files themselves (the metadata, or volume structure data). Because the file system drivers run in kernel mode, however, they could, if the cache manager were properly informed, modify data directly in the system cache. To permit this optimization, the cache manager provides the functions shown in Table 11-6. These functions permit the file system drivers to find where in virtual memory the file system metadata resides, thus allowing direct modification without the use of intermediary buffers.

Table 11-6 Functions for Finding Metadata Locations

Function	Description
CcMapData	Maps the byte range for read access
CcPinRead	Maps the byte range for read/write access and pins it
CcPreparePinWrite	Maps and pins the byte range for write access (reads are not valid)
CcPinMappedData	Pins a previously mapped buffer
CcSetDirtyPinnedData	Notifies the cache manager that the data has been modified
CcUnpinData	Releases the pages so that they can be removed from memory

If a file system driver needs to read file system metadata in the cache, it calls the cache manager's mapping interface to obtain the virtual address of the desired data. The cache manager touches all the requested pages to bring them into memory and then returns control to the file system driver. The file system driver can then access the data directly.

If the file system driver needs to modify cache pages, it calls the cache manager's pinning services, which keep the pages being modified in memory. The pages aren't actually locked into memory (such as when a device driver locks pages for direct memory access transfers). Most of the time, a file system driver will mark its metadata stream "no write", which instructs the memory manager's mapped page writer (explained in Chapter 7) to not write the pages to disk until explicitly told to do so. When the file system driver unpins (releases) them, the cache manager flushes any changes to disk and releases the cache view that the metadata occupied.

The mapping and pinning interfaces solve one thorny problem of implementing a file system: buffer management. Without directly manipulating cached metadata, a file system must predict the maximum number of buffers it will need when updating a volume's structure. By allowing the file system to access and update its metadata directly in the cache, the cache manager eliminates the need for buffers, simply updating the volume structure in the virtual memory the memory manager provides. The only limitation the file system encounters is the amount of available memory.

You can examine pinning and mapping activity in the cache via the performance counters or system variables listed in Table 11-7.

Table 11-7 System Variables for Examining Pinning and Mapping Activity

Performance Counter (frequency)	System Variable (count)	Description
Cache: Data Map Hits %	(*CcMapDataWait* + *CcMapDataNoWait*)/ (*CcMapDataWait* + *CcMapDataNoWait*) + (*CcMapDataWaitMiss* + *CcMapDataNoWaitMiss*)	Percentage of data maps to parts of files that were in the cache (A copy read can still generate paging I/O.)
Cache: Data Maps/Sec	*CcMapDataWait* + *CcMapDataNoWait*	Total data maps from the cache
Cache: Sync Data Maps/Sec	*CcMapDataWait*	Synchronous data maps from the cache
Cache: Async Data Maps/Sec	*CcMapDataNoWait*	Asynchronous data maps from the cache
Cache: Data Map Pins/Sec	*CcPinMappedDataCount*	Number of requests to pin mapped data
Cache: Pin Read Hits %	(*CcPinReadWait* + *CcPinReadNoWait*) / (*CcPinReadWait* + *CcPinReadNoWait*) + (*CcPinReadWaitMiss* + *CcPinReadNoWaitMiss*)	Percentage of pinned reads to parts of files that were in the cache (A copy read can still generate paging I/O.)
Cache: Pin Reads/Sec	*CcPinReadWait* + *CcPinReadNoWait*	Total pinned reads from the cache
Cache: Sync Pin Reads/Sec	*CcPinReadWait*	Synchronous pinned reads from the cache
Cache: Async Pin Reads/Sec	*CcPinReadNoWait*	Asynchronous pinned reads from the cache

Caching with the Direct Memory Access Interfaces

In addition to the mapping and pinning interfaces used to access metadata directly in the cache, the cache manager provides a third interface to cached data: *direct memory access* (DMA). The DMA functions are used to read from or write to cache pages without intervening buffers, such as when a network file system is doing a transfer over the network.

The DMA interface returns to the file system the physical addresses of cached user data (rather than the virtual addresses, which the mapping and pinning interfaces return), which can then be used to transfer data directly from physical memory to a network device. Although small amounts of data (1 KB to 2 KB) can use the usual buffer-based copying interfaces, for larger transfers, the DMA interface can result in significant performance improvements for a network server processing file requests from remote systems.

To describe these references to physical memory, a *memory descriptor list* (MDL) is used. (MDLs were introduced in Chapter 7.) The four separate functions described in Table 11-8 create the cache manager's DMA interface.

Table 11-8 Functions That Create the DMA Interface

Function	Description
CcMdlRead	Returns an MDL describing the specified byte range
CcMdlReadComplete	Frees the MDL
CcMdlWrite	Returns an MDL describing a specified byte range (possibly containing zeros)
CcMdlWriteComplete	Frees the MDL, and marks the range as modified

You can examine MDL activity from the cache via the performance counters or system variables listed in Table 11-9.

Table 11-9 System Variables for Examining MDL Activity from the Cache

Performance Counter (frequency)	System Variable (count)	Description
Cache: MDL Read Hits %	(*CcMdlReadWait* + *CcMdlReadNoWait*) / (*CcMdlReadWait* + *CcMdlReadNoWait*) + (*CcMdlReadWaitMiss* + (*CcMdlReadWaitMiss* + *CcMdlReadNoWaitMiss*)	Percentage of MDL reads to parts of files that were in the cache (References to pages satisfied by an MDL read can still generate paging I/O.)
Cache: MDL Reads/Sec	*CcMdlReadWait* + *CcMdlReadNoWait*	Total MDL reads from the cache
Cache: Sync MDL Reads/Sec	*CcMdlReadWait*	Synchronous MDL reads from the cache
Cache: Async MDL Reads/Sec	*CcMdlReadNoWait*	Asynchronous MDL reads from the cache

Fast I/O

Whenever possible, reads and writes to cached files are handled by a high-speed mechanism named *fast I/O*. Fast I/O is a means of reading or writing a cached file without going through the work of generating an IRP, as described in Chapter 9. With fast I/O, the I/O manager calls the file system driver's fast I/O routine to see whether I/O can be satisfied directly from the cache manager without generating an IRP.

Because the cache manager is architected on top of the virtual memory subsystem, file system drivers can use the cache manager to access file data simply by copying to or from pages mapped to the actual file being referenced without going through the overhead of generating an IRP.

Fast I/O doesn't always occur. For example, the first read or write to a file requires setting up the file for caching (mapping the file into the cache and setting up the cache data structures, as explained earlier in the section "Cache Data Structures"). Also, if the caller specified an asynchronous read or write, fast I/O isn't used because the caller might be stalled during paging I/O operations required to satisfy the buffer copy to or from the system cache and thus not really providing the requested asynchronous I/O operation. But even on a synchronous I/O, the file system driver might decide that it can't process the I/O operation by using the fast I/O mechanism, say, for example, if the file in question has a locked range of bytes (as a result of calls to the Windows32 *LockFile* and *UnlockFile* functions). Because the cache manager doesn't know what parts of which files are locked, the file system driver must check the validity of the read or write, which requires generating an IRP. The decision tree for fast I/O is shown in Figure 11-14.

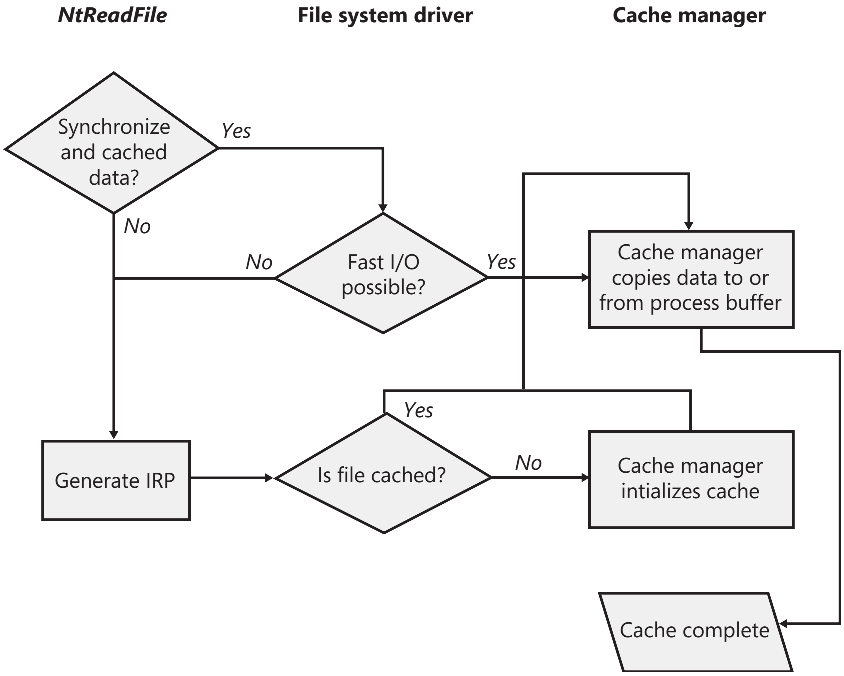


Figure 11-14 Fast I/O decision tree

These steps are involved in servicing a read or a write with fast I/O:

1. A thread performs a read or write operation.
2. If the file is cached and the I/O is synchronous, the request passes to the fast I/O entry point of the file system driver. If the file isn't cached, the file system driver sets up the file for caching so that the next time, fast I/O can be used to satisfy a read or write request.
3. If the file system driver's fast I/O routine determines that fast I/O is possible, it calls the cache manager read or write routine to access the file data directly in the cache. (If fast I/O isn't possible, the file system driver returns to the I/O system, which then generates an IRP for the I/O and eventually calls the file system's regular read routine.)

4. The cache manager translates the supplied file offset into a virtual address in the cache.
5. For reads, the cache manager copies the data from the cache into the buffer of the process requesting it; for writes, it copies the data from the buffer to the cache.
6. One of the following actions occurs:
    - For reads where FILE_FLAG_RANDOM_ACCESS wasn't specified when the file was opened, the read-ahead information in the caller's private cache map is updated.
    - For writes, the dirty bit of any modified page in the cache is set so that the lazy writer will know to flush it to disk.
    - For write-through files, any modifications are flushed to disk.

> **Note** The fast I/O path isn't limited to occasions when the requested data already resides in physical memory. As noted in steps 5 and 6 of the preceding list, the cache manager simply accesses the virtual addresses of the already opened file where it expects the data to be. If a cache miss occurs, the memory manager dynamically pages the data into physical memory.

The performance counters or system variables listed in Table 11-10 can be used to determine the fast I/O activity on the system.

Table 11-10 System Variables for Determining Fast I/O Activity

Performance Counter (frequency)	System Variable (count)	Description
Cache: Sync Fast Reads/Sec	*CcFastReadWait*	Synchronous reads that were handled as fast I/O requests
Cache: Async Fast Reads/Sec	*CcFastReadNoWait*	Asynchronous reads that were handled as fast I/O requests (These are always zero because asynchronous fast reads aren't done in Windows.)
Cache: Fast Read Resource Misses/Sec	*CcFastReadResourceMiss*	Fast I/O operations that couldn't be satisfied because of resource conflicts (This situation can occur with FAT but not with NTFS.)
Cache: Fast Read Not Possibles/ Sec	*CcFastReadNotPossible*	Fast I/O operations that couldn't be satisfied (The file system driver decides; for example, files with byte range locks can't use fast I/O.)

Read Ahead and Write Behind

In this section, you'll see how the cache manager implements reading and writing file data on behalf of file system drivers. Keep in mind that the cache manager is involved in file I/O only when a file is opened without the FILE_FLAG_NO_BUFFERING flag and then read from or written to using the Windows I/O functions (for example, using the Windows *ReadFile* and *WriteFile* functions). Mapped files don't go through the cache manager, nor do files opened with the FILE_FLAG_NO_BUFFERING flag set.

Intelligent Read-Ahead

The cache manager uses the principle of spatial locality to perform *intelligent read-ahead* by predicting what data the calling process is likely to read next based on the data that it is reading currently. Because the system cache is based on virtual addresses, which are contiguous for a particular file, it doesn't matter whether they're juxtaposed in physical memory. File read-ahead for logical block caching is more complex and requires tight cooperation between file system drivers and the block cache because that cache system is based on the relative positions of the accessed data on the disk, and of course, files aren't necessarily stored contiguously on disk. You can examine read-ahead activity by using the Cache: Read Aheads/Sec performance counter or the *CcReadAheadIos* system variable.

Reading the next block of file that is being accessed sequentially provides an obvious performance improvement. To extend read-ahead benefits to cases of strided data accesses (both forward and backward through a file), the cache manager maintains a history of the last two read requests in the private cache map for the file handle being accessed, a method known as *asynchronous read-ahead with history.* If a pattern can be determined from the caller's apparently random reads, the cache manager extrapolates it. For example, if the caller reads page 4000 and then page 3000, the cache manager assumes that the next page the caller will require is page 2000 and prereads it.

Note Although a caller must issue a minimum of three read operations to establish a predictable sequence, only two are stored in the private cache map.

To make read-ahead even more efficient, the Windows32 *CreateFile* function provides a flag indicating forward sequential file access: FILE_FLAG_SEQUENTIAL_SCAN. If this flag is set, the cache manager doesn't keep a read history for the caller for prediction but instead performs sequential read-ahead. However, as the file is read into the cache's working set, the cache manager unmaps views of the file that are no longer active and, if they are unmodified, directs the memory manager to place the pages belonging to the unmapped views at the front of the standby list so that they will be quickly reused. It also reads ahead two times as much data (128 KB instead of 64 KB, for example). As the caller continues reading, the cache manager prereads additional blocks of data, always staying about one read (of the size of the current read) ahead of the caller.

The cache manager's read-ahead is asynchronous because it is performed in a thread separate from the caller's thread and proceeds concurrently with the caller's execution. When called to retrieve cached data, the cache manager first accesses the requested virtual page to satisfy the request and then queues an additional I/O request to retrieve additional data to a system worker thread. The worker thread then executes in the background, reading additional data in anticipation of the caller's next read request. The preread pages are faulted into memory while the program continues executing so that when the caller requests the data it's already in memory.

For applications that have no predictable read pattern, the FILE_FLAG_ RANDOM_ACCESS flag can be specified when the *CreateFile* function is called. This flag instructs the cache manager not to attempt to predict where the application is reading next and thus disables read-ahead. The flag also stops the cache manager from aggressively unmapping views of the file as the file is accessed so as to minimize the mapping/unmapping activity for the file when the application revisits portions of the file.

Write-Back Caching and Lazy Writing

The Cache manager implements a write-back cache with lazy write. This means that data written to files is first stored in memory in cache pages and then written to disk later. Thus, write operations are allowed to accumulate for a short time and are then flushed to disk all at once, reducing the overall number of disk I/O operations.

The cache manager must explicitly call the memory manager to flush cache pages because otherwise the memory manager writes memory contents to disk only when demand for physical memory exceeds supply, as is appropriate for volatile data. Cached file data, however, represents nonvolatile disk data. If a process modifies cached data, the user expects the contents to be reflected on disk in a timely manner.

The decision about how often to flush the cache is an important one. If the cache is flushed too frequently, system performance will be slowed by unnecessary I/O. If the cache is flushed too rarely, you risk losing modified file data in the cases of a system failure (a loss especially irritating to users who know that they asked the application to save the changes) and running out of physical memory (because it's being used by an excess of modified pages).

To balance these concerns, once per second the cache manager's *lazy writer* function executes on a system worker thread and queues one-eighth of the dirty pages in the system cache to be written to disk. If the rate at which dirty pages are being produced is greater than the amount the lazy writer had determined it should write, the lazy writer writes an additional number of dirty pages that it calculates are necessary to match that rate. System worker threads from the systemwide critical worker thread pool actually perform the I/O operations.

Note The cache manager provides a means for file system drivers to track when and how much data has been written to a file. After the lazy writer flushes dirty pages to the disk, the cache manager notifies the file system, instructing it to update its view of the valid data length for the file. (The cache manager and file systems separately track the valid data length for a file in memory.)

You can examine the activity of the lazy writer by examining the cache performance counters or system variables listed in Table 11-11.

Table 11-11 System Variables for Examining the Activity of the Lazy Writer

Performance Counter (frequency)	System Variable (count)	Description
Cache: Lazy Write Flushes/Sec	*CcLazyWriteIos*	Number of lazy writer flushes
Cache: Lazy Write Pages/Sec	*CcLazyWritePages*	Number of pages written by the lazy writer

Disabling Lazy Writing for a File

If you create a temporary file by specifying the flag FILE_ATTRIBUTE_TEMPORARY in a call to the Windows *CreateFile* function, the lazy writer won't write dirty pages to the disk unless there is a severe shortage of physical memory or the file is explicitly flushed. This characteristic of the lazy writer improves system performance—the lazy writer doesn't immediately write data to a disk that might ultimately be discarded. Applications usually delete temporary files soon after closing them.

Forcing the Cache to Write Through to Disk

Because some applications can't tolerate even momentary delays between writing a file and seeing the updates on disk, the cache manager also supports write-through caching on a per-file object basis; changes are written to disk as soon as they're made. To turn on write-through caching, set the FILE_FLAG_WRITE_ THROUGH flag in the call to the *CreateFile* function. Alternatively, a thread can explicitly flush an open file, by using the Windows *FlushFileBuffers* function, when it reaches a point at which the data needs to be written to disk. You can observe cache flush operations that are the result of write-through I/O requests or explicit calls to *FlushFileBuffers* via the performance counters or system variables shown in Table 11-12.

Table 11-12 System Variables for Viewing Cache Flush Operations

Performance Counter (frequency)	System Variable (count)	Description
Cache: Data Flushes/Sec	*CcDataFlushes*	Number of times cache pages were flushed explicitly, because of write through, or because of the lazy writer
Cache: Data Flush Pages/ Sec	*CcDataPages*	Number of pages flushed explicitly, because of write through, or because of the lazy writer

Flushing Mapped Files

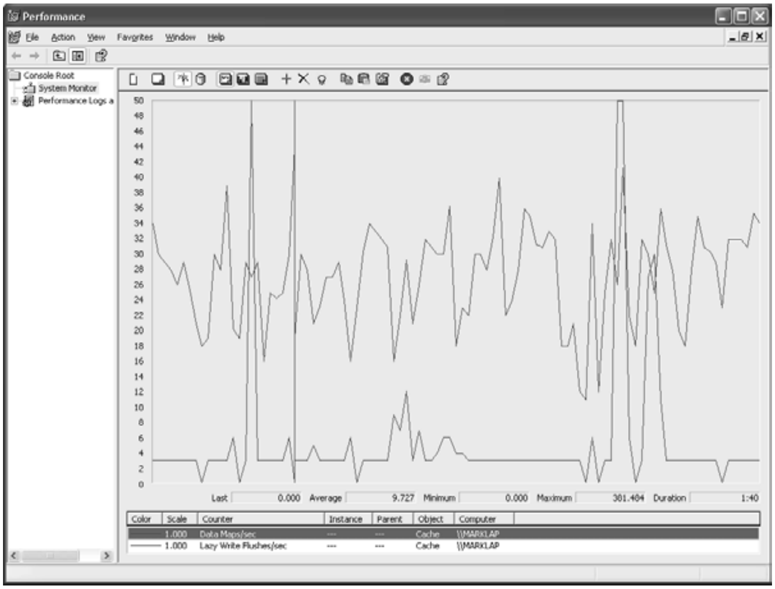
If the lazy writer must write data to disk from a view that's also mapped into another process's address space, the situation becomes a little more complicated because the cache manager will only know about the pages it has modified. (Pages modified by another process are known only to that process because the modified bit in the page table entries for modified pages are kept in the process private page tables.) To address this situation, the memory manager informs the cache manager when a user maps a file. When such a file is flushed in the cache (for example, as a result of a call to the Windows *FlushFileBuffers* function), the cache manager writes the dirty pages in the cache and then checks to see whether the file is also mapped by another process. When the cache manager sees that the file is, the cache manager then flushes the entire view of the section to write out pages that the second process might have modified. If a user maps a view of a file that is also open in the cache, when the view is unmapped, the modified pages are marked as dirty so that when the lazy writer thread later flushes the view, those dirty pages will be written to disk. This procedure works as long as the sequence occurs in the following order:

1. A user unmaps the view.
2. A process flushes file buffers.

If this sequence isn't followed, you can't predict which pages will be written to disk.

EXPERIMENT: Watching Cache Flushes

You can see the cache manager map views into the system cache and flush pages to disk by running the Performance tool and adding the Data Maps/sec and Lazy Write Flushes/sec counter and then copying a large file from one location to another. The line with the generally higher line in the following screen shot is the Data Maps/sec and the other is Lazy Write Flushes/sec.



Last 0.000 Average 9.727 Minimum 0.000 Maximum 381.404 Duration 1:40

Color	Scale	Counter	Instance	Parent	Object	Computer
—	1.000	Data Maps/sec	---	---	Cache	\\MARKLAP
—	1.000	Lazy Write Flushes/sec	---	---	Cache	\\MARKLAP

Write Throttling

The file system and cache manager must determine whether a cached write request will affect system performance and then schedule any delayed writes. First the file system asks the cache manager whether a certain number of bytes can be written right now without hurting performance by using the *CcCanIWrite* function and blocking that write if necessary. Then the file system sets up a callback with the cache manager for automatically writing the bytes when writes are again permitted by calling *CcDeferWrite*. Once it's notified of an impending write operation, the cache manager determines how many dirty pages are in the cache and how much physical memory is available. If few physical pages are free, the cache manager momentarily blocks the file system thread that's requesting to write data to the cache. The cache manager's lazy writer flushes some of the dirty pages to disk and then allows the blocked file system thread to continue. This *write throttling* prevents system performance from degrading because of a lack of memory when a file system or network server issues a large write operation.

Note The effects of write throttling are global to the system because the resource it is based on, available physical memory, is global to the system. This means that if heavy write activity to a slow device triggers write throttling, writes to other devices will also be throttled.

The *dirty page threshold* is the number of pages that the system cache will allow to be dirty before throttling cached writers. This value is computed at system initialization time and depends on physical memory size and the value of the registry LargeSystemCache value, described earlier.

Table 11-13 contains the algorithm used to calculate the dirty page threshold. The calculations in Table 11-13 are overridden if the system maximum working set size is greater than 4 MB—and it often is. (See Chapter 7 to find out how the memory manager chooses system working set sizes—that is, how it determines whether the size is small, medium, or large.) When the maximum working set size exceeds 4 MB, the dirty page threshold is set to the value of the system maximum working set size minus 2 MB.

Table 11-13 Algorithm for Calculating the Dirty Page Threshold

System Memory Size	Dirty Page Threshold
Small	Physical pages / 8
Medium	Physical pages / 4
Large	Sum of the above two values

Write throttling is also useful for network redirectors transmitting data over slow communication lines. For example, suppose a local process writes a large amount of data to a remote file system over a 9600-baud line. The data isn't written to the remote disk until the cache manager's lazy writer flushes the cache. If the redirector has accumulated lots of dirty pages that are flushed to disk at once, the recipient could receive a network timeout before the data transfer completes. By using the *CcSetDirtyPageThreshold* function, the cache manager allows network redirectors to set a limit on the number of dirty cache pages they can tolerate, thus preventing this scenario. By limiting the number of dirty pages, the redirector ensures that a cache flush operation won't cause a network timeout.

Note On Windows XP and later network redirectors, do not set the dirty page threshold and instead rely on the default system values.

EXPERIMENT: Viewing the Write-Throttle Parameters

The *!defwrites* kernel debugger command dumps the values of the kernel variables the cache manager uses, including the number of dirty pages in the file cache (*CcTotalDirtyPages*) when determining whether it should throttle write operations:

```
kd> !defwrites
*** Cache Write Throttle Analysis ***

        CcTotalDirtyPages:                 758 (    3032 Kb)
        CcDirtyPageThreshold:              770 (    3080 Kb)
        MmAvailablePages:                42255 (  169020 Kb)
        MmThrottleTop:                     250 (    1000 Kb)
        MmThrottleBottom:                   30 (     120 Kb)
        MmModifiedPageListHead.Total:      689 (    2756 Kb)

CcTotalDirtyPages within 64 (max charge) pages of the threshold, writes
  may be throttled

Check critical workqueue for the lazy writer, !exqueue 16
```

This output shows that the number of dirty pages is close to the number that triggers write throttling (*CcDirtyPageThreshold*), so if a process tried to write more than 12 pages (48 KB) at the time of the experiment, it would be delayed until the lazy writer lowered the number of dirty pages.

System Threads

As mentioned earlier, the cache manager performs lazy write and read-ahead I/O operations by submitting requests to the common critical system worker thread pool. However, it does limit the use of these threads to one less than the total number of critical worker system threads for small and medium memory systems (two less than the total for large memory systems).

Internally, the cache manager organizes its work requests into two lists (though these are serviced by the same set of executive worker threads):

- The *express queue* is used for read-ahead operations.
- The *regular queue* is used for lazy write scans (for dirty data to flush), write behinds, and lazy closes.

To keep track of the work items the worker threads need to perform, the cache manager creates its own internal *per-processor look-aside list*, a fixed-length list—one for each processor—of worker queue item structures. (Look-aside lists are discussed in Chapter 7.) The number of worker queue items depends on system size: 32 for small-memory systems, 64 for medium-memory systems, 128 for large-memory Windows Professional systems, and 256 for large-memory Windows Server systems.

Conclusion

The cache manager provides a high-speed, intelligent mechanism for reducing disk I/O and increasing overall system throughput. By caching on the basis of virtual blocks, the cache manager can perform intelligent read-ahead. By relying on the global memory manager's mapped file primitive to access file data, the cache manager can provide the special fast I/O mechanism to reduce the CPU time required for read and write operations and also leave all matters related to physical memory management to the single Windows global memory manager, thus reducing code duplication and increasing efficiency.

Chapter 12

File Systems

In this chapter, we present an overview of the file system formats supported by Microsoft Windows. We then describe the types of file system drivers and their basic operation, including how they interact with other system components such as the memory manager and the cache manager. Following that is a description of how to use Filemon (from *www.sysinternals.com*) to troubleshoot a wide variety of file system access problems. In the balance of the chapter, we focus on the on-disk layout of the native Windows file system format called NTFS and the advanced features of NTFS, such as compression, recoverability, quotas, and encryption.

To fully understand this chapter, you should be familiar with the terminology introduced in Chapter 10, including the terms *volume* and *partition*. You'll also need to be acquainted with these additional terms:

- *Sectors* are hardware-addressable blocks on a storage medium. Hard disks for x86 systems almost always define a 512-byte sector size. Thus, if the operating system wants to modify the 632nd byte on a disk, it must write a 512-byte block of data to the second sector on the disk.
- *File system formats* define the way that file data is stored on storage media, and they affect a file system's features. For example, a format that doesn't allow user permissions to be associated with files and directories can't support security. A file system format can also impose limits on the sizes of files and storage devices that the file system supports. Finally, some file system formats efficiently implement support for either large or small files or for large or small disks.
- *Clusters* are the addressable blocks that many file system formats use. Cluster size is always a multiple of the sector size, as shown in Figure 12-1. File system formats use clusters to manage disk space more efficiently; a cluster size that is larger than the sector size divides a disk into more manageable blocks. The potential trade-off of a larger cluster size is wasted disk space, or internal fragmentation, that results because file sizes typically aren't perfect multiples of cluster sizes.

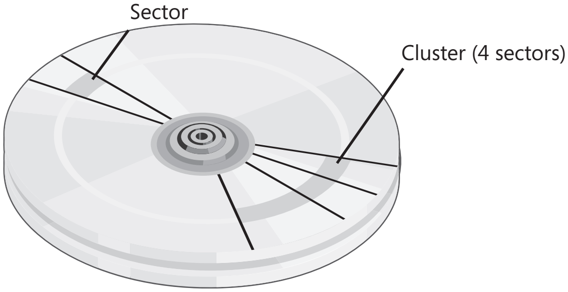


Figure 12-1 Sectors and a cluster on a disk

- *Metadata* is data stored on a volume in support of file system format management. It isn't typically made accessible to applications. Metadata includes the data that defines the placement of files and directories on a volume, for example.

Windows File System Formats

Windows includes support for the following file system formats:

- CDFS
- UDF
- FAT12, FAT16, and FAT32
- NTFS

Each of these formats is best suited for certain environments, as you'll see in the following sections.

CDFS

CDFS (\Windows\System32\Drivers\Cdfs.sys), or CD-ROM file system, is a read-only file system driver that supports a superset of the ISO-9660 format as well as a superset of the Joliet disk format. While the ISO-9660 format is relatively simple and has limitations such as ASCII uppercase names with a maximum length of 32 characters, Joliet is more flexible and supports UNICODE names of arbitrary length, for example. If structures for both formats are present on a disk (to offer maximum compatibility), CDFS uses the Joliet format. CDFS has a number of restrictions:

- A maximum file size of 4 GB
- A maximum of 65,535 directories

CDFS is considered a legacy format because the industry has adopted the Universal Disk Format (UDF) as the standard for read-only media.

UDF

The Windows UDF file system implementation is OSTA (Optical Storage Technology) UDF-compliant. (UDF is a subset of the ISO-13346 format with extensions for formats such as CD-R and DVD-R/RW.) OSTA defined UDF in 1995 as a format to replace the ISO-9660 format for magneto-optical storage media, mainly DVD-ROM. UDF is included in the DVD specification and is more flexible than CDFS. The UDF driver supports UDF versions 1.02, version 1.5 on Windows 2000, and versions 2.0 and 2.01 on Windows XP and Windows Server 2003. UDF file systems have the following traits:

- Directory and filenames can be 254 ASCII or 127 UNICODE characters long.
- Files can be sparse. (Sparse files are defined later in the chapter.)
- File sizes are specified with 64-bits.

Although the UDF format was designed with rewritable media in mind, the Windows UDF driver (\Windows\System32\Drivers\Udfs.sys) provides read-only support. In addition, Windows does not implement support for other UDF features, including named streams, access control lists, or extended attributes.

FAT12, FAT16, and FAT32

Windows supports the FAT file system primarily to enable upgrades from other versions of Microsoft Windows, for compatibility with other operating systems in multiboot systems, and as a floppy disk format. The Windows FAT file system driver is implemented in \Windows\System32\Drivers\Fastfat.sys.

The name of each FAT format includes a number that indicates the number of bits the format uses to identify clusters on a disk. FAT12's 12-bit cluster identifier limits a partition to storing a maximum of 212 (4096) clusters. Windows uses cluster sizes from 512 bytes to 8 KB in size, which limits a FAT12 volume size to 32 MB. Therefore, Windows uses FAT12 as the format for all 5-inch floppy disks and 3.5-inch floppy disks, which store up to 1.44 MB of data.

> **Note** All FAT file system types reserve the first two clusters and the last 16 clusters of a volume, so the number of usable clusters for a FAT12 volume, for instance, is slightly less than 4096.

FAT16, with a 16-bit cluster identifier, can address 216 (65,536) clusters. On Windows, FAT16 cluster sizes range from 512 bytes (the sector size) to 64 KB, which limits FAT16 volume sizes to 4 GB. The cluster size Windows uses depends on the size of a volume. The various sizes are listed in Table 12-1. If you format a volume that is less than 16 MB as FAT by using the *format* command or the Disk Management snap-in, Windows uses the FAT12 format instead of FAT16.

Table 12-1 Default FAT16 Cluster Sizes in Windows

Volume Size	Cluster Size
0–32 MB	512 bytes
33 MB–64 MB	1 KB
65 MB–128 MB	2 KB
129 MB–256 MB	4 KB
257 MB–511 MB	8 KB
512 MB–1023 MB	16 KB
1024 MB–2047 MB	32 KB
2048 MB–4095 MB	64 KB

A FAT volume is divided into several regions, which are shown in Figure 12-2. The file allocation table, which gives the FAT file system format its name, has one entry for each cluster on a volume. Because the file allocation table is critical to the successful interpretation of a volume's contents, the FAT format maintains two copies of the table so that if a file system driver or consistency-checking program (such as Chkdsk) can't access one (because of a bad disk sector, for example) it can read from the other.

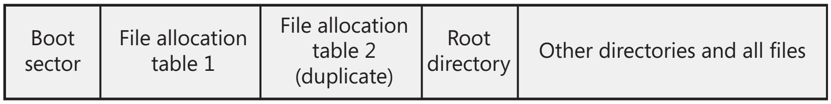

Boot sector
File allocation table 1
File allocation table 2 (duplicate)
Root directory
Other directories and all files

Figure 12-2 FAT format organization

Entries in the file allocation table define file-allocation chains (shown in Figure 12-3) for files and directories, where the links in the chain are indexes to the next cluster of a file's data. A file's directory entry stores the starting cluster of the file. The last entry of the file's allocation chain is the reserved value of 0xFFFF for FAT16 and 0xFFF for FAT12. The FAT entries for unused clusters have a value of 0. You can see in Figure 12-3 that FILE1 is assigned clusters 2, 3, and 4; FILE2 is fragmented and uses clusters 5, 6, and 8; and FILE3 uses only cluster 7. Reading a file from a FAT volume can involve reading large portions of a file allocation table to traverse the file's allocation chains.

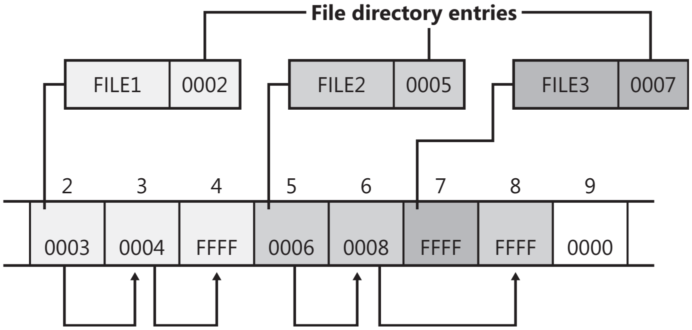


Figure 12-3 Sample FAT file-allocation chains

The root directory of FAT12 and FAT16 volumes are preassigned enough space at the start of a volume to store 256 directory entries, which places an upper limit on the number of files and directories that can be stored in the root directory. (There's no preassigned space or size limit on FAT32 root directories.) A FAT directory entry is 32 bytes and stores a file's name, size, starting cluster, and time stamp (last-accessed, created, and so on) information. If a file has a name that is Unicode or that doesn't follow the MS-DOS 8.3 naming convention, additional directory entries are allocated to store the long filename. The supplementary entries precede the file's main entry. Figure 12-4 shows a sample directory entry for a file named "The quick brown fox." The system has created a THEQUI~1.FOX 8.3 representation of the name (that is, you don't see a "." in the directory entry because it is assumed to come after the eighth character) and used two more directory entries to store the Unicode long filename. Each row in the figure is made up of 16 bytes.

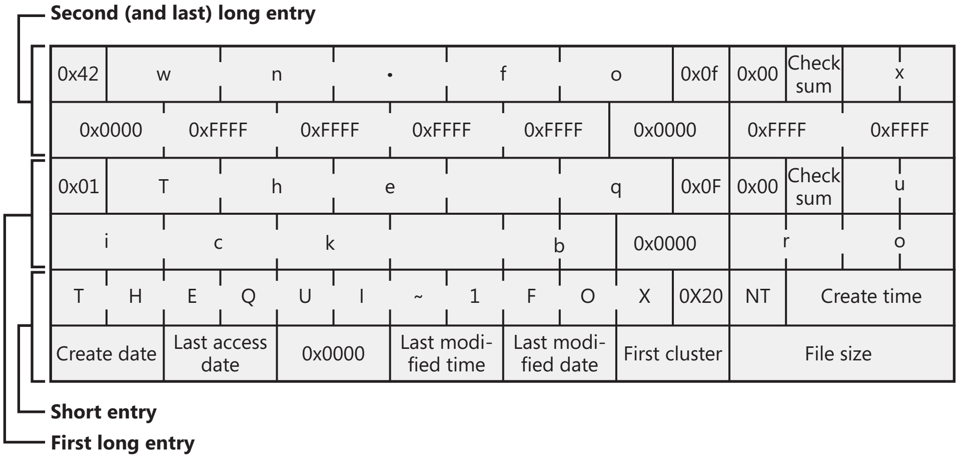

Second (and last) long entry
0x42 w n • f o 0x0f 0x00 Check sum x
0x0000 0xFFFF 0xFFFF 0xFFFF 0xFFFF 0x0000 0xFFFF 0xFFFF
0x01 T h e q 0x0F 0x00 Check sum u
i c k b 0x0000 r o
T H E Q U I ~ 1 F O X 0X20 NT Create time
Create date Last access date 0x0000 Last modified time Last modified date First cluster File size
Short entry
First long entry

Figure 12-4 FAT directory entry

FAT32 is the most recently defined FAT-based file system format, and it's included with Windows 95 OSR2, Windows 98, and Windows Millennium Edition. FAT32 uses 32-bit cluster identifiers but reserves the high 4 bits, so in effect it has 28-bit cluster identifiers. Because FAT32 cluster sizes can be as large as 32 KB, FAT32 has a theoretical ability to address 8-terabyte (TB) volumes. Although Windows works with existing FAT32 volumes of larger sizes (created in other operating systems), it limits new FAT32 volumes to a maximum of 32 GB. FAT32's higher potential cluster numbers let it manage disks more efficiently than FAT16; it can handle up to 128-MB volumes with 512-byte clusters. Table 12-2 shows default cluster sizes for FAT32 volumes.

Table 12-2 Default Cluster Sizes for FAT32 Volumes

Partition Size	Cluster Size
32 MB–8 GB	4 KB
8 GB–16 GB	8 KB
16 GB–32 GB	16 KB
32 GB	32 KB

Besides the higher limit on cluster numbers, other advantages FAT32 has over FAT12 and FAT16 include the fact that the FAT32 root directory isn't stored at a predefined location on the volume, the root directory doesn't have an upper limit on its size, and FAT32 stores a second copy of the boot sector for reliability. A limitation FAT32 shares with FAT16 is that the maximum file size is 4 GB because directories store file sizes as 32-bit values.

Note Windows XP introduces support for FAT32 on DVD-RAM devices.

NTFS

As we said at the beginning of the chapter, the NTFS file system is the native file system format of Windows. NTFS uses 64-bit cluster numbers. This capacity gives NTFS the ability to address volumes of up to 16 exabytes (16 billion GB); however, Windows limits the size of an NTFS volume to that addressable with 32-bit clusters, which is slightly less than 256 TB (using 64-KB clusters). Table 12-3 shows the default cluster sizes for NTFS volumes. (You can override the default when you format an NTFS volume.) NTFS also supports 2^{32}-1 files per volume. The NTFS format allows for files that are 16 exabytes in size, but the implementation limits the maximum file size to 16 TB.

Table 12-3 Default Cluster Sizes for NTFS Volumes

Volume Size	Default Cluster Size
512 MB or less	512 bytes
513 MB–1024 MB (1 GB)	1 KB
1025 MB–2048 MB (2 GB)	2 KB
Greater than 2048 MB	4 KB

NTFS includes a number of advanced features, such as file and directory security, disk quotas, file compression, directory-based symbolic links, and encryption. One of its most significant features is *recoverability*. If a system is halted unexpectedly, the metadata of a FAT volume can be left in an inconsistent state, leading to the corruption of large amounts of file and directory data. NTFS logs changes to metadata in a transactional manner so that file system structures can be repaired to a consistent state with no loss of file or directory structure information. (File data can be lost, however.)

We'll describe NTFS data structures and advanced features in detail later in this chapter.

File System Driver Architecture

File system drivers (FSDs) manage file system formats. Although FSDs run in kernel mode, they differ in a number of ways from standard kernel-mode drivers. Perhaps most significant, they must register as an FSD with the I/O manager and they interact more extensively with the memory manager. For enhanced performance, file system drivers also usually rely on the services of the cache manager. Thus, they use a superset of the exported Ntoskrnl functions that standard drivers use. Whereas you need the Windows DDK in order to build standard kernel-mode drivers, you must have the Windows Installable File System (IFS) Kit to build file system drivers. (See Chapter 1 for more information on the DDK, and see *www.microsoft.com/whdc/devtools/ifskit* for more information on the IFS Kit.)

Windows has two different types of file system drivers:

- *Local FSDs* manage volumes directly connected to the computer.
- *Network FSDs* allow users to access data volumes connected to remote computers.

Local FSDs

Local FSDs include Ntfs.sys, Fastfat.sys, Udfs.sys, Cdfs.sys, and the Raw FSD (integrated in Ntoskrnl.exe). Figure 12-5 shows a simplified view of how local FSDs interact with the I/O manager and storage device drivers. As we described in the section “Volume Mounting” in Chapter 10, a local FSD is responsible for registering with the I/O manager. Once the FSD is registered, the I/O manager can call on it to perform volume recognition when applications or the system initially access the volumes. Volume recognition involves an examination of a volume’s boot sector and often, as a consistency check, the file system metadata.

The first sector of every Windows-supported file system format is reserved as the volume’s boot sector. A boot sector contains enough information so that a local FSD can both identify the volume on which the sector resides as containing a format that the FSD manages and locate any other metadata necessary to identify where metadata is stored on the volume.

When a local FSD recognizes a volume, it creates a device object that represents the mounted file system format. The I/O manager makes a connection through the volume parameter block (VPB) between the volume’s device object (which is created by a storage device) and the device object that the FSD created. The VPB’s connection results in the I/O manager redirecting I/O requests targeted at the volume device object to the FSD device object. (See Chapter 10 for more information on VPBs.)

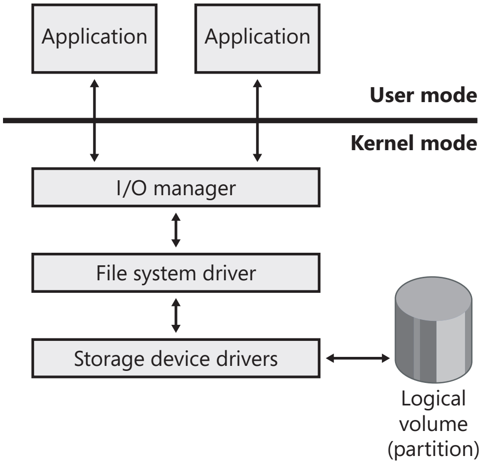


Figure 12-5 Local FSD

To improve performance, local FSDs usually use the cache manager to cache file system data, including metadata. They also integrate with the memory manager so that mapped files are implemented correctly. For example, they must query the memory manager whenever an application attempts to truncate a file in order to verify that no processes have mapped the part of the file beyond the truncation point. Windows doesn't permit file data that is mapped by an application to be deleted either through truncation or file deletion.

Local FSDs also support file system dismount operations, which permit the system to disconnect the FSD from the volume object. A dismount occurs whenever an application requires raw access to the on-disk contents of a volume or the media associated with a volume is changed. The first time an application accesses the media after a dismount, the I/O manager reinitiates a volume mount operation for the media.

Remote FSDs

Remote FSDs consist of two components: a client and a server. A client-side remote FSD allows applications to access remote files and directories. The client FSD accepts I/O requests from applications and translates them into network file system protocol commands that the FSD sends across the network to a server-side component, which is typically a remote FSD. A server-side FSD listens for commands coming from a network connection and fulfills them by issuing I/O requests to the local FSD that manages the volume on which the file or directory that the command is intended for resides.

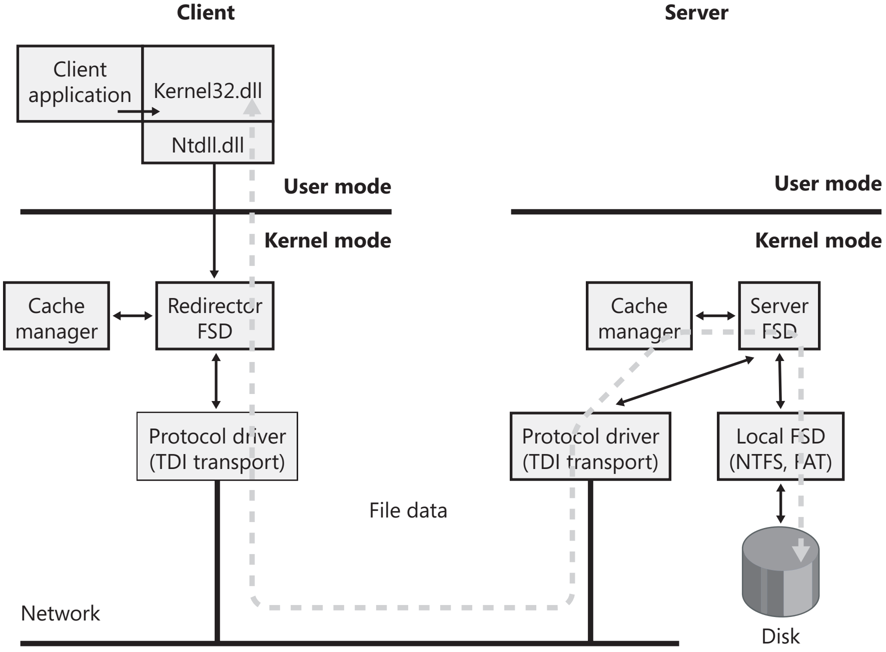


Figure 12-6 CIFS file sharing

Windows includes a client-side remote FSD named LANMan Redirector (redirector) and a server-side remote FSD server named LANMan Server (\Windows\System32\Drivers\Srv.sys - server). The redirector is implemented as a port/miniport driver combination, where the port driver (\Windows\System32\Drivers\Rdbss.sys) is implemented as a driver subroutine library and the miniport (\Windows\System32\Drivers\Mrxsmb.sys) uses services implemented by the port driver. Another redirector miniport driver is WebDAV (\Windows\System32\Drivers\Mrxdav.sys), which implements the client side of file access over HTTP. The port/miniport model simplifies redirector development because the port driver, which all remote FSD miniport drivers share, handles many of the mundane details involved with interfacing a client-side remote FSD to the Windows I/O manager. In addition to the FSD components, both LANMan Redirector and LANMan Server include Windows services named Workstation and Server, respectively. Figure 12-6 shows the relationship bewteen a client accessing files remotely from a server through the redirector and server FSDs.

Windows relies on the Common Internet File System (CIFS) protocol to format messages exchanged between the redirector and the server. CIFS is a version of Microsoft's Server Message Block (SMB) protocol. (For more information on CIFS, go to *www.cifs.com*.)

Like local FSDs, client-side remote FSDs usually use cache manager services to locally cache file data belonging to remote files and directories. However, client-side remote FSDs must implement a distributed cache coherency protocol, called *oplocks* (opportunistic locking), so that the data an application sees when it accesses a remote file is the same as the data applications running on other computers that are accessing the same file see. Although server-side remote FSDs participate in maintaining cache coherency across their clients, they don't cache data from the local FSDs because local FSDs cache their own data.

When a client wants to access a server file, it must first request an oplock. The client dictates the kind of caching that the client can perform based on the type of oplock that the server grants.

There are three main types of oplock:

- A Level I oplock is granted when a client has exclusive access to a file. A client holding this type of oplock for a file can cache both reads and writes on the client system.
- A Level II oplock represents a shared file lock. Clients that hold a Level II oplock can cache reads, but writing to the file invalidates the Level II oplock.
- A Batch oplock is the most permissive kind of oplock. A client with this oplock can cache both reads and writes to the file as well as open and close the file without requesting additional oplocks. Batch oplocks are typically used only to support the execution of batch files, which can open and close a file repeatedly as they execute.

If a client has no oplock, it can cache neither read nor write data locally and instead must retrieve data from the server and send all modifications directly to the server.

An example, shown in Figure 12-7, will help illustrate oplock operation. The server automatically grants a Level I oplock to the first client to open a server file for access. The redirector on the client caches the file data for both reads and writes in the file cache of the client machine. If a second client opens the file, it too requests a Level I oplock. However, because there are now two clients accessing the same file, the server must take steps to present a consistent view of the file's data to both clients. If the first client has written to the file, as is the case in Figure 12-7, the server revokes its oplock and grants neither client an oplock. When the first client's oplock is revoked, or *broken*, the client flushes any data it has cached for the file back to the server.

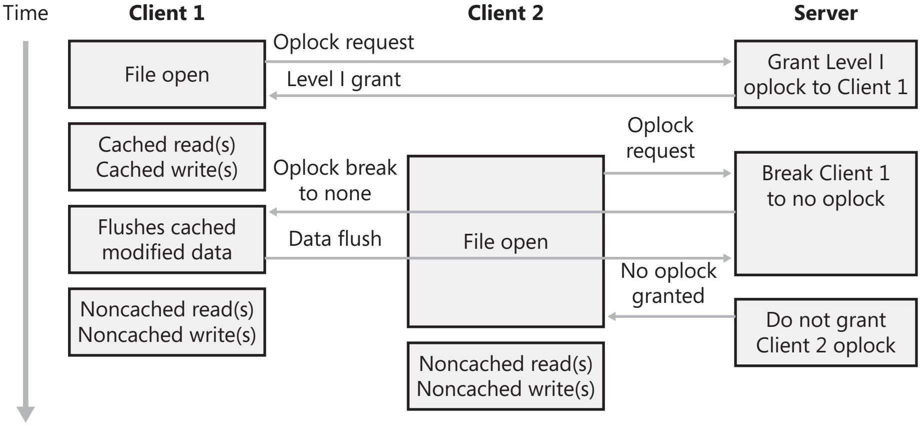
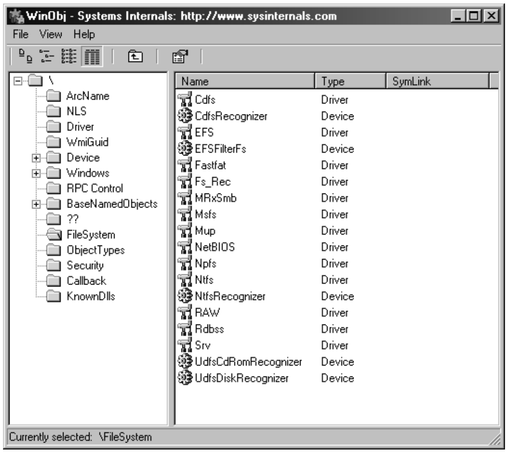

Time
Client 1
Client 2
Server
File open
Oplock request
Level I grant
Grant Level I oplock to Client 1
Cached read(s) Cached write(s)
Oplock break to none
Oplock request
Break Client 1 to no oplock
Flushes cached modified data
Data flush
File open
No oplock granted
Noncached read(s) Noncached write(s)
Do not grant Client 2 oplock
Noncached read(s) Noncached write(s)

Figure 12-7 Oplock example

If the first client hadn't written to the file, the first client's oplock would have been broken to a Level II oplock, which is the same type of oplock the server grants to the second client. Now both clients can cache reads, but if either writes to the file, the server revokes their oplocks so that noncached operation commences. Once oplocks are broken, they aren't granted again for the same open instance of a file. However, if a client closes a file and then reopens it, the server reassesses what level of oplock to grant the client based on what other clients have the file open and whether or not at least one of them has written to the file.

EXPERIMENT: Viewing the List of Registered File Systems

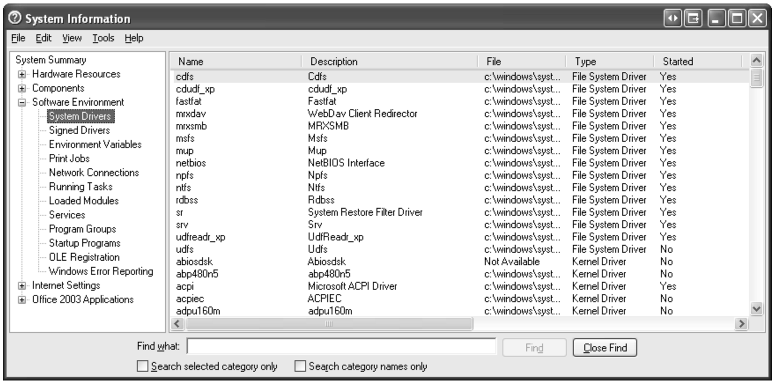
When the I/O manager loads a device driver into memory, it typically names the driver object it creates to represent the driver so that it's placed in the \Drivers object manager directory. The driver objects for any driver the I/O manager loads that have a Type attribute value of SERVICE_FILE_SYSTEM_DRIVER (2) are placed in the \FileSystem directory by the I/O manager. Thus, using a tool such as Winobj (from *www.sysinternals.com*), you can see the file systems that have registered on a system, as shown in the following screen shot. (Note that some file system drivers also place device objects in the \FileSystem directory.)



Name	Type	SymLink
Cdfs	Driver	
CdfsRecognizer	Device	
EFS	Driver	
EFSFilterFs	Device	
Fastfat	Driver	
Fs_Rec	Driver	
MRxSmb	Driver	
Msfs	Driver	
Mup	Driver	
NetBIOS	Driver	
Npfs	Driver	
Ntfs	Driver	
NtfsRecognizer	Device	
RAW	Driver	
Rdbss	Driver	
Srv	Driver	
UdfsCdRomRecognizer	Device	
UdfsDiskRecognizer	Device	

Currently selected: \FileSystem

Another way to see registered file systems is to run the System Information viewer. On Windows 2000, run the Computer Management MMC snap-in and select Drivers under the Software Environment in the System Information node; on Windows XP and Windows Server 2003, run Msinfo32 from the Start menu's Run dialog box and select System Drivers under Software Environment. Sort the list of drivers by clicking the Type column and drivers with a Type attribute of SERVICE_FILE_SYSTEM_DRIVER group together.



Name	Description	File	Type	Started
cdfs	Cdfs	c:\windows\syst...	File System Driver	Yes
cdudf_xp	cdudf_xp	c:\windows\syst...	File System Driver	Yes
fastfat	Fastfat	c:\windows\syst...	File System Driver	Yes
mrxdav	WebDav Client Redirector	c:\windows\syst...	File System Driver	Yes
mrxsmb	MRXSMB	c:\windows\syst...	File System Driver	Yes
msfs	Msfs	c:\windows\syst...	File System Driver	Yes
mup	Mup	c:\windows\syst...	File System Driver	Yes
netbios	NetBIOS Interface	c:\windows\syst...	File System Driver	Yes
npfs	Npfs	c:\windows\syst...	File System Driver	Yes
ntfs	Ntfs	c:\windows\syst...	File System Driver	Yes
rdbss	Rdbss	c:\windows\syst...	File System Driver	Yes
sr	System Restore Filter Driver	c:\windows\syst...	File System Driver	Yes
srv	Srv	c:\windows\syst...	File System Driver	Yes
udfreadr_xp	UdfReadr_xp	c:\windows\syst...	File System Driver	Yes
udfs	Udfs	c:\windows\syst...	File System Driver	No
abiosdsk	Abiosdsk	Not Available	Kernel Driver	No
abp480n5	abp480n5	c:\windows\syst...	Kernel Driver	No
acpi	Microsoft ACPI Driver	c:\windows\syst...	Kernel Driver	Yes
acpiec	ACPIEC	c:\windows\syst...	Kernel Driver	No
adpu160m	adpu160m	c:\windows\syst...	Kernel Driver	No

Note that just because a driver registers as a file system driver type doesn't mean that it is a local or remote FSD. For example, Npfs (Named Pipe File System), which is visible in the list just shown, is a network API driver that supports named pipes but implements a private namespace, and therefore, is in some ways like a file system driver. See Chapter 13 for an experiment that reveals the Npfs namespace.

File System Operation

Applications and the system access files in two ways: directly, via file I/O functions (such as *ReadFile* and *WriteFile*), and indirectly, by reading or writing a portion of their address space that represents a mapped file section. (See Chapter 7 for more information on mapped files.) Figure 12-8 is a simplified diagram that shows the components involved in these file system operations and the ways in which they interact. As you can see, an FSD can be invoked through several paths:

- From a user or system thread performing explicit file I/O
- From the memory manager's modified and mapped page writers
- Indirectly from the cache manager's lazy writer
- Indirectly from the cache manager's read-ahead thread
- From the memory manager's page fault handler

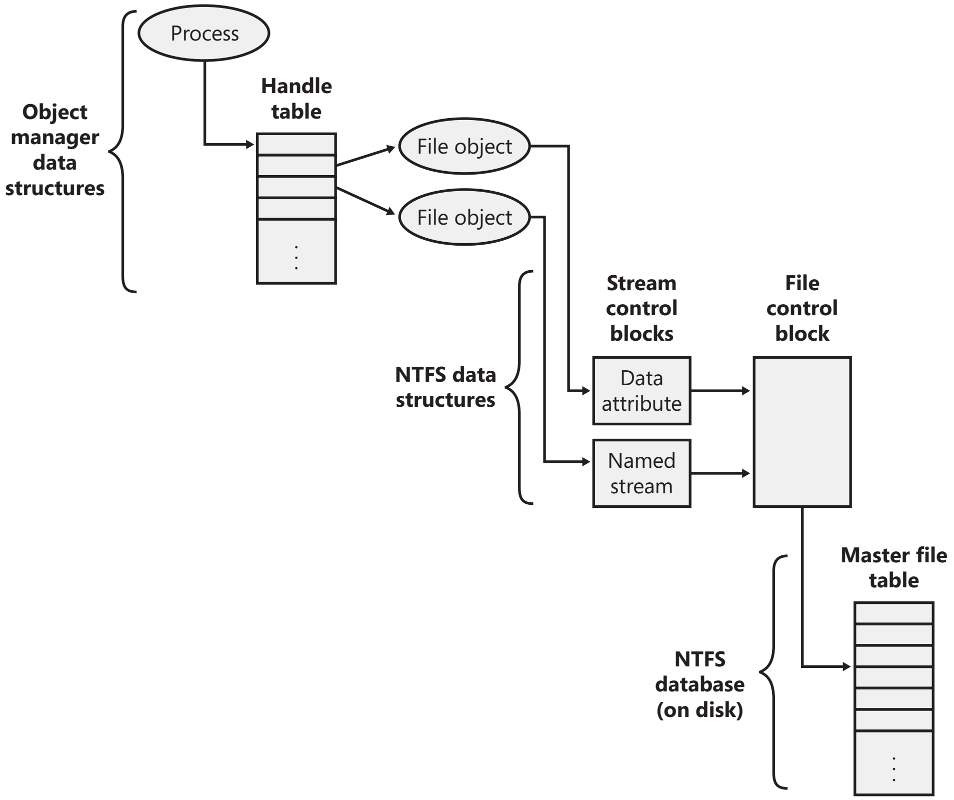


Figure 12-8 Components involved in file system I/O

The following sections describe the circumstances surrounding each of these scenarios and the steps FSDs typically take in response to each one. You'll see how much FSDs rely on the memory manager and the cache manager.

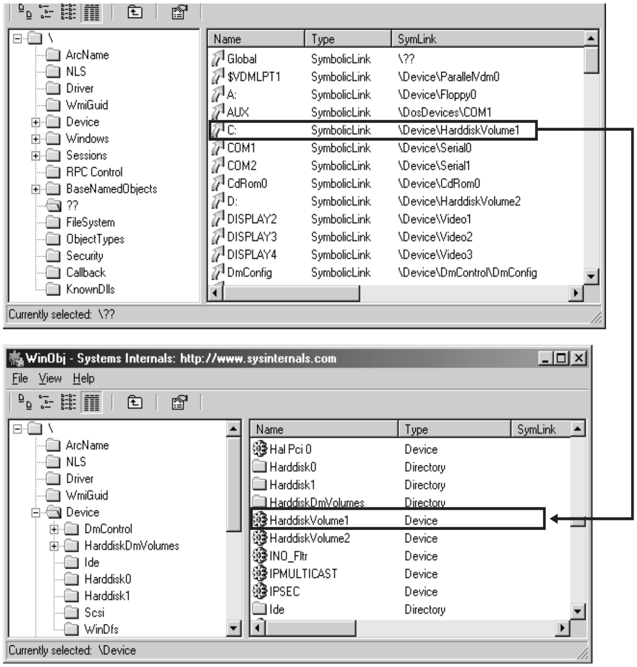
Explicit File I/O

The most obvious way an application accesses files is by calling Windows I/O functions such as *CreateFile*, *ReadFile*, and *WriteFile*. An application opens a file with *CreateFile* and then reads, writes, or deletes the file by passing the handle returned from *CreateFile* to other Windows functions. The *CreateFile* function, which is implemented in the Kernel32.dll Windows client-side DLL, invokes the native function *NtCreateFile*, forming a complete root-relative pathname for the path that the application passed to it (processing "." and ".." symbols in the pathname) and prepending the path with "\??" (for example, \??\C:\Daryl\Todo.txt).

The *NtCreateFile* system service uses *ObOpenObjectByName* to open the file, which parses the name starting with the object manager root directory and the first component of the path name ("??"). Chapter 3 includes a thorough description of object manager name resolution and its use of process device maps, but we'll review the steps it follows here with a focus on volume drive letter lookup.

The first step the object manager takes is to translate \?? to the process's per-session namespace directory that the DosDevicesDirectory field of the device map structure in the process object references. On Windows 2000 systems without Terminal Services, the DosDevicesDirectory field references the \?? directory; and on Windows 2000 systems with Terminal Services, the device map references a per-session directory in which symbolic link objects representing all valid volume drive letters are stored. On Windows XP and Windows Server 2003, however, only volume names for network shares are typically stored in the per-session directory, so on those systems when a name (C: in this example) is not present in the per-session directory, the object manager restarts its search in the directory referenced by the GlobalDosDevicesDirectory field of the device map associated with the per-session directory. The GlobalDosDevicesDirectory always points at the \Global?? directory, which is where Windows XP and Windows Server 2003 store volume drive letters for local volumes. (See the section "Session Namespace" in Chapter 3 for more information.)

The symbolic link for a volume drive letter points to a volume device object under \Device, so when the object manager encounters the volume object, the object manager hands the rest of the pathname to the parse function that the I/O manager has registered for device objects, *IopParseDevice*. (In volumes on dynamic disks, a symbolic link points to an intermediary symbolic link, which points to a volume device object.) Figure 12-9 shows how volume objects are accessed through the object manager namespace. The figure shows how the \??\C: symbolic link points to the \Device\HarddiskVolume1 volume device object on Windows 2000 without Terminal Services.



Name	Type	SymLink
Global	SymbolicLink	\??
$VDMLPT1	SymbolicLink	\Device\ParallelVdm0
A:	SymbolicLink	\Device\Floppy0
AUX	SymbolicLink	\DosDevices\COM1
C:	SymbolicLink	\Device\HarddiskVolume1
COM1	SymbolicLink	\Device\Serial0
COM2	SymbolicLink	\Device\Serial1
CdRom0	SymbolicLink	\Device\CdRom0
D:	SymbolicLink	\Device\HarddiskVolume2
DISPLAY2	SymbolicLink	\Device\Video1
DISPLAY3	SymbolicLink	\Device\Video2
DISPLAY4	SymbolicLink	\Device\Video3
DmConfig	SymbolicLink	\Device\DmControl\DmConfig

Currently selected: \??

Name	Type	SymLink
Hal Pci 0	Device	
Harddisk0	Directory	
Harddisk1	Directory	
HarddiskDmVolumes	Directory	
HarddiskVolume1	Device	
HarddiskVolume2	Device	
INO_Fltr	Device	
IPMULTICAST	Device	
IPSEC	Device	
Ide	Directory	

Currently selected: \Device

Figure 12-9 Drive-letter name resolution

After locking the caller's security context and obtaining security information from the caller's token, *IopParseDevice* creates an I/O request packet (IRP) of type IRP_MJ_CREATE, creates a file object that stores the name of the file being opened, follows the VPB of the volume device object to find the volume's mounted file system device object, and uses *IoCallDriver* to pass the IRP to the file system driver that owns the file system device object.

When an FSD receives an IRP_MJ_CREATE IRP, it looks up the specified file, performs security validation, and if the file exists and the user has permission to access the file in the way requested, returns a success code. The object manager creates a handle for the file object in the process's handle table, and the handle propagates back through the calling chain, finally reaching the application as a return parameter from *CreateFile*. If the file system fails the create, the I/O manager deletes the file object it created for it.

We've skipped over the details of how the FSD locates the file being opened on the volume, but a *ReadFile* function call operation shares many of the FSD's interactions with the cache manager and storage driver. The path into the kernel taken as the result of a call to *ReadFile* is the same as for a call to *CreateFile*, but the *NtReadFile* system service doesn't need to perform a name lookup—it calls on the object manager to translate the handle passed from *ReadFile* into a file object pointer. If the handle indicates that the caller obtained permission to read the file when the file was opened, *NtReadFile* proceeds to create an IRP of type IRP_MJ_READ and sends it to the FSD on which the file resides. *NtReadFile* obtains the FSD's device object, which is stored in the file object, and calls *IoCallDriver*, and the I/O manager locates the FSD from the device object and gives the IRP to the FSD.

If the file being read can be cached (that is, the FILE_FLAG_NO_BUFFERING flag wasn't passed to *CreateFile* when the file was opened), the FSD checks to see whether caching has already been initiated for the file object. The *PrivateCacheMap* field in a file object points to a private cache map data structure (which we described in Chapter 11) if caching is initiated for a file object. If the FSD hasn't initialized caching for the file object (which it does the first time a file object is read from or written to), the *PrivateCacheMap* field will be null. The FSD calls the cache manager *CcInitializeCacheMap* function to initialize caching, which involves the cache manager creating a private cache map and, if another file object referring to the same file hasn't initiated caching, a shared cache map and a section object.

After it has verified that caching is enabled for the file, the FSD copies the requested file data from the cache manager's virtual memory to the buffer that the thread passed to the *ReadFile* function. The file system performs the copy within a try/except block so that it catches any faults that are the result of an invalid application buffer. The function the file system uses to perform the copy is the cache manager's *CcCopyRead* function. *CcCopyRead* takes as parameters a file object, file offset, and length.

When the cache manager executes *CcCopyRead*, it retrieves a pointer to a shared cache map, which is stored in the file object. Recall from Chapter 11 that a shared cache map stores pointers to virtual address control blocks (VACBs), with one VACB entry per 256-KB block of the file. If the VACB pointer for a portion of a file being read is null, *CcCopyRead* allocates a VACB, reserving a 256-KB view in the cache manager's virtual address space, and maps (using *MmMapViewInSystemCache*) the specified portion of the file into the view. Then *CcCopyRead* simply copies the file data from the mapped view to the buffer it was passed (the buffer originally passed to *ReadFile*). If the file data isn't in physical memory, the copy operation generates page faults, which are serviced by *MmAccessFault*.

When a page fault occurs, *MmAccessFault* examines the virtual address that caused the fault and locates the virtual address descriptor (VAD) in the VAD tree of the process that caused the fault. (See Chapter 7 for more information on VAD trees.) In this scenario, the VAD describes the cache manager's mapped view of the file being read, so *MmAccessFault* calls *MiDispatchFault* to handle a page fault on a valid virtual memory address. *MiDispatchFault* locates the control area (which the VAD points to) and through the control area finds a file object representing the open file. (If the file has been opened more than once, there might be a list of file objects linked through pointers in their private cache maps.)

With the file object in hand, *MiDispatchFault* calls the I/O manager function *IoPageRead* to build an IRP (of type IRP_MJ_READ) and sends the IRP to the FSD that owns the device object the file object points to. Thus, the file system is reentered to read the data that it requested via *CcCopyRead*, but this time the IRP is marked as noncached and paging I/O. These flags signal the FSD that it should retrieve file data directly from disk, and it does so by determining which clusters on disk contain the requested data and sending IRPs to the volume manager that owns the volume device object on which the file resides. The volume parameter block (VPB) field in the FSD's device object points to the volume device object.

The virtual memory manager waits for the FSD to complete the IRP read and then returns control to the cache manager, which continues the copy operation that was interrupted by a page fault. When *CcCopyRead* completes, the FSD returns control to the thread that called *NtReadFile*, having copied the requested file data—with the aid of the cache manager and the virtual memory manager—to the thread's buffer.

The path for *WriteFile* is similar except that the *NtWriteFile* system service generates an IRP of type IRP_MJ_WRITE and the FSD calls *CcCopyWrite* instead of *CcCopyRead*. *CcCopyWrite*, like *CcCopyRead*, ensures that the portions of the file being written are mapped into the cache and then copies to the cache the buffer passed to *WriteFile*.

If a file's data is already stored in the system's working set, there are several variants on the scenario we've just described. If a file's data is already stored in the cache, *CcCopyRead* doesn't incur page faults. Also, under certain conditions, *NtReadFile* and *NtWriteFile* call an FSD's fast I/O entry point instead of immediately building and sending an IRP to the FSD. Some of these conditions follow: the portion of the file being read must reside in the first 4 GB of the file, the file can have no locks, and the portion of the file being read or written must fall within the file's currently allocated size.

The fast I/O read and write entry points for most FSDs call the cache manager's *CcFastCopyRead* and *CcFastCopyWrite* functions. These variants on the standard copy routines ensure that the file's data is mapped in the file system cache before performing a copy operation. If this condition isn't met, *CcFastCopyRead* and *CcFastCopyWrite* indicate that fast I/O isn't possible. When fast I/O isn't possible, *NtReadFile* and *NtWriteFile* fall back on creating an IRP. (See the section "Fast I/O" in Chapter 11 for a more complete description of fast I/O.)

Memory Manager's Modified and Mapped Page Writer

The memory manager's modified and mapped page writer threads wake up periodically and when available memory runs low to flush modified pages. The threads call *IoAsynchronousPageWrite* to create IRPs of type IRP_MJ_WRITE and write pages to either a paging file or a file that was modified after being mapped. Like the IRPs that *MiDispatchFault* creates, these IRPs are flagged as noncached and paging I/O. Thus, an FSD bypasses the file system cache and issues IRPs directly to a storage driver to write the memory to disk.

Cache Manager's Lazy Writer

The cache manager's lazy writer thread also plays a role in writing modified pages because it periodically flushes views of file sections mapped in the cache that it knows are dirty. The flush operation, which the cache manager performs by calling *MmFlushSection*, triggers the memory manager to write any modified pages in the portion of the section being flushed to disk. Like the modified and mapped page writers, *MmFlushSection* uses *IoSynchronousPageWrite* to send the data to the FSD.

Cache Manager's Read-Ahead Thread

The cache manager includes a thread that is responsible for attempting to read data from files before an application, a driver, or a system thread explicitly requests it. The read-ahead thread uses the history of read operations that were performed on a file, which are stored in a file object's private cache map, to determine how much data to read. When the thread performs a read-ahead, it simply maps the portion of the file it wants to read into the cache (allocating VACBs as necessary) and touches the mapped data. The page faults caused by the memory accesses invoke the page fault handler, which reads the pages into the system's working set.

Memory Manager's Page Fault Handler

We described how the page fault handler is used in the context of explicit file I/O and cache manager read-ahead, but it is also invoked whenever any application accesses virtual memory that is a view of a mapped file and encounters pages that represent portions of a file that aren't part of the application's working set. The memory manager's *MmAccessFault* handler follows the same steps it does when the cache manager generates a page fault from *CcCopyRead* or *CcCopyWrite*, sending IRPs via *IoPageRead* to the file system on which the file is stored.

File System Filter Drivers

A filter driver that layers over a file system driver is called a *file system filter driver*. (See Chapter 9 for more information on filter drivers.) The ability to see all file system requests and optionally modify or complete them enables a range of applications, including remote file replication services, file encryption, efficient backup, and licensing. Every commercial on-access virus scanner includes a file system filter driver that intercepts IRPs that deliver IRP_MJ_CREATE commands that issue whenever an application opens a file. Before propagating the IRP to the file system driver to which the command is directed, the virus scanner examines the file being opened to ensure it's clean of a virus. If the file is clean, the virus scanner passes the IRP on, but if the file is infected the virus scanner communicates with its associated Windows service process to quarantine or clean the file. If the file can't be cleaned, the driver fails the IRP (typically with an access-denied error) so that the virus cannot become active.

In this section, we'll describe the operation of two specific file system filter drivers: Filemon and System Restore. Filemon, a file system activity monitoring utility from *www.sysinternals.com* that has been used throughout this book, is an example of a passive filter driver, which is one that does not modify the flow of IRPs between applications and file system drivers. System Restore, a feature that was introduced in Windows XP, uses a file system filter driver to watch for changes to key system files and make backups so that the files can be returned to the state they had at particular points in time called *restore points*.

Note Windows XP Service Pack 2 and Windows Server 2003 include the Filesystem Filter Manager (\Windows\System32\Drivers\Fltmgr.sys), which will also be made available for Windows 2000, as part of a port/miniport model for file system filter drivers. The Filesystem Filter Manager greatly simplifies the development of filter drivers by interfacing a filter miniport driver to the Windows I/O system and providing services for querying filenames, attaching to volumes, and interacting with other filters. Vendors, including Microsoft, will write new file system filters and migrate existing filters to the framework provided by the Filesystem Filter Manager.

Filemon

Filemon works by extracting a file system filter device driver (Filem.sys) from its executable image (Filemon.exe) the first time you run it after a boot, installing the driver in memory, and then deleting the driver image from disk. Through the Filemon GUI, you can direct it to monitor file system activity on local volumes that have assigned drive letters, network shares, named pipes, and mail slots. When the driver receives a command to start monitoring a volume, it creates a filter device object and attaches it to the device object that represents a mounted file system on the volume. For example, if the NTFS driver had mounted a volume, Filemon's driver would attach, using the I/O manager function *IoAttachDeviceToDeviceStackSafe*, its own device object to that of NTFS. After an attach operation, the I/O manager redirects an IRP targeted at the underlying device object to the driver owning the attached device, in this case Filemon.

When the Filemon driver intercepts an IRP, it records information about the IRP's command, including target file name and other parameters specific to the command (such as read and write lengths and offsets) to a nonpaged kernel buffer. Twice a second the Filemon GUI sends an IRP to Filemon's interface device object, which requests a copy of the buffer containing the latest activity and displays the activity in its output window. Filemon's use is described further in the "Troubleshooting File System Problems" section later in this chapter.

System Restore

System Restore, which originally appeared in a more rudimentary form in Windows Me (Millennium Edition), provides a way to restore a Windows XP system to a previously known good point that would otherwise require you to reinstall an application or even the entire operating system. (System Restore is not available on Windows 2000 or Windows Server 2003.) For example, if you install one or more applications or make other system file changes, registry changes, or both that cause applications to fail, you can use System Restore to revert the system files and the Registry to the state it had before the change occurred. System Restore is especially useful when you install an application that makes changes you would like to undo. Windows XP–compatible setup applications integrate with System Restore to create a "restore point" before an installation begins.

System Restore's core lies in a service named SrService, which executes from a DLL (\Windows\System32\Srsvc.dll) running in an instance of a generic service host (\Windows\System32\Svchost.exe) process. (See Chapter 4 for a description of Svchost.) The service's role is both to automatically create restore points and to export an API so that other applications—such as setup programs—can manually initiate restore point creation. System Restore reads its configuration parameters from HKLM\Software\Microsoft\System Restore, including ones that specify how much disk space must be available for it to operate and at what interval automated restore-point creation occurs. By default, the service creates a restore point prior to the installation of an unsigned device driver and tries to create an automatic checkpoint every 24 hours. (See Chapter 9 for information on driver signing.) If the DWORD registry value RPGlobalInterval is set under System Restore's parameter key, HKLM\System\CurrentControlSet\Services\SR\Parameters, it overrides this interval and specifies the minimum time interval in seconds between automatic restore points.

When the System Restore service creates a new restore point, it creates a restore point directory and then "snapshots" a set of critical system files, including the system and user-profile Registry hives, WMI configuration information, the IIS metabase file (if IIS is installed), and the COM registration database. Then the system restore file system filter driver, \Windows\System32\Drivers\Sr.sys, begins to track changes to files and directories, saving copies of files that are being deleted or modified in the restore point, and noting other changes, such as directory creation and deletion, in a restore point tracking log.

Restore point data is maintained on a per-volume basis, so tracking logs and saved files are stored under the \System Volume Information_restore{*XX-XXX-XXX* } directory (where the *X*s represent the computer's system-assigned GUID) of a file's original volume. The restore directory contains restore-point subdirectories having names in the form RP*n*, where *n* is a restore point's unique identifier. Files that make up a restore point's initial snapshot are stored under a restore point's Snapshot directory.

Backup files copied by the System Restore driver are given unique names, such as A0000135.dll, in an appropriate restore-point directory that reflect the assignment of an identifier and the preservation of the file's original extension. A restore point can have multiple tracking logs, each having a name like change.log.*N*, where *N* is a unique tracking log ID. A tracking log contains records that store enough information regarding a change to a file or directory for the change to be undone. For example, if a file was deleted, the tracking log entry for that operation would store the copy's name in the restore point (for example, A0000135.dll) and the file's original long and short file names. The System Restore driver starts a new tracking log when a current one grows larger than 1 MB. Figure 12-10 depicts the flow of file system requests as the System Restore driver updates a restore point in response to modifications.

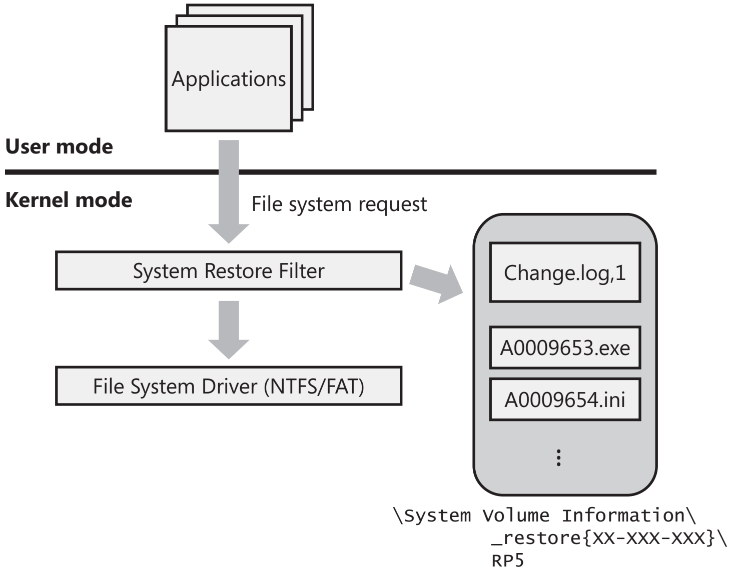
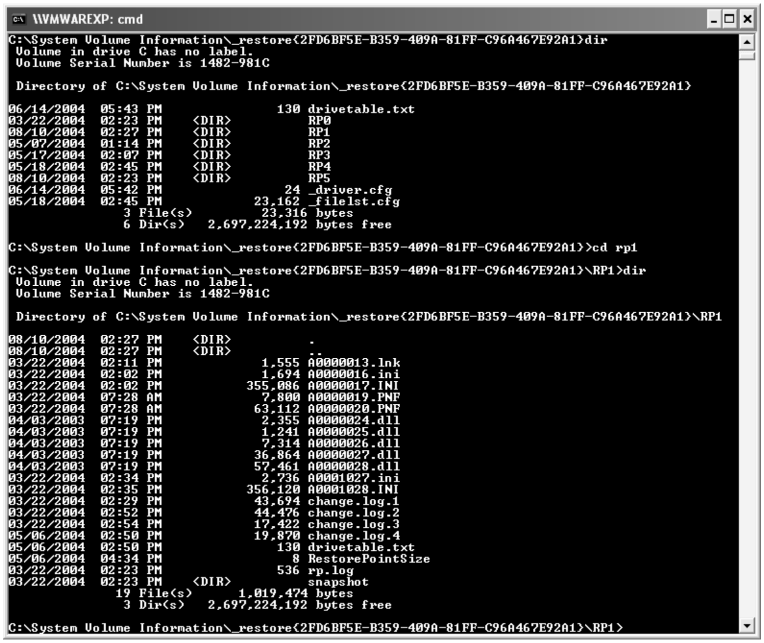


Figure 12-10 System Restore filter driver operation

Figure 12-11 shows a screen shot of a System Restore directory, which includes several restore point subdirectories, as well as the contents of the subdirectory corresponding to restore point 1. Note that the \System Volume Information directories are not accessible by user or even administrator accounts, but they are by the local system account. To view the contents of this folder, follow these steps with the PsExec utility from *www.sysinternals.com*:

```
C:\WINDOWS\SYSTEM32>psexec -s cmd

PsExec v1.55 - Execute processes remotely
Copyright (C) 2001-2004 Mark Russinovich
Sysinternals - www.sysinternals.com


Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\WINDOWS\system32>cd \system*

C:\System Volume Information>cd _rest*

C:\System Volume Information\_restore{987E0331-0F01-427C-A58A-7A2E4AABF84D}>
```

Once in the System Restore directory, you can examine its contents with the DIR command or navigate into directories associated with restore points.



```
C:\System Volume Information\_restore{2FD6BF5E-B359-409A-81FF-C96A467E92A1}>dir
 Volume in drive C has no label.
 Volume Serial Number is 1482-981C

 Directory of C:\System Volume Information\_restore{2FD6BF5E-B359-409A-81FF-C96A467E92A1}

06/14/2004  05:43 PM               130 drivetable.txt
03/22/2004  02:23 PM    <DIR>          RP0
08/10/2004  02:27 PM    <DIR>          RP1
05/07/2004  01:14 PM    <DIR>          RP2
05/17/2004  02:07 PM    <DIR>          RP3
05/18/2004  02:45 PM    <DIR>          RP4
08/10/2004  02:23 PM    <DIR>          RP5
06/14/2004  05:42 PM                24 _driver.cfg
05/18/2004  02:45 PM            23,162 _filelst.cfg
               3 File(s)         23,316 bytes
               6 Dir(s)   2,697,224,192 bytes free

C:\System Volume Information\_restore{2FD6BF5E-B359-409A-81FF-C96A467E92A1}>cd rp1

C:\System Volume Information\_restore{2FD6BF5E-B359-409A-81FF-C96A467E92A1}\RP1>dir
 Volume in drive C has no label.
 Volume Serial Number is 1482-981C

 Directory of C:\System Volume Information\_restore{2FD6BF5E-B359-409A-81FF-C96A467E92A1}\RP1

08/10/2004  02:27 PM    <DIR>          .
08/10/2004  02:27 PM    <DIR>          ..
03/22/2004  02:11 PM             1,555 A0000013.lnk
03/22/2004  02:02 PM             1,694 A0000016.ini
03/22/2004  02:02 PM           355,086 A0000017.INI
03/22/2004  07:28 AM             7,800 A0000019.PNF
03/22/2004  07:28 AM            63,112 A0000020.PNF
04/03/2003  07:19 PM             2,355 A0000024.dll
04/03/2003  07:19 PM             1,241 A0000025.dll
04/03/2003  07:19 PM             7,314 A0000026.dll
04/03/2003  07:19 PM            36,864 A0000027.dll
04/03/2003  07:19 PM            57,461 A0000028.dll
03/22/2004  02:34 PM             2,736 A0001027.ini
03/22/2004  02:35 PM           356,120 A0001028.INI
03/22/2004  02:29 PM            43,694 change.log.1
03/22/2004  02:52 PM            44,476 change.log.2
03/22/2004  02:54 PM            17,422 change.log.3
05/06/2004  02:50 PM            19,870 change.log.4
05/06/2004  02:50 PM               130 drivetable.txt
05/06/2004  04:34 PM                 8 RestorePointSize
03/22/2004  02:23 PM               536 rp.log
03/22/2004  02:23 PM    <DIR>          snapshot
              19 File(s)      1,019,474 bytes
               3 Dir(s)   2,697,224,192 bytes free

C:\System Volume Information\_restore{2FD6BF5E-B359-409A-81FF-C96A467E92A1}\RP1>
```

Figure 12-11 System Restore directory and restore point contents

The restore point directory on the boot volume also stores a file named _filelst.cfg, which is a binary file that includes the extensions of files for which changes should be stored in a restore point and the list of directories—such as those that store temporary files—for which changes should be ignored. This list, which is documented in the Platform SDK, directs System Restore to track only nondata files. For example, you wouldn't want an important Microsoft Word document to be deleted just because you rolled back the system to correct an application configuration problem.

EXPERIMENT: Looking at System Restore Filter Device Objects

To monitor changes to files and directories, the System Restore filter driver must attach filter device objects to the FAT and NTFS device objects representing volumes. In addition, it attaches a filter device object to the device objects representing file system drivers so that it can become aware of new volumes as they are mounted by the file system and then subsequently attach filter device objects to them. You can see System Restore's device objects with a kernel debugger:

```
lkd> !drvobj \filesystem\sr
Driver object (81543850) is for:
 \FileSystem\sr
Driver Extension List: (id , addr)

Device Object list:
814ee370  81542dd0  81543728
```

In this sample output, the System Restore driver has three device objects. The last one in the list is named *SystemRestore*, so it serves as the interface to which the user-mode components of System Restore direct commands:

```
lkd> !devobj 81543728
Device object (81543728) is for:
 SystemRestore \FileSystem\sr DriverObject 81543850
Current Irp 00000000 RefCount 1 Type 00000022 Flags 00000040
Dacl e128feac DevExt 00000000 DevObjExt 815437e0
ExtensionFlags (0x80000000)  DOE_DESIGNATED_FDO
Device queue is not busy.
```

The first and second objects are attached to NTFS file system device objects:

```
lkd> !devobj 814ee370
Device object (814ee370) is for:
  \FileSystem\sr DriverObject 81543850
Current Irp 00000000 RefCount 0 Type 00000008 Flags 00000000
DevExt 814ee428 DevObjExt 814ee570
ExtensionFlags (0x80000000)  DOE_DESIGNATED_FDO
AttachedTo (Lower) 81532020 \FileSystem\Ntfs
Device queue is not busy.
lkd> !devobj 81542dd0
Device object (81542dd0) is for:
  \FileSystem\sr DriverObject 81543850
Current Irp 00000000 RefCount 0 Type 00000008 Flags 00000000
DevExt 81542e88 DevObjExt 81542fd0
ExtensionFlags (0x80000000)  DOE_DESIGNATED_FDO
AttachedTo (Lower) 815432e8 \FileSystem\Ntfs
Device queue is not busy.
```

One of the NTFS device objects is the NTFS file system driver's interface device because its name is NTFS:

```
lkd> !devobj 815432e8
Device object (815432e8) is for:
 Ntfs \FileSystem\Ntfs DriverObject 81543410
Current Irp 00000000 RefCount 1 Type 00000008 Flags 00000040
Dacl e1297154 DevExt 00000000 DevObjExt 815433a0
ExtensionFlags (0x80000000)  DOE_DESIGNATED_FDO
AttachedDevice (Upper) 81542dd0 \FileSystem\sr
Device queue is not busy.
```

The other represents the mounted NTFS volume on C:, the system's only volume, so it does not have a name:

```
lkd> !devobj 81532020
Device object (81532020) is for:
  \FileSystem\Ntfs DriverObject 81543410
Current Irp 00000000 RefCount 0 Type 00000008 Flags 00000000
DevExt 815320d8 DevObjExt 81532880
ExtensionFlags (0x80000000)  DOE_DESIGNATED_FDO
AttachedDevice (Upper) 814ee370 \FileSystem\sr
Device queue is not busy.
```

When the user directs the system to perform a restore, the System Restore Wizard (\Windows\System32\Restore\Rstrui.exe) creates a DWORD value named *RestoreInProgress* under the System Restore parameters key and sets it to 1. Then it initiates a system shutdown with reboot by calling the Windows *ExitWindowsEx* function. After the reboot, the WinLogon process (\Windows\System32\Winlogon.exe) realizes that it should perform a restore, and then copies saved files from the Restore Point's directory to their original locations and uses the log files to undo file system changes to files and directories. When the process is complete, the boot continues. Besides making restores safer, the reboot is necessary to activate restored Registry hives.

The Platform SDK documents two System Restore–related APIs, *SRSetRestorePoint* and *SRRemoveRestorePoint*, for use by installation programs, and developers should examine the file extensions that their applications use in light of System Restore. Files that store user data should not have extensions matching those protected by System Restore; otherwise, users could lose data when rolling back to a restore point.

Troubleshooting File System Problems

Chapter 4 describes the way that the system and applications store data in the registry, making registry-related problems such as misconfigured security and missing registry values and keys the source of many system and application failures. The system and applications also use files to store data, and they access executable and DLL image files. Misconfigured NTFS security and missing files or directories are therefore also a common source of system and application failures. This is because the system and applications often make assumptions about what they should be able to access and then misbehave in unexpected ways when the assumptions are violated.

Filemon shows all file activity as it occurs, which makes it an ideal tool for troubleshooting file system–related system and application failures. Filemon's user interface is virtually identical to Regmon's, and Filemon includes the same filtering, highlighting and search features as Regmon. To run Filemon the first time on a system, an account must have the same privileges as to run Regmon: Load Driver and Debug. After loading, the driver remains resident, so subsequent executions require only the Debug privilege.

Filemon Basic vs. Advanced Modes

When you run Filemon, it starts in basic mode, which shows the file system activity most often useful for troubleshooting. When in basic mode, Filemon omits certain file system operations from display, including:

- Accesses to NTFS metadata files
- Activity that occurs in the System process
- I/O to the paging file

- I/O generated by the Filemon process
- Fast I/O failures

While in basic mode, Filemon also reports file I/O operations with friendly names rather than the IRP types used to represent them. For example, both IRP_MJ_WRITE and FASTIO_WRITE operations display as Write, and IRP_MJ_CREATE operations show as Open if they represent an open operation and as Create for the creation of new files.

EXPERIMENT: Viewing File System Activity on an Idle System

Windows file system drivers implement support for *file change notification*, which enables applications to request notification of file system changes without polling for them. The Windows functions for doing so include *ReadDirectoryChangesW* and the *FindFirstChangeNotification*, *FindNextChangeNofification* pair. When you run Filemon on a system that's idle, you should therefore not see the repeated accesses to files or directories because that activity unnecessarily negatively affects a system's overall performance.

Run Filemon, and after several seconds examine the output log to see whether you can spot polling behavior. Right-click on an output line associated with polling and choose Process Properties from the context menu to view details of the process performing the activity.

Filemon Troubleshooting Techniques

The two basic Filemon troubleshooting techniques are identical to those of Regmon: looking at the last thing in a Filemon trace that an application did before it failed, or comparing a Filemon trace of a failing application with a trace from a working system. See the section "Regmon Troubleshooting Techniques" in Chapter 4 for more information on these techniques.

Entries in a Filemon trace that have values of FILE NOT FOUND, NO SUCH FILE, PATH NOT FOUND, SHARING VIOLATION, and ACCESS DENIED in the Result column are ones that you should investigate. The first three are reported when an application or the system attempts to open a nonexistent file or directory. In many cases, these errors do not indicate a serious problem. When you execute a program from the Start menu's Run dialog box without specifying its full path, for instance, Explorer will search the directories listed in the system PATH environment variable for the image file until it locates the file or has searched all the listed directories. Each attempt to find the image in a directory that does not contain it results in a Filemon output line similar to this:

```
5:28:26 PMEXPLORER.EXE:1568FASTIO_QUERY_OPENC:\Documents and Settings\mark.AUSTIN\Start Menu
\test.exe    FILE NOT FOUND    Attributes: Error
```

Access-denied errors are a common source of file system–related application failures, and they occur when an application does not have permission to open the file or directory for the access types it desires. Some applications do not check error codes or perform error recovery, and they fail by crashing or terminating; others display misleading error messages that mask the root cause of the error.

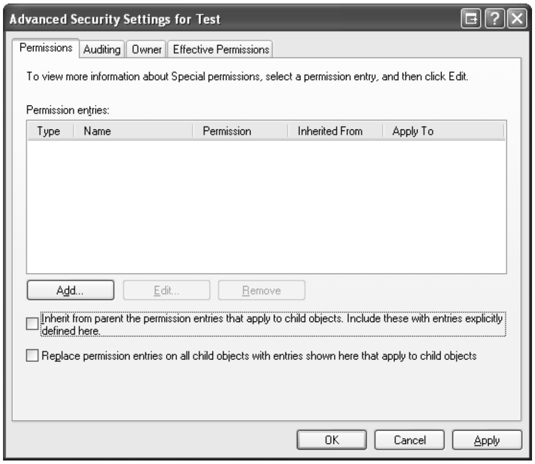
Buffer-overflow exploits are a serious security concern, but a code result of BUFFER OVERFLOW is simply a file system driver's way to indicate to an application that the buffer it specified to store result data was too small to hold the data. Application developers use this behavior to determine how large a buffer should be because the file system driver also returns the size of the buffer required to store the data. Operations with a buffer overflow result are usually followed by the same operation with a successful result.

EXPERIMENT: Seeing an Error's Root Cause with Filemon

Applications sometimes present error messages in response to an error condition that do not reveal the root cause of the error. These error messages can be frustrating because they might lead you to spend time diagnosing or resolving problems that do not exist. If the error message is related to a file system issue, Filemon will show you what underlying errors might have occurred prior to the appearance of an error message.

In this experiment, you'll set permission on a directory and then perform a file save operation in Notepad that results in a misleading error message. Filemon's trace shows the actual error and the source of the message displayed in Notepad's error dialog box.

1. Run Filemon, and set the include filter to "notepad.exe".
2. Open Explorer, and create a directory named "Test" in a directory on an NTFS volume. (In this example, the root directory was used.)
3. Edit the security permissions on the Test directory to remove all access. This might require you to open the Advanced Security Settings dialog box and use the settings on the Permissions tab to remove inherited security.



To view more information about Special permissions, select a permission entry, and then click Edit.

Permission entries:

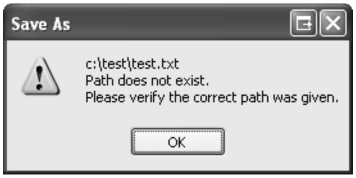
Type	Name	Permission	Inherited From	Apply To

☐ Inherit from parent the permission entries that apply to child objects. Include these with entries explicitly defined here.

☐ Replace permission entries on all child objects with entries shown here that apply to child objects

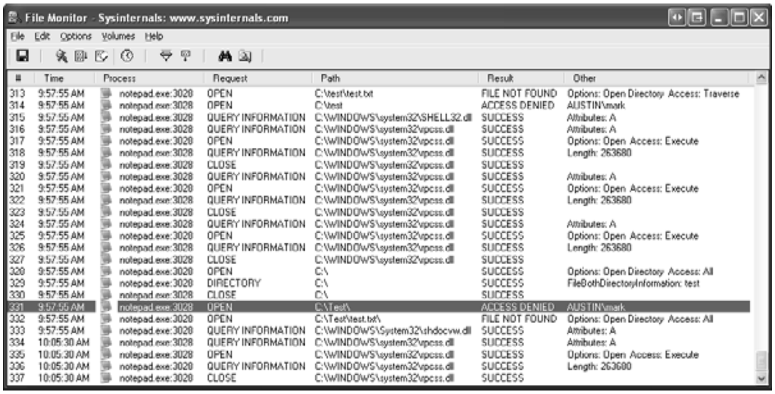
When you apply the modified security, Explorer should warn you that no one will have access to the folder.

4. Run Notepad, and enter some text into its window. Then select the Save entry in the File menu. In the File Name field of the Save dialog box, enter **c:\test\test.txt** (assuming the folder you created is on the volume C:).
5. Notepad will display the following error message:



c:\test\test.txt
Path does not exist.
Please verify the correct path was given.

6. The message implies that C:\Test does not exist.
7. The Filemon trace you see should look something like this:

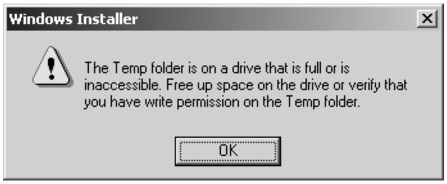

File Monitor - Sysinternals: www.sysinternals.com
File Edit Options Volumes Help

#	Time	Process	Request	Path	Result	Other
313	9:57:55 AM	notepad.exe:3028	OPEN	C:\test\test.txt	FILE NOT FOUND	Options: Open Directory Access: Traverse
314	9:57:55 AM	notepad.exe:3028	OPEN	C:\test	ACCESS DENIED	AUSTIN\mark
315	9:57:55 AM	notepad.exe:3028	QUERY INFORMATION	C:\WINDOWS\system32\SHELL32.dll	SUCCESS	Attributes: A
316	9:57:55 AM	notepad.exe:3028	QUERY INFORMATION	C:\WINDOWS\system32\vpcss.dll	SUCCESS	Attributes: A
317	9:57:55 AM	notepad.exe:3028	OPEN	C:\WINDOWS\system32\vpcss.dll	SUCCESS	Options: Open Access: Execute
318	9:57:55 AM	notepad.exe:3028	QUERY INFORMATION	C:\WINDOWS\system32\vpcss.dll	SUCCESS	Length: 263680
319	9:57:55 AM	notepad.exe:3028	CLOSE	C:\WINDOWS\system32\vpcss.dll	SUCCESS	
320	9:57:55 AM	notepad.exe:3028	QUERY INFORMATION	C:\WINDOWS\system32\vpcss.dll	SUCCESS	Attributes: A
321	9:57:55 AM	notepad.exe:3028	OPEN	C:\WINDOWS\system32\vpcss.dll	SUCCESS	Options: Open Access: Execute
322	9:57:55 AM	notepad.exe:3028	QUERY INFORMATION	C:\WINDOWS\system32\vpcss.dll	SUCCESS	Length: 263680
323	9:57:55 AM	notepad.exe:3028	CLOSE	C:\WINDOWS\system32\vpcss.dll	SUCCESS	
324	9:57:55 AM	notepad.exe:3028	QUERY INFORMATION	C:\WINDOWS\system32\vpcss.dll	SUCCESS	Attributes: A
325	9:57:55 AM	notepad.exe:3028	OPEN	C:\WINDOWS\system32\vpcss.dll	SUCCESS	Options: Open Access: Execute
326	9:57:55 AM	notepad.exe:3028	QUERY INFORMATION	C:\WINDOWS\system32\vpcss.dll	SUCCESS	Length: 263680
327	9:57:55 AM	notepad.exe:3028	CLOSE	C:\WINDOWS\system32\vpcss.dll	SUCCESS	
328	9:57:55 AM	notepad.exe:3028	OPEN	C:\	SUCCESS	Options: Open Directory Access: All
329	9:57:55 AM	notepad.exe:3028	DIRECTORY	C:\	SUCCESS	FileBothDirectoryInformation: test
330	9:57:55 AM	notepad.exe:3028	CLOSE	C:\	SUCCESS	
331	9:57:55 AM	notepad.exe:3028	OPEN	C:\Test\	ACCESS DENIED	AUSTIN\mark
332	9:57:55 AM	notepad.exe:3028	OPEN	C:\Test\test.txt\	FILE NOT FOUND	Options: Open Directory Access: All
333	9:57:55 AM	notepad.exe:3028	QUERY INFORMATION	C:\WINDOWS\System32\shdocvw.dll	SUCCESS	Attributes: A
334	10:05:30 AM	notepad.exe:3028	QUERY INFORMATION	C:\WINDOWS\system32\vpcss.dll	SUCCESS	Attributes: A
335	10:05:30 AM	notepad.exe:3028	OPEN	C:\WINDOWS\system32\vpcss.dll	SUCCESS	Options: Open Access: Execute
336	10:05:30 AM	notepad.exe:3028	QUERY INFORMATION	C:\WINDOWS\system32\vpcss.dll	SUCCESS	Length: 263680
337	10:05:30 AM	notepad.exe:3028	CLOSE	C:\WINDOWS\system32\vpcss.dll	SUCCESS	

The output at the bottom of the figure was generated immediately prior to the appearance of the error message. Line 331 shows that Notepad tried to open C:\Test and got an access-denied error. Immediately afterward on line 332, it tried to open the C:\Test\Test.txt directory (as indicated by the Directory option in the Other column of that line) and received a file-not-found error because the directory does not exist. The error message Notepad displays, "Path does not exist", is consistent with a file-not-found error, not an access-denied error. So it appears that Notepad first tried to open the directory, and when that failed it assumed for some reason that the name C:\Test\Test.txt was the name of a directory instead of a file. When it couldn't open that directory, Notepad presented the error message, but the root cause, which Filemon reveals, is the access denied error.

Filemon has been used extensively within Microsoft and other organizations to solve difficult or nearly impossible-to-diagnose problems. One example of Filemon being used to troubleshoot a problem revealed the root cause of a misleading error message generated by the

Windows Installer service. When the user tried to install an application using its Windows Installer Package file, the Windows Installer service reported the error shown in Figure 12-12, which states that the service cannot write to the Temp folder. The user verified that, contrary to the error message's claims, his profile's temporary directory (that he obtained by typing **set temp** in a console window) was located on a volume with adequate free space and had default permissions.

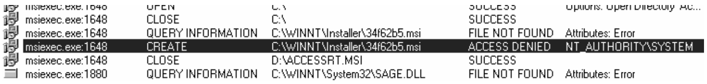


Windows Installer

The Temp folder is on a drive that is full or is inaccessible. Free up space on the drive or verify that you have write permission on the Temp folder.

Figure 12-12 Microsoft Windows Installer error message

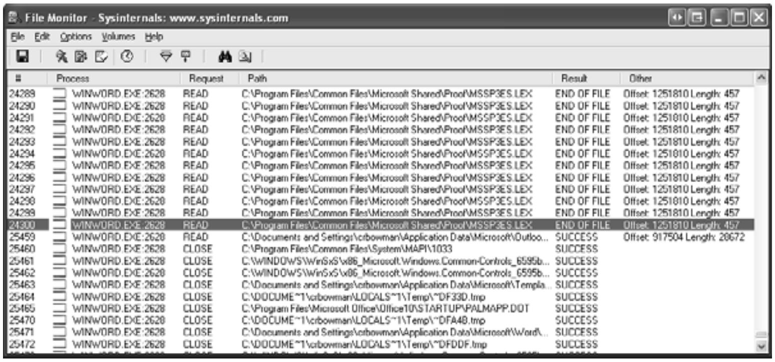
The user ran Filemon to capture a trace of the file system activity leading up to the error and identified the highlighted line in Figure 12-13 as the root cause of the error. The conclusion that the user drew from the trace is that the Windows Installer service was referring to \Windows\Installer, not his profile's temporary directory, as the Temp folder in the error message. The access-denied line in the output reports that the Windows Installer service was running in the local system account, so the user modified the permissions on \Windows\Installer to allow the local system account write access and resolved the problem.



```
msiexec.exe:1648   OPEN                C:\                              SUCCESS          Options: Open Directory Ac...
msiexec.exe:1648   CLOSE               C:\                              SUCCESS
msiexec.exe:1648   QUERY INFORMATION   C:\WINNT\Installer\34f62b5.msi   FILE NOT FOUND   Attributes: Error
msiexec.exe:1648   CREATE              C:\WINNT\Installer\34f62b5.msi   ACCESS DENIED    NT_AUTHORITY\SYSTEM
msiexec.exe:1648   CLOSE               D:\ACCESSRT.MSI                  SUCCESS
msiexec.exe:1880   QUERY INFORMATION   C:\WINNT\System32\SAGE.DLL       FILE NOT FOUND   Attributes: Error
```

Figure 12-13 Microsoft Windows Installer Service Filemon trace

Another Filemon troubleshooting example involved Microsoft Word. The user in this case would launch Word and type for a few seconds, only to have the Word window close without notice. A Filemon trace of the scenario, part of which is shown in Figure 12-14, shows Word repeatedly reading from the same part of a file named Mssp3es.lex immediately before exiting. (When a process exits, the system closes all its handles, which you can see happening from line 25460 onward.) The user determined that .lex files are related to Microsoft Office Proofing Tools and reinstalled that component, which resolved the problem.

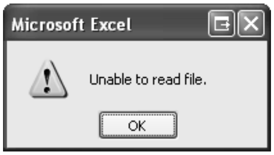


File Monitor - Sysinternals: www.sysinternals.com

#	Process	Request	Path	Result	Other
24289	WINWORD.EXE:2628	READ	C:\Program Files\Common Files\Microsoft Shared\Proof\MSSP3ES.LEX	END OF FILE	Offset: 1251810 Length: 457
24290	WINWORD.EXE:2628	READ	C:\Program Files\Common Files\Microsoft Shared\Proof\MSSP3ES.LEX	END OF FILE	Offset: 1251810 Length: 457
24291	WINWORD.EXE:2628	READ	C:\Program Files\Common Files\Microsoft Shared\Proof\MSSP3ES.LEX	END OF FILE	Offset: 1251810 Length: 457
24292	WINWORD.EXE:2628	READ	C:\Program Files\Common Files\Microsoft Shared\Proof\MSSP3ES.LEX	END OF FILE	Offset: 1251810 Length: 457
24293	WINWORD.EXE:2628	READ	C:\Program Files\Common Files\Microsoft Shared\Proof\MSSP3ES.LEX	END OF FILE	Offset: 1251810 Length: 457
24294	WINWORD.EXE:2628	READ	C:\Program Files\Common Files\Microsoft Shared\Proof\MSSP3ES.LEX	END OF FILE	Offset: 1251810 Length: 457
24295	WINWORD.EXE:2628	READ	C:\Program Files\Common Files\Microsoft Shared\Proof\MSSP3ES.LEX	END OF FILE	Offset: 1251810 Length: 457
24296	WINWORD.EXE:2628	READ	C:\Program Files\Common Files\Microsoft Shared\Proof\MSSP3ES.LEX	END OF FILE	Offset: 1251810 Length: 457
24297	WINWORD.EXE:2628	READ	C:\Program Files\Common Files\Microsoft Shared\Proof\MSSP3ES.LEX	END OF FILE	Offset: 1251810 Length: 457
24298	WINWORD.EXE:2628	READ	C:\Program Files\Common Files\Microsoft Shared\Proof\MSSP3ES.LEX	END OF FILE	Offset: 1251810 Length: 457
24299	WINWORD.EXE:2628	READ	C:\Program Files\Common Files\Microsoft Shared\Proof\MSSP3ES.LEX	END OF FILE	Offset: 1251810 Length: 457
24300	WINWORD.EXE:2628	READ	C:\Program Files\Common Files\Microsoft Shared\Proof\MSSP3ES.LEX	END OF FILE	Offset: 1251810 Length: 457
25459	WINWORD.EXE:2628	READ	C:\Documents and Settings\crbowman\Application Data\Microsoft\Outloo...	SUCCESS	Offset: 917504 Length: 20672
25460	WINWORD.EXE:2628	CLOSE	C:\Program Files\Common Files\System\MAPI\1033	SUCCESS	
25461	WINWORD.EXE:2628	CLOSE	C:\WINDOWS\WinSxS\x86_Microsoft.Windows.Common-Controls_6595b...	SUCCESS	
25462	WINWORD.EXE:2628	CLOSE	C:\WINDOWS\WinSxS\x86_Microsoft.Windows.Common-Controls_6595b...	SUCCESS	
25463	WINWORD.EXE:2628	CLOSE	C:\Documents and Settings\crbowman\Application Data\Microsoft\Templa...	SUCCESS	
25464	WINWORD.EXE:2628	CLOSE	C:\DOCUME~1\crbowman\LOCALS~1\Temp\~DF33D.tmp	SUCCESS	
25465	WINWORD.EXE:2628	CLOSE	C:\Program Files\Microsoft Office\Office10\STARTUP\PALMAPP.DOT	SUCCESS	
25470	WINWORD.EXE:2628	CLOSE	C:\DOCUME~1\crbowman\LOCALS~1\Temp\~DFA48.tmp	SUCCESS	
25471	WINWORD.EXE:2628	CLOSE	C:\Documents and Settings\crbowman\Application Data\Microsoft\Word\...	SUCCESS	
25472	WINWORD.EXE:2628	CLOSE	C:\DOCUME~1\crbowman\LOCALS~1\Temp\~DFDDF.tmp	SUCCESS	

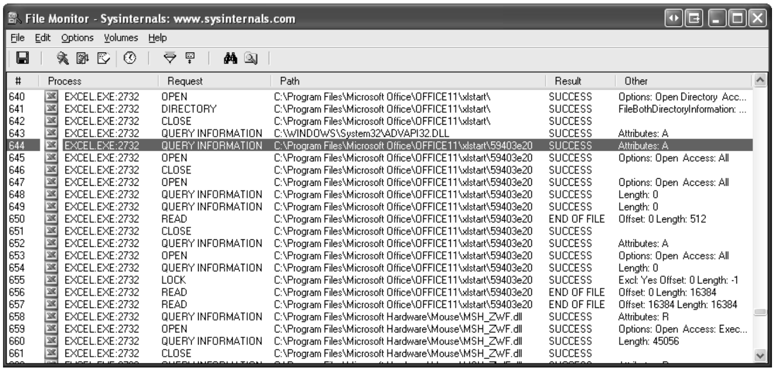
Figure 12-14 Microsoft Word reading a .lex file

In another example, a user encountered the error message shown in Figure 12-15 every time she started Microsoft Excel. The Filemon trace in Figure 12-16 that the user captured during Excel's startup reveals Excel reading from a file named 59403e20 in a directory named Xlstart in the Microsoft Office installation directory. The user investigated the problem and learned from Excel's documentation that Excel automatically tries to open files stored in the Xlstart directory. However, the file visible in the trace was not an Excel file, which resulted in the error message. Deleting the file caused the errors to cease.


Microsoft Excel

Unable to read file.

Figure 12-15 Microsoft Excel startup error message

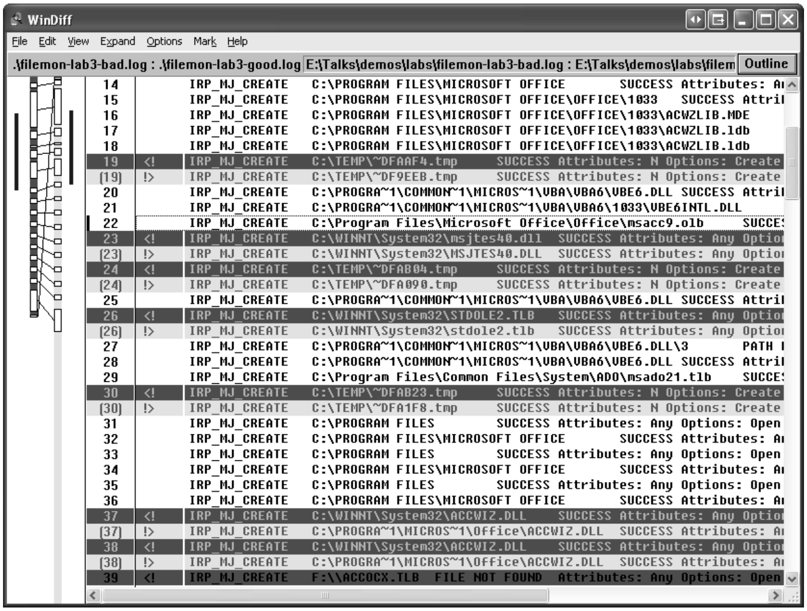


#	Process	Request	Path	Result	Other
640	EXCEL.EXE:2732	OPEN	C:\Program Files\Microsoft Office\OFFICE11\xlstart\	SUCCESS	Options: Open Directory Acc...
641	EXCEL.EXE:2732	DIRECTORY	C:\Program Files\Microsoft Office\OFFICE11\xlstart\	SUCCESS	FileBothDirectoryInformation: ...
642	EXCEL.EXE:2732	CLOSE	C:\Program Files\Microsoft Office\OFFICE11\xlstart\	SUCCESS	
643	EXCEL.EXE:2732	QUERY INFORMATION	C:\WINDOWS\System32\ADVAPI32.DLL	SUCCESS	Attributes: A
644	EXCEL.EXE:2732	QUERY INFORMATION	C:\Program Files\Microsoft Office\OFFICE11\xlstart\59403e20	SUCCESS	Attributes: A
645	EXCEL.EXE:2732	OPEN	C:\Program Files\Microsoft Office\OFFICE11\xlstart\59403e20	SUCCESS	Options: Open Access: All
646	EXCEL.EXE:2732	CLOSE	C:\Program Files\Microsoft Office\OFFICE11\xlstart\59403e20	SUCCESS	
647	EXCEL.EXE:2732	OPEN	C:\Program Files\Microsoft Office\OFFICE11\xlstart\59403e20	SUCCESS	Options: Open Access: All
648	EXCEL.EXE:2732	QUERY INFORMATION	C:\Program Files\Microsoft Office\OFFICE11\xlstart\59403e20	SUCCESS	Length: 0
649	EXCEL.EXE:2732	QUERY INFORMATION	C:\Program Files\Microsoft Office\OFFICE11\xlstart\59403e20	SUCCESS	Length: 0
650	EXCEL.EXE:2732	READ	C:\Program Files\Microsoft Office\OFFICE11\xlstart\59403e20	END OF FILE	Offset: 0 Length: 512
651	EXCEL.EXE:2732	CLOSE	C:\Program Files\Microsoft Office\OFFICE11\xlstart\59403e20	SUCCESS	
652	EXCEL.EXE:2732	QUERY INFORMATION	C:\Program Files\Microsoft Office\OFFICE11\xlstart\59403e20	SUCCESS	Attributes: A
653	EXCEL.EXE:2732	OPEN	C:\Program Files\Microsoft Office\OFFICE11\xlstart\59403e20	SUCCESS	Options: Open Access: All
654	EXCEL.EXE:2732	QUERY INFORMATION	C:\Program Files\Microsoft Office\OFFICE11\xlstart\59403e20	SUCCESS	Length: 0
655	EXCEL.EXE:2732	LOCK	C:\Program Files\Microsoft Office\OFFICE11\xlstart\59403e20	SUCCESS	Excl: Yes Offset: 0 Length: -1
656	EXCEL.EXE:2732	READ	C:\Program Files\Microsoft Office\OFFICE11\xlstart\59403e20	END OF FILE	Offset: 0 Length: 16384
657	EXCEL.EXE:2732	READ	C:\Program Files\Microsoft Office\OFFICE11\xlstart\59403e20	END OF FILE	Offset: 16384 Length: 16384
658	EXCEL.EXE:2732	QUERY INFORMATION	C:\Program Files\Microsoft Hardware\Mouse\MSH_ZWF.dll	SUCCESS	Attributes: R
659	EXCEL.EXE:2732	OPEN	C:\Program Files\Microsoft Hardware\Mouse\MSH_ZWF.dll	SUCCESS	Options: Open Access: Exec...
660	EXCEL.EXE:2732	QUERY INFORMATION	C:\Program Files\Microsoft Hardware\Mouse\MSH_ZWF.dll	SUCCESS	Length: 45056
661	EXCEL.EXE:2732	CLOSE	C:\Program Files\Microsoft Hardware\Mouse\MSH_ZWF.dll	SUCCESS	

Figure 12-16 Filemon trace of Microsoft Excel startup

A final example of Filemon troubleshooting involves out-of-date DLLs. A particular user would run Microsoft Access 2000 and experience a hang when he tried to import a Microsoft Excel file. The user tried to import the same file on another system with Microsoft Access 2000 and was able to do so successfully. After capturing traces of the import operation on both systems and saving them to log files, the user compared the logs with Windiff. The results of the comparison are shown in Figure 12-17.

After discounting unimportant differences such as the different temporary file names seen in line 19 and the different casing of file names seen in line 26, the user determined that the first relevant difference between the traces is line 37. On the system where the import fails, Microsoft Access loads a copy of Accwiz.dll from the \Winnt\System32 directory, whereas on the system where the import succeeds it reads Accwiz.dll from \Progra~1\Files\Microsoft\Office. The user examined the Accwiz.dll copy in \Winnt\System32 and discovered that it was from an older version of Microsoft Access, but the system's DLL search order caused it to find that instance instead of the one in the Microsoft Access installation directory. Deleting that copy and registering the proper copy fixed the problem.



```
.\filemon-lab3-bad.log : .\filemon-lab3-good.log E:\Talks\demos\labs\filemon-lab3-bad.log : E:\Talks\demos\labs\filem
 14        IRP_MJ_CREATE   C:\PROGRAM FILES\MICROSOFT OFFICE       SUCCESS Attributes: A
 15        IRP_MJ_CREATE   C:\PROGRAM FILES\MICROSOFT OFFICE\OFFICE\1033   SUCCESS Attri
 16        IRP_MJ_CREATE   C:\PROGRAM FILES\MICROSOFT OFFICE\OFFICE\1033\ACWZLIB.MDE
 17        IRP_MJ_CREATE   C:\PROGRAM FILES\MICROSOFT OFFICE\OFFICE\1033\ACWZLIB.ldb
 18        IRP_MJ_CREATE   C:\PROGRAM FILES\MICROSOFT OFFICE\OFFICE\1033\ACWZLIB.ldb
 19   <!   IRP_MJ_CREATE   C:\TEMP\~DFAAF4.tmp      SUCCESS Attributes: N Options: Create
(19)  !>   IRP_MJ_CREATE   C:\TEMP\~DF9EEB.tmp      SUCCESS Attributes: N Options: Create
 20        IRP_MJ_CREATE   C:\PROGRA~1\COMMON~1\MICROS~1\VBA\VBA6\VBE6.DLL SUCCESS Attri
 21        IRP_MJ_CREATE   C:\PROGRA~1\COMMON~1\MICROS~1\VBA\VBA6\1033\VBE6INTL.DLL
 22        IRP_MJ_CREATE   C:\Program Files\Microsoft Office\Office\msacc9.olb     SUCCE
 23   <!   IRP_MJ_CREATE   C:\WINNT\System32\msjtes40.dll  SUCCESS Attributes: Any Optio
(23)  !>   IRP_MJ_CREATE   C:\WINNT\System32\MSJTES40.DLL  SUCCESS Attributes: Any Optio
 24   <!   IRP_MJ_CREATE   C:\TEMP\~DFAB04.tmp      SUCCESS Attributes: N Options: Create
(24)  !>   IRP_MJ_CREATE   C:\TEMP\~DFA090.tmp      SUCCESS Attributes: N Options: Create
 25        IRP_MJ_CREATE   C:\PROGRA~1\COMMON~1\MICROS~1\VBA\VBA6\VBE6.DLL SUCCESS Attri
 26   <!   IRP_MJ_CREATE   C:\WINNT\System32\STDOLE2.TLB   SUCCESS Attributes: Any Optio
(26)  !>   IRP_MJ_CREATE   C:\WINNT\System32\stdole2.tlb   SUCCESS Attributes: Any Optio
 27        IRP_MJ_CREATE   C:\PROGRA~1\COMMON~1\MICROS~1\VBA\VBA6\VBE6.DLL\3       PATH
 28        IRP_MJ_CREATE   C:\PROGRA~1\COMMON~1\MICROS~1\VBA\VBA6\VBE6.DLL SUCCESS Attri
 29        IRP_MJ_CREATE   C:\Program Files\Common Files\System\ADO\msado21.tlb    SUCCE
 30   <!   IRP_MJ_CREATE   C:\TEMP\~DFAB23.tmp      SUCCESS Attributes: N Options: Create
(30)  !>   IRP_MJ_CREATE   C:\TEMP\~DFA1F8.tmp      SUCCESS Attributes: N Options: Create
 31        IRP_MJ_CREATE   C:\PROGRAM FILES         SUCCESS Attributes: Any Options: Open
 32        IRP_MJ_CREATE   C:\PROGRAM FILES\MICROSOFT OFFICE       SUCCESS Attributes: A
 33        IRP_MJ_CREATE   C:\PROGRAM FILES         SUCCESS Attributes: Any Options: Open
 34        IRP_MJ_CREATE   C:\PROGRAM FILES\MICROSOFT OFFICE       SUCCESS Attributes: A
 35        IRP_MJ_CREATE   C:\PROGRAM FILES         SUCCESS Attributes: Any Options: Open
 36        IRP_MJ_CREATE   C:\PROGRAM FILES\MICROSOFT OFFICE       SUCCESS Attributes: A
 37   <!   IRP_MJ_CREATE   C:\WINNT\System32\ACCWIZ.DLL    SUCCESS Attributes: Any Optio
(37)  !>   IRP_MJ_CREATE   C:\PROGRA~1\MICROS~1\Office\ACCWIZ.DLL  SUCCESS Attributes: A
 38   <!   IRP_MJ_CREATE   C:\WINNT\System32\ACCWIZ.DLL    SUCCESS Attributes: Any Optio
(38)  !>   IRP_MJ_CREATE   C:\PROGRA~1\MICROS~1\Office\ACCWIZ.DLL  SUCCESS Attributes: A
 39   <!   IRP_MJ_CREATE   F:\\ACCOCX.TLB  FILE NOT FOUND  Attributes: Any Options: Open
```

Figure 12-17 Comparison of Microsoft Access trace logs

These are just a few examples showing how Filemon can be used to discover the root cause of file system problems that might not be reported clearly by the application. The remainder of the chapter focuses on the native Windows file system, NTFS.

NTFS Design Goals and Features

In the following section, we'll look at the requirements that drove the design of NTFS. Then in the subsequent section, we'll examine the advanced features of NTFS.

High-End File System Requirements

From the start, NTFS was designed to include features required of an enterprise-class file system. To minimize data loss in the face of an unexpected system outage or crash, a file system must ensure that the integrity of its metadata is guaranteed at all times; and to protect sensitive data from unauthorized access, a file system must have an integrated security model. Finally, a file system must allow for software-based data redundancy as a low-cost alternative to hardware-redundant solutions for protecting user data. In this section, you'll find out how NTFS implements each of these capabilities.

Recoverability

To address the requirement for reliable data storage and data access, NTFS provides file system recovery based on the concept of an *atomic transaction*. Atomic transactions are a technique for handling modifications to a database so that system failures don't affect the correctness or integrity of the database. The basic tenet of atomic transactions is that some database operations, called *transactions*, are all-or-nothing propositions. (A *transaction* is defined as an I/O operation that alters file system data or changes the volume's directory structure.) The separate disk updates that make up the transaction must be executed *atomically*—that is, once the transaction begins to execute, all its disk updates must be completed. If a system failure interrupts the transaction, the part that has been completed must be undone, or *rolled back*. The rollback operation returns the database to a previously known and consistent state, as if the transaction had never occurred.

NTFS uses atomic transactions to implement its file system recovery feature. If a program initiates an I/O operation that alters the structure of an NTFS volume—that is, changes the directory structure, extends a file, allocates space for a new file, and so on—NTFS treats that operation as an atomic transaction. It guarantees that the transaction is either completed or, if the system fails while executing the transaction, rolled back. The details of how NTFS does this are explained in the section "NTFS Recovery Support" later in the chapter.

In addition, NTFS uses redundant storage for vital file system information so that if a sector on the disk goes bad, NTFS can still access the volume's critical file system data. This redundancy of file system data contrasts with the on-disk structures of both the FAT file system and the HPFS file system (OS/2's native file system format), which have single sectors containing critical file system data. On these file systems, if a read error occurs in one of those sectors an entire volume is lost.

Security

Security in NTFS is derived directly from the Windows object model. Files and directories are protected from being accessed by unauthorized users. (For more information on Windows security, see Chapter 8.) An open file is implemented as a file object with a security descriptor stored on disk as a part of the file. Before a process can open a handle to any object, including a file object, the Windows security system verifies that the process has appropriate authorization to do so. The security descriptor, combined with the requirement that a user log on to the system and provide an identifying password, ensures that no process can access a file unless given specific permission to do so by a system administrator or by the file's owner. (For more information about security descriptors, see the section "Security Descriptors and Access Control" in Chapter 8, and for more details about file objects, see the section "File Objects" in Chapter 9.)

Data Redundancy and Fault Tolerance

In addition to recoverability of file system data, some customers require that their own data not be endangered by a power outage or catastrophic disk failure. The NTFS recovery capabilities do ensure that the file system on a volume remains accessible, but they make no guarantees for complete recovery of user files. Protection for applications that can't risk losing file data is provided through data redundancy.

Data redundancy for user files is implemented via the Windows layered driver model (explained in Chapter 9), which provides fault-tolerant disk support. NTFS communicates with a volume manager, which in turn communicates with a hard disk driver to write data to disk. A volume manager can *mirror*, or duplicate, data from one disk onto another disk so that a redundant copy can always be retrieved. This support is commonly called *RAID level 1*. Volume managers also allow data to be written in *stripes* across three or more disks, using the equivalent of one disk to maintain parity information. If the data on one disk is lost or becomes inaccessible, the driver can reconstruct the disk's contents by means of exclusive-OR operations. This support is called *RAID level 5*. (See Chapter 10 for more information on striped volumes, mirrored volumes, and RAID-5 volumes.)

Advanced Features of NTFS

In addition to NTFS being recoverable, secure, reliable, and efficient for mission-critical systems, it includes the following advanced features that allow it to support a broad range of applications. Some of these features are exposed as APIs for applications to leverage, and others are internal features:

- Multiple data streams
- Unicode-based names
- General indexing facility
- Dynamic bad-cluster remapping
- Hard links and junctions
- Compression and sparse files
- Change logging
- Per-user volume quotas
- Link tracking
- Encryption
- POSIX support
- Defragmentation
- Read-only support

The following sections provide an overview of these features.

Multiple Data Streams

In NTFS, each unit of information associated with a file—including its name, its owner, its time stamps, its contents, and so on—is implemented as a file attribute (NTFS object attribute). Each attribute consists of a single *stream*—that is, a simple sequence of bytes. This generic implementation makes it easy to add more attributes (and therefore more streams) to a file.

Because a file's data is "just another attribute" of the file and because new attributes can be added, NTFS files (and file directories) can contain multiple data streams.

An NTFS file has one default data stream, which has no name. An application can create additional, named data streams and access them by referring to their names. To avoid altering the Microsoft Windows I/O APIs, which take a string as a filename argument, the name of the data stream is specified by appending a colon (:) to the filename. Because the colon is a reserved character, it can serve as a separator between the filename and the data stream name, as illustrated in this example:

```
myfile.dat:stream2
```

Each stream has a separate allocation size (which defines how much disk space has been reserved for it), actual size (which is how many bytes the caller has used), and valid data length (which is how much of the stream has been initialized). In addition, each stream is given a separate file lock that is used to lock byte ranges and to allow concurrent access.

One component in Windows that uses multiple data streams is the Apple Macintosh file server support that comes with Windows Server. Macintosh systems use two streams per file—one to store data and the other to store resource information, such as the file type and the icon used to represent the file. Because NTFS allows multiple data streams, a Macintosh user can copy an entire Macintosh folder to a Windows server, and another Macintosh user can copy the folder from the server without losing resource information.

Windows Explorer is another application that uses streams. When you right-click on an NTFS file and select Properties, the Summary tab of the resulting dialog box lets you associate information with the file, such as a title, subject, author, and keywords. Windows Explorer stores the information in an alternate stream it adds to the file, named Summary Information.

Other applications can use the multiple data stream feature as well. A backup utility, for example, might use an extra data stream to store backup-specific time stamps on files. Or an archival utility might implement hierarchical storage in which files that are older than a certain date or that haven't been accessed for a specified period of time are moved to tape. The utility could copy the file to tape, set the file's default data stream to 0, and add a data stream that specifies the name and location of the tape on which the file is stored.

EXPERIMENT: Looking at Streams

Most Windows applications aren't designed to work with alternate named streams, but both the *echo* and *more* commands are. Thus, a simple way to view streams in action is to create a named stream using *echo* and then display it using *more*. The following command sequence creates a file named test with a stream named stream:

```
C:\>echo hello > test:stream
C:\>more < test:stream
hello
C:\>
```

If you perform a directory listing, test's file size doesn't reflect the data stored in the alternate stream because NTFS returns the size of only the unnamed data stream for file query operations, including directory listings.

```
C:\>dir test
 Volume in drive C is WINDOWS
 Volume Serial Number is 3991-3040

 Directory of C:\

08/01/00  02:37p                    0 test
               1 File(s)             0 bytes
                            112,558,080 bytes free
```

You can determine what files and directories on your system have alternate data streams with the Streams utility from *www.sysinternals.com*:

```
C:\>streams test

Streams v1.5 - Enumerate alternate NTFS data streams
Copyright (C) 1999-2003 Mark Russinovich
Sysinternals - www.sysinternals.com
 C:\Temp\test:
          :stream:$DATA 8
```

Unicode-Based Names

Like Windows as a whole, NTFS is fully Unicode enabled, using Unicode characters to store names of files, directories, and volumes. Unicode, a 16-bit character-coding scheme, allows each character in each of the world's major languages to be uniquely represented, which aids in moving data easily from one country to another. Unicode is an improvement over the traditional representation of international characters—using a double-byte coding scheme that stores some characters in 8 bits and others in 16 bits, a technique that requires loading various code pages to establish the available characters. Because Unicode has a unique representation for each character, it doesn't depend on which code page is loaded. Each directory and filename in a path can be as many as 255 characters long and can contain Unicode characters, embedded spaces, and multiple periods.

General Indexing Facility

The NTFS architecture is structured to allow indexing of file attributes on a disk volume. This structure enables the file system to efficiently locate files that match certain criteria—for example, all the files in a particular directory. The FAT file system indexes filenames but doesn't sort them, making lookups in large directories slow.

Several NTFS features take advantage of general indexing, including consolidated security descriptors, in which the security descriptors of a volume's files and directories are stored in a single internal stream, have duplicates removed, and are indexed using an internal security identifier that NTFS defines. The use of indexing by these features is described in the section "NTFS On-Disk Structure" later in this chapter.

Dynamic Bad-Cluster Remapping

Ordinarily, if a program tries to read data from a bad disk sector, the read operation fails and the data in the allocated cluster becomes inaccessible. If the disk is formatted as a fault-tolerant NTFS volume, however, the Windows fault-tolerant driver dynamically retrieves a good copy of the data that was stored on the bad sector and then sends NTFS a warning that the sector is bad. NTFS allocates a new cluster, replacing the cluster in which the bad sector resides, and copies the data to the new cluster. It flags the bad cluster and no longer uses it. This data recovery and dynamic bad-cluster remapping is an especially useful feature for file servers and fault-tolerant systems or for any application that can't afford to lose data. If the volume manager isn't loaded when a sector goes bad, NTFS still replaces the cluster and doesn't reuse it, but it can't recover the data that was on the bad sector.

Hard Links and Junctions

A hard link allows multiple paths to refer to the same file. (Hard links are not supported on directories.) If you create a hard link named C:\Users\Documents\Spec.doc that refers to the existing file C:\My Documents\Spec.doc, the two paths link to the same on-disk file and you can make changes to the file using either path. Processes can create hard links with the Windows *CreateHardLink* function or the *ln* POSIX function.

In addition to hard links, NTFS supports another type of redirection called *junctions*. Junctions, also called symbolic links, allow a directory to redirect file or directory pathname translation to an alternate directory. For example, if the path C:\Drivers is a junction that redirects to C:\Windows\System32\Drivers, an application reading C:\Drivers\Ntfs.sys actually reads C:\Windows\System\Drivers\Ntfs.sys. Junctions are a useful way to lift directories that are deep in a directory tree to a more convenient depth without disturbing the original tree's structure or contents. The example just cited lifts the drivers directory to the volume's root directory, reducing the directory depth of Ntfs.sys from three levels to one when Ntfs.sys is accessed through the junction. You can't use junctions to link to remote directories—only to directories on local volumes.

Junctions are based on an NTFS mechanism called *reparse points*. (Reparse points are discussed further in the section "Reparse Points" later in this chapter.) A reparse point is a file or directory that has a block of data called *reparse data* associated with it. Reparse data is user-defined data about the file or directory, such as its state or location, that can be read from the reparse point by the application that created the data, a file system filter driver, or the I/O manager. When NTFS encounters a reparse point during a file or directory lookup, it returns a *reparse status code*, which signals file system filter drivers that are attached to the volume, and the I/O manager, to examine the reparse data. Each reparse point type has a unique *reparse* *tag*. The reparse tag allows the component responsible for interpreting the reparse point's reparse data to recognize the reparse point without having to check the reparse data. A reparse tag owner, either a file system filter driver or the I/O manager, can choose one of the following options when it recognizes reparse data:

- The reparse tag owner can manipulate the pathname specified in the file I/O operation that crosses the reparse point and let the I/O operation reissue with the altered pathname. Junctions take this approach to redirect a directory lookup, for example.
- The reparse tag owner can remove the reparse point from the file, alter the file in some way, and then reissue the file I/O operation. The Windows Hierarchical Storage Management (HSM) system uses reparse points in this way. HSM archives files by moving their contents to tape, leaving reparse points in their place. When a process accesses a file that has been archived, the HSM filter driver (\Windows\System32\Drivers\Rsfilter.sys) removes the reparse point from the file, reads the file's data from the archival media, and reissues the access. Thus, the retrieval of the offline data is transparent to a process accessing an archived file.

There are no Windows functions for creating reparse points. Instead, processes must use the FSCTL_SET_REPARSE_POINT file system control code with the Windows *DeviceIoControl* function. A process can query a reparse point's contents with the FSCTL_GET_REPARSE_POINT file system control code. The FILE_ATTRIBUTE_REPARSE_POINT flag is set in a reparse point's file attributes, so applications can check for reparse points by using the Windows *GetFileAttributes* function.

EXPERIMENT: Creating a Junction

Windows doesn't include any tools for creating junctions, but you can create a junction with either the Junction tool from *www.sysinternals.com* (which includes source code) or the Windows resource kits tool Linkd. The Linkd tool also lets you view the definition of existing junctions, and Junction lets you view information about junctions and other reparse point tags.

Compression and Sparse Files

NTFS supports compression of file data. Because NTFS performs compression and decompression procedures transparently, applications don't have to be modified to take advantage of this feature. Directories can also be compressed, which means that any files subsequently created in the directory are compressed.

Applications compress and decompress files by passing *DeviceIoControl* the FSCTL_SET_COMPRESSION file system control code. They query the compression state of a file or directory with the FSCTL_GET_COMPRESSION file system control code. A file or directory that is compressed has the FILE_ATTRIBUTE_COMPRESSED flag set in its attributes, so applications can also determine a file or directory's compression state with *GetFileAttributes*.

A second type of compression is known as *sparse files*. If a file is marked as sparse, NTFS doesn't allocate space on a volume for portions of the file that an application designates as empty. NTFS returns 0-filled buffers when an application reads from empty areas of a sparse file. This type of compression can be useful for client/server applications that implement circular-buffer logging, in which the server records information to a file and clients asynchronously read the information. Because the information that the server writes isn't needed after a client has read it, there's no need to store the information in the file. By making such a file sparse, the client can specify the portions of the file it reads as empty, freeing up space on the volume. The server can continue to append new information to the file, without fear that the file will grow to consume all available space on the volume.

As for compressed files, NTFS manages sparse files transparently. Applications specify a file's sparseness state by passing the FSCTL_SET_SPARSE file system control code to *DeviceIoControl*. To set a range of a file to empty, applications use the FSCTL_SET_ZERO_DATA code, and they can ask NTFS for a description of what parts of a file are sparse by using the control code FSCTL_QUERY_ALLOCATED_RANGES. One application of sparse files is the NTFS *change journal*, described next.

Change Logging

Many types of applications need to monitor volumes for file and directory changes. For example, an automatic backup program might perform an initial full backup and then incremental backups based on file changes. An obvious way for an application to monitor a volume for changes is for it to scan the volume, recording the state of files and directories, and on a subsequent scan detect differences. This process can adversely affect system performance, however, especially on computers with thousands or tens of thousands of files.

An alternate approach is for an application to register a directory notification by using the *FindFirstChangeNotification* or *ReadDirectoryChangesW* Windows functions. As an input parameter, the application specifies the name of a directory it wants to monitor, and the function returns whenever the contents of the directory changes. Although this approach is more efficient than volume scanning, it requires the application to be running at all times. Using these functions can also require an application to scan directories because *FindFirstChangeNotification* doesn't indicate what changed—just that something in the directory has changed. An application can pass a buffer to *ReadDirectoryChangesW* that the FSD fills in with change records. If the buffer overflows, however, the application must be prepared to fall back on scanning the directory.

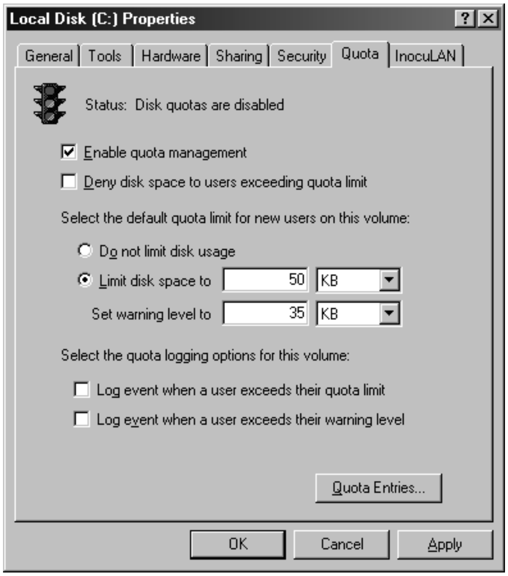
NTFS provides a third approach that overcomes the drawbacks of the first two: an application can configure the NTFS change journal facility by using the *DeviceIoControl* function's FSCTL_CREATE_USN_JOURNAL file system control code to have NTFS record information about file and directory changes to an internal file called the *change journal*. A change journal is usually large enough to virtually guarantee that applications get a chance to process changes without missing any. Applications use the FSCTL_QUERY_USN_JOURNAL file system control to read records from a change journal, and they can specify that the *DeviceIoControl* function not complete until new records are available.

Per-User Volume Quotas

Systems administrators often need to track or limit user disk space usage on shared storage volumes, so NTFS includes quota-management support. NTFS quota-management support allows for per-user specification of quota enforcement, which is useful for usage tracking and tracking when a user reaches warning and limit thresholds. NTFS can be configured to log an event indicating the occurrence to the system Event Log if a user surpasses his warning limit. Similarly, if a user attempts to use more volume storage then her quota limit permits, NTFS can log an event to the system Event Log and fail the application file I/O that would have caused the quota violation with a "disk full" error code.

NTFS tracks a user's volume usage by relying on the fact that it tags files and directories with the security ID (SID) of the user who created them. (See Chapter 8 for a definition of SIDs.) The logical sizes of files and directories a user owns count against the user's administrator-defined quota limit. Thus, a user can't circumvent his or her quota limit by creating an empty sparse file that is larger than the quota would allow and then fill the file with nonzero data. Similarly, whereas a 50-KB file might compress to 10 KB, the full 50 KB is used for quota accounting.

By default, volumes don't have quota tracking enabled. You need to use the Quota tab of a volume's Properties dialog box, shown in Figure 12-18, to enable quotas, to specify default warning and limit thresholds, and to configure the NTFS behavior that occurs when a user hits the warning or limit threshold. The Quota Entries tool, which you can launch from this dialog box, enables an administrator to specify different limits and behavior for each user. Applications that want to interact with NTFS quota management use COM quota interfaces, including *IDiskQuotaControl*, *IDiskQuotaUser*, and *IDiskQuotaEvents*.



Status: Disk quotas are disabled

☑ Enable quota management

☐ Deny disk space to users exceeding quota limit

Select the default quota limit for new users on this volume:

○ Do not limit disk usage

● Limit disk space to 50 KB

Set warning level to 35 KB

Select the quota logging options for this volume:

☐ Log event when a user exceeds their quota limit

☐ Log event when a user exceeds their warning level

Figure 12-18 Volume Properties dialog box

Link Tracking

Shell shortcuts allow users to place files in their shell namespace (on their desktop, for example) that link to files located in the file system namespace. The Windows Start menu uses shell shortcuts extensively. Similarly, object linking and embedding (OLE) links allow documents from one application to be transparently embedded in the documents of other applications. The products of the Microsoft Office suite, including PowerPoint, Excel, and Word, use OLE linking.

Although shell and OLE links provide an easy way to connect files with one another and with the shell namespace, they have in the past been difficult to manage. If a user moves the source of a shell or OLE link (a link source is the file or directory to which a link points) in Windows NT 4, Windows 95, or Windows 98, the link will be broken and the system has to rely on heuristics to attempt to locate the link's source. NTFS in Windows includes support for a service application called distributed link-tracking, which maintains the integrity of shell and OLE links when link targets move. Using the NTFS link-tracking support, if a link source located on an NTFS volume moves to any other NTFS volume within the originating volume's domain, the link-tracking service can transparently follow the movement and update the link to reflect the change.

NTFS link-tracking support is based on an optional file attribute known as an *object ID*. An application can assign an object ID to a file by using the FSCTL_CREATE_OR_GET_OBJECT_ID (which assigns an ID if one isn't already assigned) and FSCTL_SET_OBJECT_ID file system control codes. Object IDs are queried with the FSCTL_CREATE_OR_GET_OBJECT_ID and FSCTL_GET_OBJECT_ID file system control codes. The FSCTL_DELETE_OBJECT_ID file system control code lets applications delete object IDs from files.

Encryption

Corporate users often store sensitive information on their computers. Although data stored on company servers is usually safely protected with proper network security settings and physical access control, data stored on laptops can be exposed when a laptop is lost or stolen. NTFS file permissions don't offer protection because NTFS volumes can be fully accessed without regard to security by using NTFS file-reading software that doesn't require Windows to be running. Furthermore, NTFS file permissions are rendered useless when an alternate Windows installation is used to access files from an administrator account. Recall from Chapter 8 that the administrator account has the take-ownership and backup privileges, both of which allow it to access any secured object by overriding the object's security settings.

NTFS includes a facility called Encrypting File System (EFS), which users can use to encrypt sensitive data. The operation of EFS, as that of file compression, is completely transparent to applications, which means that file data is automatically decrypted when an application running in the account of a user authorized to view the data reads it and is automatically encrypted when an authorized application changes the data.

Note NTFS doesn't permit the encryption of files located in the system volume's root directory or in the \Windows directory because many files in these locations are required during the boot process and EFS isn't active during the boot process.

EFS relies on cryptographic services supplied by Windows in user mode, so it consists of both a kernel-mode component that tightly integrates with NTFS as well as user-mode DLLs that communicate with the Local Security Authority Subsystem (LSASS) and cryptographic DLLs.

Files that are encrypted can be accessed only by using the private key of an account's EFS private/public key pair, and private keys are locked using an account's password. Thus, EFS-encrypted files on lost or stolen laptops can't be accessed using any means (other than a brute-force cryptographic attack) without the password of an account that is authorized to view the data.

Applications can use the *EncryptFile* and *DecryptFile* Windows API functions to encrypt and decrypt files, and *FileEncryptionStatus* to retrieve a file or directory's EFS-related attributes, such as whether the file or directory is encrypted.

POSIX Support

As explained in Chapter 2, one of the mandates for Windows was to fully support the POSIX 1003.1 standard. In the file system area, the POSIX standard requires support for case-sensitive file and directory names, traversal permissions (where security for each directory of a path is used when determining whether a user has access to a file or directory), a "file-change-time" time stamp (which is different than the MS-DOS "time-last-modified" stamp), and hard links (multiple directory entries that point to the same file). NTFS implements each of these features.

Defragmentation

A common myth that many people have held since the introduction of NTFS is that it automatically optimizes file placement on disk so as not to fragment the files. While it does make efforts to keep files contiguous, a volume's files can become fragmented over time, especially when there is limited free space. A file is fragmented if its data occupies discontiguous clusters. For example, Figure 12-19 shows a fragmented file consisting of three fragments. However, like most file systems (including versions of FAT on Windows), NTFS makes no special efforts to keep files contiguous, other than to reserve a region of disk space known as the master file table (MFT) zone for the MFT. (NTFS lets other files allocate from the MFT zone when volume free space runs low.) Keeping an area free for the MFT can help it stay contiguous, but it, too, can become fragmented. (See the section "Master File Table" later in this chapter for more information on MFTs.)

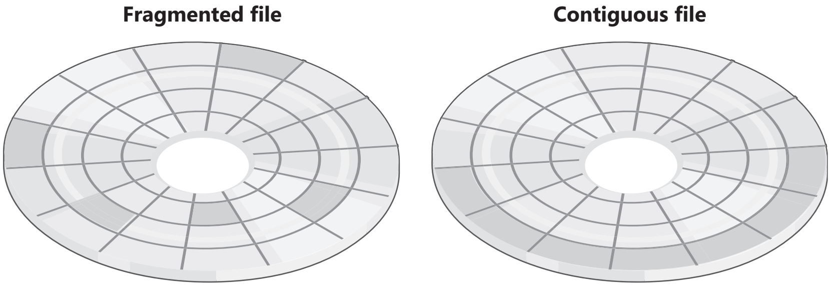


Figure 12-19 Fragmented and contiguous files

To facilitate the development of third-party disk defragmentation tools, Windows includes a defragmentation API that such tools can use to move file data so that files occupy contiguous clusters. The API consists of file system controls that let applications obtain a map of a volume's free and in-use clusters (FSCTL_GET_VOLUME_BITMAP), obtain a map of a file's cluster usage (FSCTL_GET_RETRIEVAL_POINTERS), and move a file (FSCTL_MOVE_ FILE).

Windows includes a built-in defragmentation tool that is accessible by using the Disk Defragmenter utility (\Windows\System32\Dfrg.msc). The built-in defragmentation tool has a number of limitations that on Windows 2000 include an inability to be run from the command prompt or to be automatically scheduled. In Windows XP and Windows Server 2003, the defragmenter has a command-line interface, \Windows\System32\Defrag.exe, that you can run interactively or schedule, but that does not produce detailed reports or offer control—such as excluding files or directories—over the defragmentation process. Third-party disk defragmentation products typically offer a richer feature set.

In Windows XP, the defragmentation support in NTFS was rewritten to alleviate restrictions present in Windows 2000. For example, in Windows 2000 the defragmenting support in NTFS relies on the cache manager, which introduces a number of limitations such as the inability to move pieces of a file that cross a 256-KB boundary within the file or to defragment NTFS metadata files (such as the Master File Table). (Note that 256 KB is the size of a cache manager view. See Chapter 11 for more information on the cache manager.) In Windows XP and Windows Server 2003, the only limitations imposed by the defragmentation implementation in NTFS is that paging files and NTFS log files cannot be defragmented.

Read-Only Support

Prior to Windows XP, the NTFS file system driver required that volumes it mounted be on writeable media so that it could reset its transactional log files. The Windows XP and Windows Server 2003 NTFS drivers can mount volumes on read-only media, functionality required by embedded systems that have read-only base file system images.

NTFS File System Driver

As described in Chapter 9, in the framework of the Windows I/O system, NTFS and other file systems are loadable device drivers that run in kernel mode. They are invoked indirectly by applications that use Windows or other I/O APIs (such as POSIX). As Figure 12-20 shows, the Windows environment subsystems call Windows system services, which in turn locate the appropriate loaded drivers and call them. (For a description of system service dispatching, see the section "System Service Dispatching" in Chapter 3.)

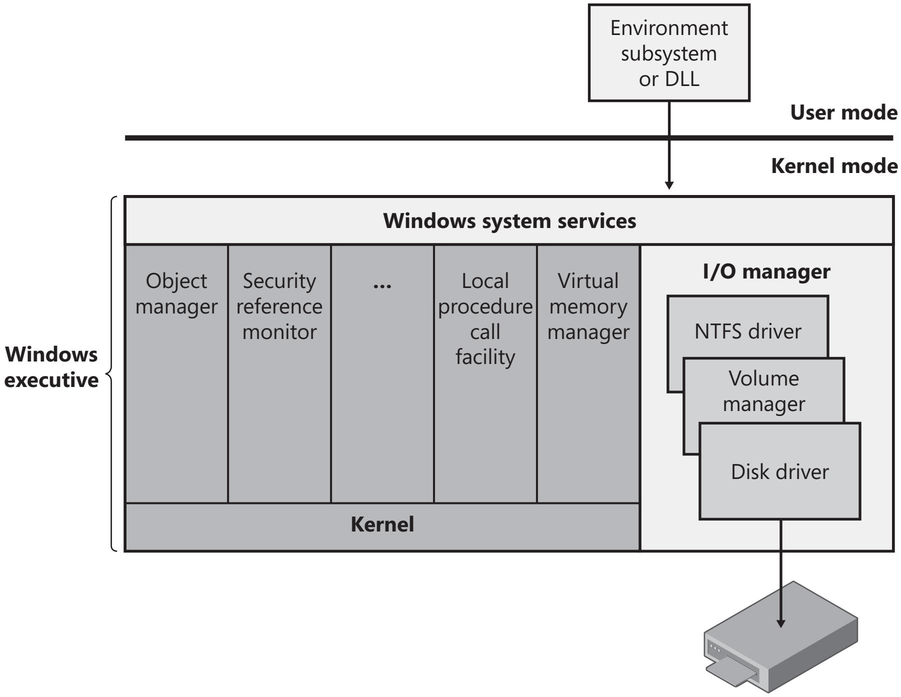


Figure 12-20 Components of the Windows I/O system

The layered drivers pass I/O requests to one another by calling the Windows executive's I/O manager. Relying on the I/O manager as an intermediary allows each driver to maintain independence so that it can be loaded or unloaded without affecting other drivers. In addition, the NTFS driver interacts with the three other Windows executive components, shown in the left side of Figure 12-21, that are closely related to file systems.

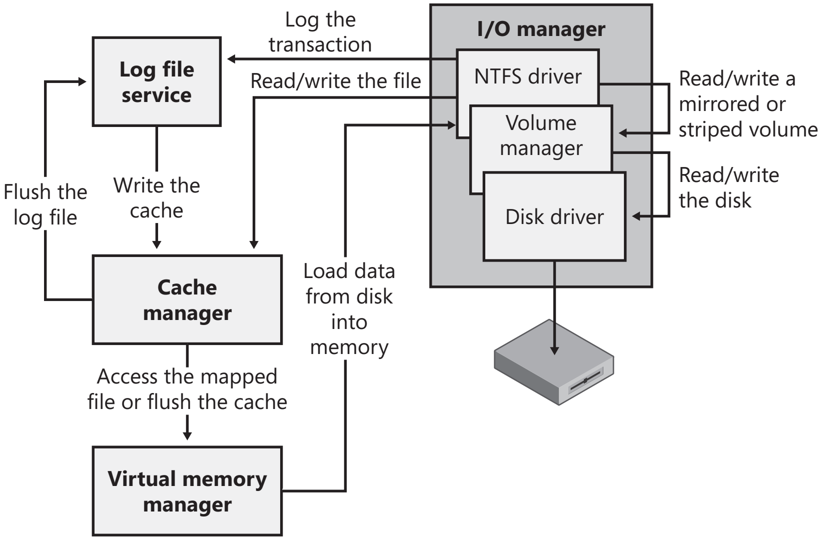


Figure 12-21 NTFS and related components

The log file service (LFS) is the part of NTFS that provides services for maintaining a log of disk writes. The log file LFS writes is used to recover an NTFS-formatted volume in the case of a system failure. (See the section "Log File Service (LFS).")

The cache manager is the component of the Windows executive that provides systemwide caching services for NTFS and other file system drivers, including network file system drivers (servers and redirectors). All file systems implemented for Windows access cached files by mapping them into system address space and then accessing the virtual memory. The cache manager provides a specialized file system interface to the Windows memory manager for this purpose. When a program tries to access a part of a file that isn't loaded into the cache (a *cache miss*), the memory manager calls NTFS to access the disk driver and obtain the file contents from disk. The cache manager optimizes disk I/O by using its lazy writer threads to call the memory manager to flush cache contents to disk as a background activity (asynchronous disk writing). (For a complete description of the cache manager, see Chapter 11.)

NTFS participates in the Windows object model by implementing files as objects. This implementation allows files to be shared and protected by the object manager, the component of Windows that manages all executive-level objects. (The object manager is described in the section "Object Manager" in Chapter 3.)

An application creates and accesses files just as it does other Windows objects: by means of object handles. By the time an I/O request reaches NTFS, the Windows object manager and security system have already verified that the calling process has the authority to access the file object in the way it is attempting to. The security system has compared the caller's access token to the entries in the access-control list for the file object. (See Chapter 8 for more information about access-control lists.) The I/O manager has also transformed the file handle into a pointer to a file object. NTFS uses the information in the file object to access the file on disk.

Figure 12-22 shows the data structures that link a file handle to the file system's on-disk structure.

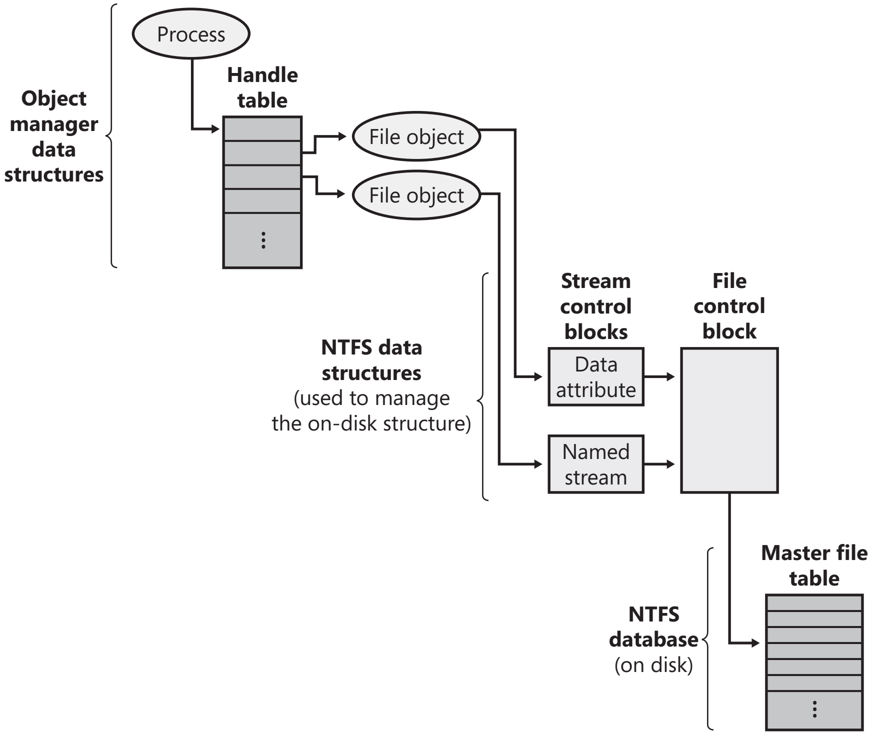


Figure 12-22 NTFS data structures

NTFS follows several pointers to get from the file object to the location of the file on disk. As Figure 12-22 shows, a file object, which represents a single call to the open-file system service, points to a *stream control block* (SCB) for the file attribute that the caller is trying to read or write. In Figure 12-22, a process has opened both the unnamed data attribute and a named stream (alternate data attribute) for the file. The SCBs represent individual file attributes and contain information about how to find specific attributes within a file. All the SCBs for a file point to a common data structure called a *file control block* (FCB). The FCB contains a pointer (actually, a file reference, explained in the section "File Reference Numbers" later in this chapter) to the file's record in the disk-based master file table (MFT), which is described in detail in the following section.

NTFS On-Disk Structure

This section describes the on-disk structure of an NTFS volume, including how disk space is divided and organized into clusters, how files are organized into directories, how the actual file data and attribute information is stored on disk, and finally, how NTFS data compression works.

Volumes

The structure of NTFS begins with a volume. A *volume* corresponds to a logical partition on a disk, and it is created when you format a disk or part of a disk for NTFS. You can also create a RAID volume that spans multiple disks by using the Windows Disk Management MMC snap-in.

A disk can have one volume or several. NTFS handles each volume independently of the others. Three sample disk configurations for a 150-MB hard disk are illustrated in Figure 12-23.

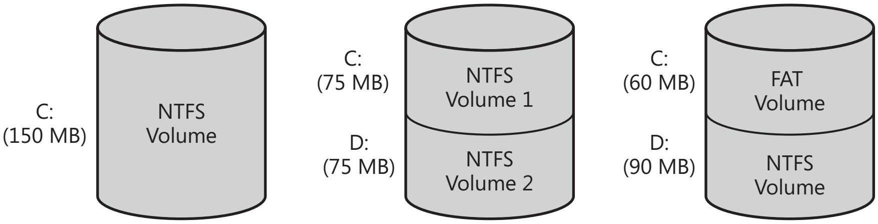


Figure 12-23 Sample disk configurations

A volume consists of a series of files plus any additional unallocated space remaining on the disk partition. In the FAT file system, a volume also contains areas specially formatted for use by the file system. An NTFS volume, however, stores all file system data, such as bitmaps and directories, and even the system bootstrap, as ordinary files.

> **Note** The on-disk format of NTFS volumes on Windows 2000 is version 3.0, and because of minor changes to the format, it is version 3.1 on Windows XP and Windows Server 2003. The version number of a volume is stored in its $Volume metadata file.

Clusters

The cluster size on an NTFS volume, or the *cluster factor*, is established when a user formats the volume with either the *format* command or the Disk Management MMC snap-in. The default cluster factor varies with the size of the volume, but it is an integral number of physical sectors, always a power of 2 (1 sector, 2 sectors, 4 sectors, 8 sectors, and so on). The cluster factor is expressed as the number of bytes in the cluster, such as 512 bytes, 1 KB, or 2 KB.

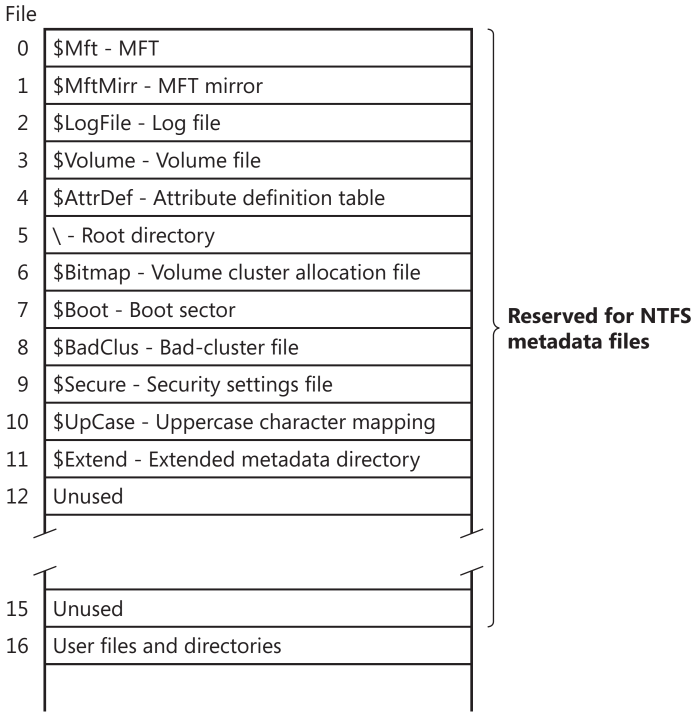
Internally, NTFS refers only to clusters. (However, NTFS forms low-level volume I/O operations such that it is sector-aligned and its length is a multiple of the sector size.) NTFS uses the cluster as its unit of allocation to maintain its independence from physical sector sizes. This independence allows NTFS to efficiently support very large disks by using a larger cluster factor or to support nonstandard disks that have a sector size other than 512 bytes. On a larger volume, use of a larger cluster factor can reduce fragmentation and speed allocation, at a small cost in terms of wasted disk space. Both the *format* command available from the Windows Command Prompt and the Format menu option under the All Tasks option on the Action menu in the Disk Management MMC snap-in choose a default cluster factor based on the volume size, but you can override this size.

NTFS refers to physical locations on a disk by means of *logical cluster numbers* (LCNs). LCNs are simply the numbering of all clusters from the beginning of the volume to the end. To convert an LCN to a physical disk address, NTFS multiplies the LCN by the cluster factor to get the physical byte offset on the volume, as the disk driver interface requires. NTFS refers to the data within a file by means of *virtual cluster numbers* (VCNs). VCNs number the clusters belonging to a particular file from 0 through *m*. VCNs aren't necessarily physically contiguous, however; they can be mapped to any number of LCNs on the volume.

Master File Table

In NTFS, all data stored on a volume is contained in files, including the data structures used to locate and retrieve files, the bootstrap data, and the bitmap that records the allocation state of the entire volume (the NTFS metadata). Storing everything in files allows the file system to easily locate and maintain the data, and each separate file can be protected by a security descriptor. In addition, if a particular part of the disk goes bad, NTFS can relocate the metadata files to prevent the disk from becoming inaccessible.

The MFT is the heart of the NTFS volume structure. The MFT is implemented as an array of file records. The size of each file record is fixed at 1 KB, regardless of cluster size. (The structure of a file record is described in the "File Records" section later in this chapter.) Logically, the MFT contains one record for each file on the volume, including a record for the MFT itself. In addition to the MFT, each NTFS volume includes a set of metadata files containing the information that's used to implement the file system structure. Each of these NTFS metadata files has a name that begins with a dollar sign ($), although the signs are hidden. For example, the filename of the MFT is $Mft. The rest of the files on an NTFS volume are normal user files and directories, as shown in Figure 12-24.



File	
0	$Mft - MFT
1	$MftMirr - MFT mirror
2	$LogFile - Log file
3	$Volume - Volume file
4	$AttrDef - Attribute definition table
5	\ - Root directory
6	$Bitmap - Volume cluster allocation file
7	$Boot - Boot sector
8	$BadClus - Bad-cluster file
9	$Secure - Security settings file
10	$UpCase - Uppercase character mapping
11	$Extend - Extended metadata directory
12	Unused
15	Unused
16	User files and directories

Figure 12-24 File records for NTFS metadata files in the MFT

Usually, each MFT record corresponds to a different file. If a file has a large number of attributes or becomes highly fragmented, however, more than one record might be needed for a single file. In such cases, the MFT first record, which stores the locations of the others, is called the *base file record*.

EXPERIMENT: Viewing the MFT

The Nfi utility included in the OEM Support Tools (part of the Windows debugging tools and available for download at *support.microsoft.com/support/kb/articles/Q253/0/66.asp*) allows you to dump the contents of an NTFS volume's MFT as well as to translate a volume cluster number or physical-disk sector number (on non-RAID volumes only) to the file that contains it, if it's part of a file. The first 16 entries of the MFT are reserved for metadata files, but optional metadata files (which are present only if a volume uses an associated feature) fall outside this area: \$Extend\$Quota, \$Extend\$ObjId, \$Extend\$UsnJrnl, and \$Extend\$Reparse. The following dump was performed on a volume that uses reparse points ($Reparse), quotas ($Quota), and object IDs ($ObjId):

```
C:\>nfi G:\
NTFS File Sector Information Utility.
Copyright (C) Microsoft Corporation 1999. All rights reserved.
```

```
File 0
Master File Table ($Mft)
    $STANDARD_INFORMATION (resident)
    $FILE_NAME (resident)
    $DATA (nonresident)
        logical sectors 32-52447 (0x20-0xccdf)
    $BITMAP (nonresident)
        logical sectors 16-23 (0x10-0x17)

File 1
Master File Table Mirror ($MftMirr)
    $STANDARD_INFORMATION (resident)
    $FILE_NAME (resident)
    $DATA (nonresident)
        logical sectors 2048728-2048735 (0x1f42d8-0x1f42df)

File 2
Log File ($LogFile)
    $STANDARD_INFORMATION (resident)
    $FILE_NAME (resident)
    $DATA (nonresident)
        logical sectors 2048736-2073343 (0x1f42e0-0x1fa2ff)
File 3
DASD ($Volume)
    $STANDARD_INFORMATION (resident)
    $FILE_NAME (resident)
    $OBJECT_ID (resident)
    $SECURITY_DESCRIPTOR (resident)
    $VOLUME_NAME (resident)
    $VOLUME_INFORMATION (resident)
    $DATA (resident)

File 4
Attribute Definition Table ($AttrDef)
    $STANDARD_INFORMATION (resident)
    $FILE_NAME (resident)
    $SECURITY_DESCRIPTOR (resident)
    $DATA (nonresident)
        logical sectors 512256-512263 (0x7d100-0x7d107)

File 5
Root Directory
    $STANDARD_INFORMATION (resident)
    $FILE_NAME (resident)
    $SECURITY_DESCRIPTOR (resident)
    $INDEX_ROOT $I30 (resident)
    $INDEX_ALLOCATION $I30 (nonresident)
        logical sectors 2073416-2073423 (0x1fa348-0x1fa34f)
    $BITMAP $I30 (resident)

File 6
Volume Bitmap ($BitMap)
    $STANDARD_INFORMATION (resident)
    $FILE_NAME (resident)
    $DATA (nonresident)
        logical sectors 2073424-2073675 (0x1fa350-0x1fa44b)

File 7
Boot Sectors ($Boot)
    $STANDARD_INFORMATION (resident)
    $FILE_NAME (resident)
    $SECURITY_DESCRIPTOR (resident)
    $DATA (nonresident)
        logical sectors 0-15 (0x0-0xf)
File 8
Bad Cluster List ($BadClus)
    $STANDARD_INFORMATION (resident)
    $FILE_NAME (resident)
    $DATA (resident)
    $DATA $Bad (nonresident)

File 9
Security ($Secure)
    $STANDARD_INFORMATION (resident)
    $FILE_NAME (resident)
    $DATA $SDS (nonresident)
        logical sectors 2073932-2074447 (0x1fa54c-0x1fa74f)
        logical sectors 523160-523163 (0x7fb98-0x7fb9b)
    $INDEX_ROOT $SDH (resident)
    $INDEX_ROOT $SII (resident)
    $INDEX_ALLOCATION $SDH (nonresident)
        logical sectors 1876152-1876159 (0x1ca0b8-0x1ca0bf)
    $INDEX_ALLOCATION $SII (nonresident)
        logical sectors 24-31 (0x18-0x1f)
    $BITMAP $SDH (resident)
    $BITMAP $SII (resident)

File 10
Upcase Table ($UpCase)
    $STANDARD_INFORMATION (resident)
    $FILE_NAME (resident)
    $DATA (nonresident)
        logical sectors 2073676-2073931 (0x1fa44c-0x1fa54b)

File 11
Extend Table ($Extend)
    $STANDARD_INFORMATION (resident)
    $FILE_NAME (resident)
    $INDEX_ROOT $I30 (resident)

File 12
(unknown/unnamed)
    $STANDARD_INFORMATION (resident)
    $SECURITY_DESCRIPTOR (resident)
    $DATA (resident)
File 13
(unknown/unnamed)
    $STANDARD_INFORMATION (resident)
    $SECURITY_DESCRIPTOR (resident)
    $DATA (resident)

File 14
(unknown/unnamed)
    $STANDARD_INFORMATION (resident)
    $SECURITY_DESCRIPTOR (resident)
    $DATA (resident)

File 15
(unknown/unnamed)
    $STANDARD_INFORMATION (resident)
    $SECURITY_DESCRIPTOR (resident)
    $DATA (resident)

File 24
\$Extend\$Quota
    $STANDARD_INFORMATION (resident)
    $FILE_NAME (resident)
    $INDEX_ROOT $O (resident)
    $INDEX_ROOT $Q (resident)

File 25
\$Extend\$ObjId
    $STANDARD_INFORMATION (resident)
    $FILE_NAME (resident)
    $INDEX_ROOT $O (resident)

File 26 \$Extend\$Reparse
    $STANDARD_INFORMATION (resident)
    $FILE_NAME (resident)
    $INDEX_ROOT $R (resident)
```

When it first accesses a volume, NTFS must *mount* it—that is, read metadata from the disk and construct internal data structures so that it can process application file system accesses. To mount the volume, NTFS looks in the boot sector to find the physical disk address of the MFT. The MFT's own file record is the first entry in the table; the second file record points to a file located in the middle of the disk called the *MFT mirror* (filename $MftMirr) that contains a copy of the first few rows of the MFT. This partial copy of the MFT is used to locate metadata files if part of the MFT file can't be read for some reason.

Once NTFS finds the file record for the MFT, it obtains the VCN-to-LCN mapping information in the file record's data attribute and stores it in memory. Each run has a VCN-to-LCN mapping and a run length because that's all the information necessary to locate an LCN for any VCN. This mapping information tells NTFS where the runs composing the MFT are located on the disk. (Runs are explained later in this chapter in the section "Resident and Nonresident Attributes.") NTFS then processes the MFT records for several more metadata files and opens the files. Next, NTFS performs its file system recovery operation (described in the section "Recovery"), and finally, it opens its remaining metadata files. The volume is now ready for user access.

Note For the sake of clarity, the text and diagrams in this chapter depict a run as including a VCN, LCN, and run length. NTFS actually compresses this information on disk into an LCN/next-VCN pair. Given a starting VCN, NTFS can determine the length of a run by subtracting the starting VCN from the next VCN.

As the system runs, NTFS writes to another important metadata file, the *log file* (filename $LogFile). NTFS uses the log file to record all operations that affect the NTFS volume structure, including file creation or any commands, such as *Copy*, that alter the directory structure. The log file is used to recover an NTFS volume after a system failure, and is also described in the "Recovery" section later in this chapter.

Another entry in the MFT is reserved for the *root directory* (also known as "\"). Its file record contains an index of the files and directories stored in the root of the NTFS directory structure. When NTFS is first asked to open a file, it begins its search for the file in the root directory's file record. After opening a file, NTFS stores the file's MFT file reference so that it can directly access the file's MFT record when it reads and writes the file later.

NTFS records the allocation state of the volume in the *bitmap file* (filename $Bitmap). The data attribute for the bitmap file contains a bitmap, each of whose bits represents a cluster on the volume, identifying whether the cluster is free or has been allocated to a file.

The *security file* (filename $Secure) stores the volume-wide security descriptor database. NTFS files and directories have individually settable security descriptors, but to conserve space, NTFS stores the settings in a common file, which allows files and directories that have the same security settings to reference the same security descriptor. In most environments, entire directory trees have the same security settings, so this optimization provides a significant savings.

Another system file, the *boot file* (filename $Boot), stores the Windows bootstrap code. For the system to boot, the bootstrap code must be located at a specific disk address. During formatting, however, the *format* command defines this area as a file by creating a file record for it. Creating the boot file allows NTFS to adhere to its rule of making everything on the disk a file. The boot file as well as NTFS metadata files can be individually protected by means of the security descriptors that are applied to all Windows objects. Using this "everything on the disk is a file" model also means that the bootstrap can be modified by normal file I/O, although the boot file is protected from editing.

NTFS also maintains a *bad-cluster file* (filename $BadClus) for recording any bad spots on the disk volume and a file known as the *volume file* (filename $Volume), which contains the volume name, the version of NTFS for which the volume is formatted, and a bit that when set signifies that a disk corruption has occurred and must be repaired by the Chkdsk utility. (The Chkdsk utility is covered in more detail later in the chapter.) The *uppercase file* (filename $UpCase) includes a translation table between lowercase and uppercase characters. NTFS maintains a file containing an *attribute definition table* (filename $AttrDef) that defines the attribute types supported on the volume and indicates whether they can be indexed, recovered during a system recovery operation, and so on.

NTFS stores several metadata files in the *extensions* (directory name $Extend) metadata directory, including the *object identifier file* (filename $ObjId), the *quota file* (filename $Quota), the *change journal file* (filename $UsnJrnl), and the *reparse point file* (filename $Reparse). These files store information related to optional features of NTFS. The object identifier file stores file object IDs, the quota file stores quota limit and behavior information on volumes that have quotas enabled, the change journal file records file and directory changes, and the reparse point file stores information about which files and directories on the volume include reparse point data.

EXPERIMENT: Viewing NTFS Information

In Windows 2000, you can use the NTFSInfo tool from *www.sysinternals.com* to view information about an NTFS volume, including the placement and size of the MFT and MFT zone; and in Windows XP and Windows Server 2003, you can use the built-in Fsutil.exe command-line program:

```
C:\Windows\System32>fsutil fsinfo ntfsinfo c:
NTFS Volume Serial Number :       0xe82828e72828b68a
Version :                         3.1
Number Sectors :                  0x0000000001e461b7
Total Clusters :                  0x00000000003c8c36
Free Clusters  :                  0x00000000000164c8
Total Reserved :                  0x00000000000001b0
Bytes Per Sector  :               512
Bytes Per Cluster :               4096
Bytes Per FileRecord Segment    : 1024
Clusters Per FileRecord Segment : 0
Mft Valid Data Length :           0x0000000006413800
Mft Start Lcn  :                  0x00000000000c5294
Mft2 Start Lcn :                  0x000000000002f427
Mft Zone Start :                  0x00000000003bf7e0
Mft Zone End   :                  0x00000000003bf800
```

File Reference Numbers

A file on an NTFS volume is identified by a 64-bit value called a *file reference*. The file reference consists of a file number and a sequence number. The file number corresponds to the position of the file's file record in the MFT minus 1 (or to the position of the base file record minus 1 if the file has more than one file record). The file reference sequence number, which is incremented each time an MFT file record position is reused, enables NTFS to perform internal consistency checks. A file reference is illustrated in Figure 12-25.

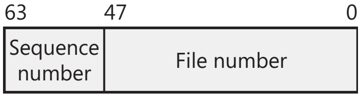


Figure 12-25 File reference

File Records

Instead of viewing a file as just a repository for textual or binary data, NTFS stores files as a collection of attribute/value pairs, one of which is the data it contains (called the *unnamed data attribute*). Other attributes that comprise a file include the filename, time stamp information, and possibly additional named data attributes. Figure 12-26 illustrates an MFT record for a small file.

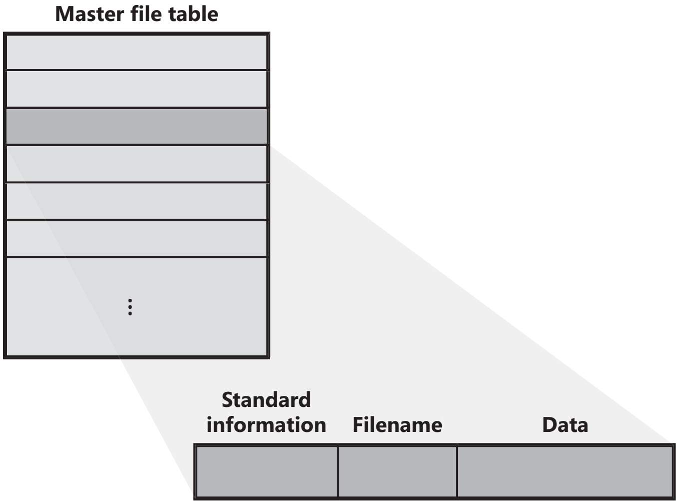


Figure 12-26 MFT record for a small file

Each file attribute is stored as a separate stream of bytes within a file. Strictly speaking, NTFS doesn't read and write files–it reads and writes attribute streams. NTFS supplies these attribute operations: create, delete, read (byte range), and write (byte range). The read and write services normally operate on the file's unnamed data attribute. However, a caller can specify a different data attribute by using the named data stream syntax.

Table 12-4 lists the attributes for files on an NTFS volume. (Not all attributes are present for every file.)

Table 12-4 Attributes for NTFS Files

Attribute	Attribute Name	Description
Volume information	$VOLUME_INFORMATION, $VOLUME_NAME	These attributes are present only in the $Volume metadata file. They store volume version and label information.
Standard information	$STANDARD_INFORMATION	File attributes such as read-only, archive, and so on; time stamps, including when the file was created or last modified; and how many directories point to the file (its *hard link count*).

Table 12-4 **Attributes for NTFS Files**

Attribute	Attribute Name	Description
Filename	$FILE_NAME	The file's name in Unicode characters. A file can have multiple filename attributes, as it does when a hard link to a file exists or when a file with a long name has an automatically generated "short name" for access by MS-DOS and 16-bit Microsoft Windows applications.
Security descriptor	$SECURITY_DESCRIPTOR	This attribute is present for backward compatibility with previous versions of NTFS. The Windows version of NTFS stores all security descriptors in the $Secure metadata file, sharing descriptors among files and directories that have the same settings. Previous versions of NTFS stored private security descriptor information with each file and directory.
Data	$DATA	The contents of the file. In NTFS, a file has one default unnamed data attribute and can have additional named data attributes—that is, a file can have multiple data streams. A directory has no default data attribute but can have optional named data attributes.
Index root, index allocation, and index bitmap	$INDEX_ROOT, $INDEX_ALLOCATION, $BITMAP	Three attributes used to implement filename allocation and bitmap indexes for large directories (directories only).
Attribute list	$ATTRIBUTE_LIST	A list of the attributes that make up the file and the file reference of the MFT file record in which each attribute is located. This seldom-used attribute is present when a file requires more than one MFT file record.
Object ID	$OBJECT_ID	A 64-byte identifier for a file or directory, with the lowest 16 bytes (128 bits) unique to the volume. The link-tracking service assigns object IDs to shell shortcut and OLE link source files. NTFS provides APIs so that files and directories can be opened with their object ID rather than their filename.
Reparse information	$REPARSE_POINT	This attribute stores a file's reparse point data. NTFS junctions and mount points include this attribute.
Extended attributes	$EA, $EA_INFORMATION	Extended attributes aren't actively used but are provided for backward compatibility with OS/2 applications.
EFS information	$EFS	EFS stores data in this attribute that's used to manage a file's encryption, such as the encrypted version of the key needed to decrypt the file and a list of users that are authorized to access the file.

Table 12-4 shows attribute names; however, attributes actually correspond to numeric type codes, which NTFS uses to order the attributes within a file record. The file attributes in an MFT record are ordered by these type codes (numerically in ascending order), with some attribute types appearing more than once—if a file has multiple data attributes, for example, or multiple filenames.

Each attribute in a file record is identified with its attribute type code and has a value and an optional name. An attribute's value is the byte stream composing the attribute. For example, the value of the $FILE_NAME attribute is the file's name; the value of the $DATA attribute is whatever bytes the user stored in the file.

Most attributes never have names, though the index-related attributes and the $DATA attribute often do. Names distinguish among multiple attributes of the same type that a file can include. For example, a file that has a named data stream has two $DATA attributes: an unnamed $DATA attribute storing the default unnamed data stream and a named $DATA attribute having the name of the alternate stream and storing the named stream's data.

Filenames

Both NTFS and FAT allow each filename in a path to be as many as 255 characters long. Filenames can contain Unicode characters as well as multiple periods and embedded spaces. However, the FAT file system supplied with MS-DOS is limited to 8 (non-Unicode) characters for its filenames, followed by a period and a 3-character extension. Figure 12-27 provides a visual representation of the different file namespaces Windows supports and shows how they intersect.

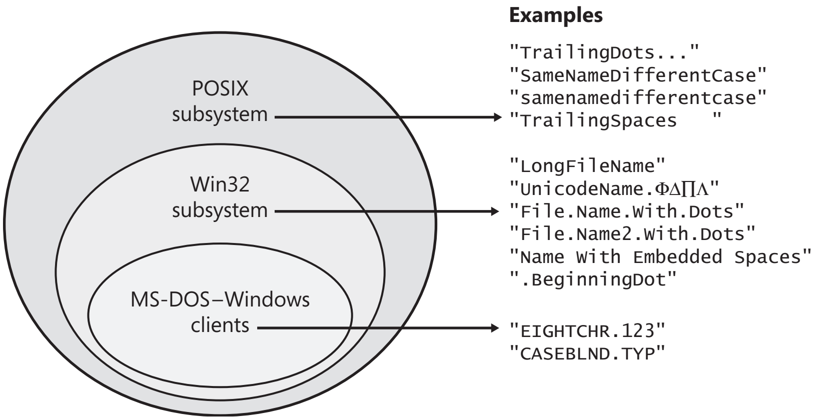


Figure 12-27 Windows file namespaces

The POSIX subsystem requires the biggest namespace of all the application execution environments that Windows supports, and therefore the NTFS namespace is equivalent to the POSIX namespace. The POSIX subsystem can create names that aren't visible to Windows and MS-DOS applications, including names with trailing periods and trailing spaces. Ordinarily, creating a file using the large POSIX namespace isn't a problem because you would do that only if you intended the POSIX subsystem or POSIX client systems to use that file.

The relationship between 32-bit Windows (Windows) applications and MS-DOS Windows applications is a much closer one, however. The Windows area in Figure 12-26 represents filenames that the Windows subsystem can create on an NTFS volume but that MS-DOS and 16-bit Windows applications can't see. This group includes filenames longer than the 8.3 format of MS-DOS names, those containing Unicode (international) characters, those with multiple period characters or a beginning period, and those with embedded spaces. When a file is created with such a name, NTFS automatically generates an alternate, MS-DOS-style filename for the file. Windows displays these short names when you use the */x* option with the *dir* command.

The MS-DOS filenames are fully functional aliases for the NTFS files and are stored in the same directory as the long filenames. The MFT record for a file with an autogenerated MS-DOS filename is shown in Figure 12-28.

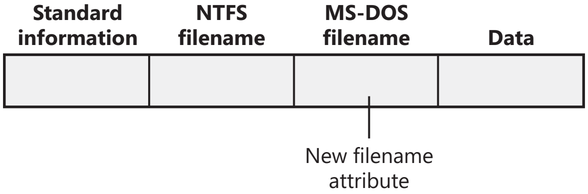


Figure 12-28 MFT file record with an MS-DOS filename attribute

The NTFS name and the generated MS-DOS name are stored in the same file record and therefore refer to the same file. The MS-DOS name can be used to open, read from, write to, or copy the file. If a user renames the file using either the long filename or the short filename, the new name replaces both the existing names. If the new name isn't a valid MS-DOS name, NTFS generates another MS-DOS name for the file.

> **Note** POSIX hard links are implemented in a similar way. When a hard link to a POSIX file is created, NTFS adds another filename attribute to the file's MFT file record. The two situations differ in one regard, however. When a user deletes a POSIX file that has multiple names (hard links), the file record and the file remain in place. The file and its record are deleted only when the last filename (hard link) is deleted. If a file has both an NTFS name and an autogenerated MS-DOS name, however, a user can delete the file using either name.

Here's the algorithm NTFS uses to generate an MS-DOS name from a long filename:

1. Remove from the long name any characters that are illegal in MS-DOS names, including spaces and Unicode characters. Remove preceding and trailing periods. Remove all other embedded periods, except the last one.
2. Truncate the string before the period (if present) to six characters, and append the string "~*n*" (where *n* is a number, starting with 1, that is used to distinguish different files that truncate to the same name). Truncate the string after the period (if present) to three characters.

3. Put the result in uppercase letters. MS-DOS is case-insensitive, and this step guarantees that NTFS won't generate a new name that differs from the old only in case.
4. If the generated name duplicates an existing name in the directory, increment the ~*n* string.

Table 12-5 shows the long Windows filenames from Figure 12-26 and their NTFS-generated MS-DOS versions. The current algorithm and the examples in Figure 12-26 should give you an idea of what NTFS-generated MS-DOS–style filenames look like.

> **Note** Although not generally recommended because it can cause incompatibilities with applications that rely on them, you can disable short name generation by setting HKLM\System\CurrentControlSet\Control\FileSystem\NtfsDisable8dot3NameCreation in the registry to a DWORD value of 1.

Table 12-5 NTFS-Generated Filenames

Windows Long Name	NTFS-Generated Short Name
LongFileName	LONGFI~1
UnicodeName.☞👎🏳☹	UNICOD~1
File.Name.With.Dots	FILENA~1.DOT
File.Name2.With.Dots	FILENA~2.DOT
Name With Embedded Spaces	NAMEWI~1
.BeginningDot	BEGINN~1

Resident and Nonresident Attributes

If a file is small, all its attributes and their values (its data, for example) fit in the file record. When the value of an attribute is stored directly in the MFT, the attribute is called a *resident attribute*. (In Figure 12-27, for example, all attributes are resident.) Several attributes are defined as always being resident so that NTFS can locate nonresident attributes. The standard information and index root attributes are always resident, for example.

Each attribute begins with a standard header containing information about the attribute, information that NTFS uses to manage the attributes in a generic way. The header, which is always resident, records whether the attribute's value is resident or nonresident. For resident attributes, the header also contains the offset from the header to the attribute's value and the length of the attribute's value, as Figure 12-29 illustrates for the filename attribute.

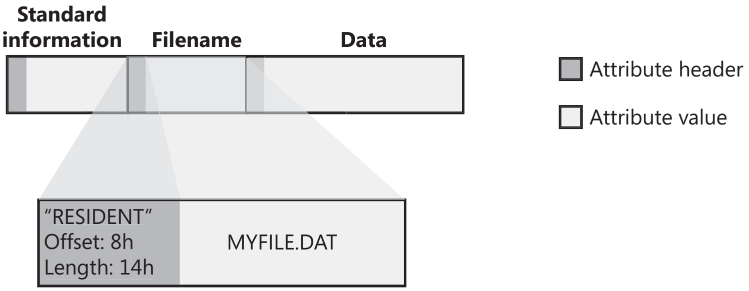


Figure 12-29 Resident attribute header and value

When an attribute's value is stored directly in the MFT, the time it takes NTFS to access the value is greatly reduced. Instead of looking up a file in a table and then reading a succession of allocation units to find the file's data (as the FAT file system does, for example), NTFS accesses the disk once and retrieves the data immediately.

The attributes for a small directory, as well as for a small file, can be resident in the MFT, as Figure 12-30 shows. For a small directory, the index root attribute contains an index of file references for the files and the subdirectories in the directory.

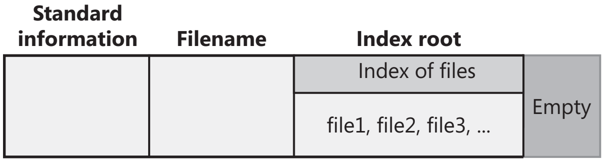


Figure 12-30 MFT file record for a small directory

Of course, many files and directories can't be squeezed into a 1-KB fixed-size MFT record. If a particular attribute, such as a file's data attribute, is too large to be contained in an MFT file record, NTFS allocates clusters for the attribute's data separate from the MFT. This area is called a *run* (or an *extent*). If the attribute's value later grows (if a user appends data to the file, for example), NTFS allocates another run for the additional data. Attributes whose values are stored in runs rather than in the MFT are called *nonresident attributes*. The file system decides whether a particular attribute is resident or nonresident; the location of the data is transparent to the process accessing it.

When an attribute is nonresident, as the data attribute for a large file might be, its header contains the information NTFS needs to locate the attribute's value on the disk. Figure 12-31 shows a nonresident data attribute stored in two runs.

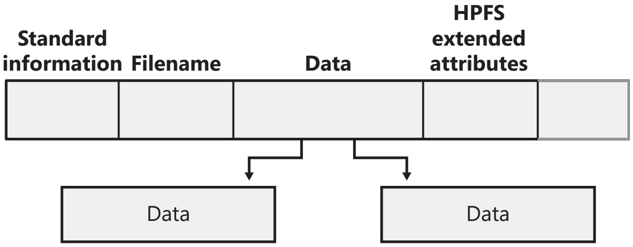


Figure 12-31 MFT file record for a large file with two data runs

Among the standard attributes, only those that can grow can be nonresident. For files, the attributes that can grow are the data and the attribute list (not shown in Figure 12-31). The standard information and filename attributes are always resident.

A large directory can also have nonresident attributes (or parts of attributes), as Figure 12-32 shows. In this example, the MFT file record doesn't have enough room to store the index of files that make up this large directory. A part of the index is stored in the index root attribute, and the rest of the index is stored in nonresident runs called *index buffers*. The index root, index allocation, and bitmap attributes are shown here in a simplified form. They are described in more detail in the next section. The standard information and filename attributes are always resident. The header and at least part of the value of the index root attribute are also resident for directories.

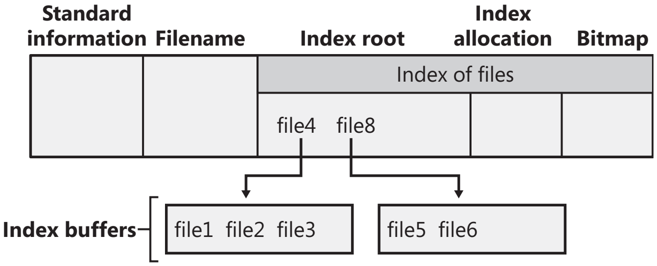


Figure 12-32 MFT file record for a large directory with a nonresident filename index

When a file's (or a directory's) attributes can't fit in an MFT file record and separate allocations are needed, NTFS keeps track of the runs by means of VCN-to-LCN mapping pairs. LCNs represent the sequence of clusters on an entire volume from 0 through *n*. VCNs number the clusters belonging to a particular file from 0 through *m*. For example, the clusters in the runs of a nonresident data attribute are numbered as shown in Figure 12-33.

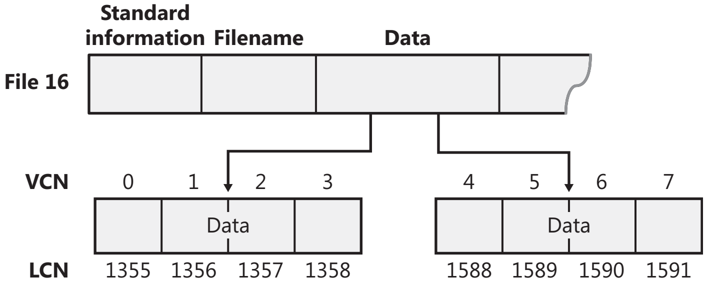


Figure 12-33 VCNs for a nonresident data attribute

If this file had more than two runs, the numbering of the third run would start with VCN 8. As Figure 12-34 shows, the data attribute header contains VCN-to-LCN mappings for the two runs here, which allows NTFS to easily find the allocations on the disk.

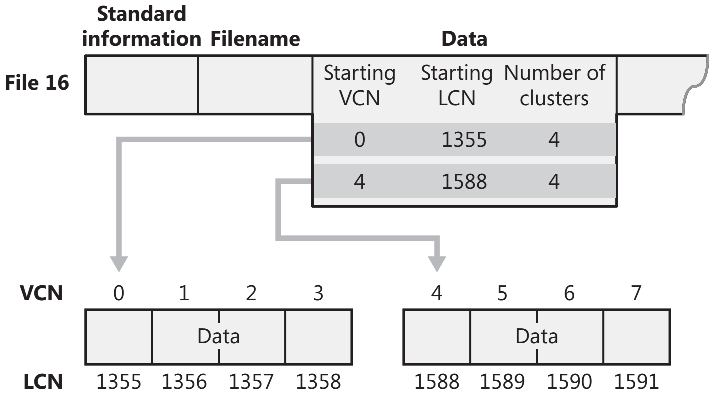


Figure 12-34 VCN-to-LCN mappings for a nonresident data attribute

Although Figure 12-33 shows just data runs, other attributes can be stored in runs if there isn't enough room in the MFT file record to contain them. And if a particular file has too many attributes to fit in the MFT record, a second MFT record is used to contain the additional attributes (or attribute headers for nonresident attributes). In this case, an attribute called the *attribute list* is added. The attribute list attribute contains the name and type code of each of the file's attributes and the file reference of the MFT record where the attribute is located. The attribute list attribute is provided for those cases in which a file grows so large or so fragmented that a single MFT record can't contain the multitude of VCN-to-LCN mappings needed to find all its runs. Files with more than 200 runs typically require an attribute list.

Data Compression and Sparse Files

NTFS supports compression on a per-file, per-directory, or per-volume basis. (NTFS compression is performed only on user data, not file system metadata.) You can tell whether a volume is compressed by using the Windows *GetVolumeInformation* function. To retrieve the actual compressed size of a file, use the Windows *GetCompressedFileSize* function. Finally, to examine or change the compression setting for a file or directory, use the Windows *DeviceIoControl* function. (See the FSCTL_GET_COMPRESSION and FSCTL_SET_COMPRESSION file system control codes.) Keep in mind that although setting a file's compression state compresses (or decompresses) the file right away, setting a directory's or volume's compression state doesn't cause any immediate compression or decompression. Instead, setting a directory's or volume's compression state sets a default compression state that will be given to all newly created files and subdirectories within that directory or volume.

The following section introduces NTFS compression by examining the simple case of compressing sparse data. The subsequent sections extend the discussion to the compression of ordinary files and sparse files.

Compressing Sparse Data

Sparse data is often large but contains only a small amount of nonzero data relative to its size. A sparse matrix is one example of sparse data. As described earlier, NTFS uses VCNs, from 0 through *m*, to enumerate the clusters of a file. Each VCN maps to a corresponding LCN, which identifies the disk location of the cluster. Figure 12-35 illustrates the runs (disk allocations) of a normal, noncompressed file, including its VCNs and the LCNs they map to.

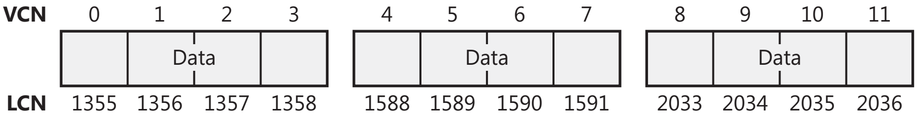
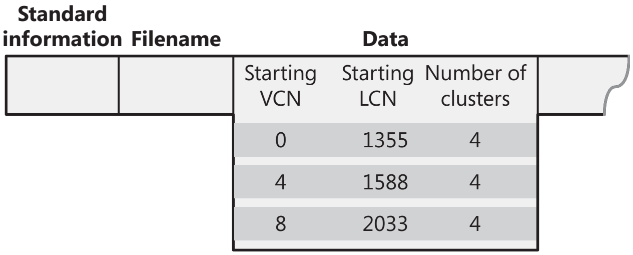


Figure 12-35 Runs of a noncompressed file

This file is stored in 3 runs, each of which is 4 clusters long, for a total of 12 clusters. Figure 12-36 shows the MFT record for this file. As described earlier, to save space, the MFT record's data attribute, which contains VCN-to-LCN mappings, records only one mapping for each run, rather than one for each cluster. Notice, however, that each VCN from 0 through 11 has a corresponding LCN associated with it. The first entry starts at VCN 0 and covers 4 clusters, the second entry starts at VCN 4 and covers 4 clusters, and so on. This entry format is typical for a noncompressed file.



Starting VCN	Starting LCN	Number of clusters
0	1355	4
4	1588	4
8	2033	4

Figure 12-36 MFT record for a noncompressed file

When a user selects a file on an NTFS volume for compression, one NTFS compression technique is to remove long strings of zeros from the file. If the file's data is sparse, it typically shrinks to occupy a fraction of the disk space it would otherwise require. On subsequent writes to the file, NTFS allocates space only for runs that contain nonzero data.

Figure 12-37 depicts the runs of a compressed file containing sparse data. Notice that certain ranges of the file's VCNs (16–31 and 64–127) have no disk allocations.

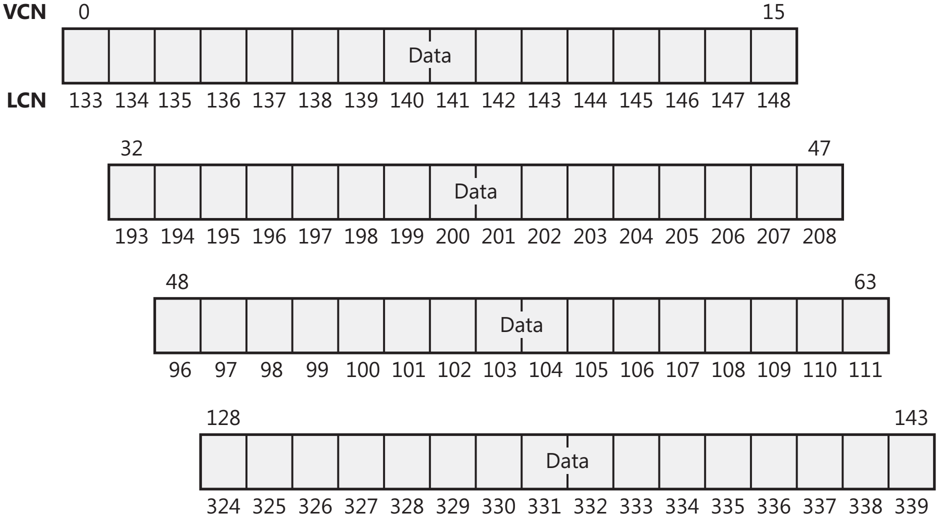
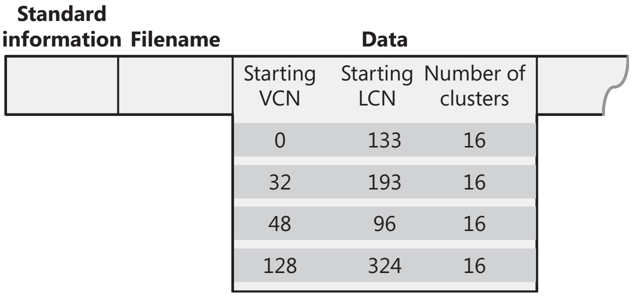


Figure 12-37 Runs of a compressed file containing sparse data

The MFT record for this compressed file omits blocks of VCNs that contain zeros and therefore have no physical storage allocated to them. The first data entry in Figure 12-38, for example, starts at VCN 0 and covers 16 clusters. The second entry jumps to VCN 32 and covers 16 clusters.



Starting VCN	Starting LCN	Number of clusters
0	133	16
32	193	16
48	96	16
128	324	16

Figure 12-38 MFT record for a compressed file containing sparse data

When a program reads data from a compressed file, NTFS checks the MFT record to determine whether a VCN-to-LCN mapping covers the location being read. If the program is reading from an unallocated "hole" in the file, it means that the data in that part of the file consists of zeros, so NTFS returns zeros without accessing the disk. If a program writes nonzero data to a "hole," NTFS quietly allocates disk space and then writes the data. This technique is very efficient for sparse file data that contains a lot of zero data.

Compressing Nonsparse Data

The preceding example of compressing a sparse file is somewhat contrived. It describes "compression" for a case in which whole sections of a file were filled with zeros but the remaining data in the file wasn't affected by the compression. The data in most files isn't sparse, but it can still be compressed by the application of a compression algorithm.

In NTFS, users can specify compression for individual files or for all the files in a directory. (New files created in a directory marked compressed are automatically compressed—existing files must be compressed individually.) When it compresses a file, NTFS divides the file's unprocessed data into *compression units* 16 clusters long (equal to 8 KB for a 512-byte cluster, for example). Certain sequences of data in a file might not compress much, if at all; so for each compression unit in the file, NTFS determines whether compressing the unit will save at least 1 cluster of storage. If compressing the unit won't free up at least 1 cluster, NTFS allocates a 16-cluster run and writes the data in that unit to disk without compressing it. If the data in a 16-cluster unit will compress to 15 or fewer clusters, NTFS allocates only the number of clusters needed to contain the compressed data and then writes it to disk. Figure 12-39 illustrates the compression of a file with four runs. The unshaded areas in this figure represent the actual storage locations that the file occupies after compression. The first, second, and fourth runs were compressed; the third run wasn't. Even with one noncompressed run, compressing this file saved 26 clusters of disk space, or 41 percent.

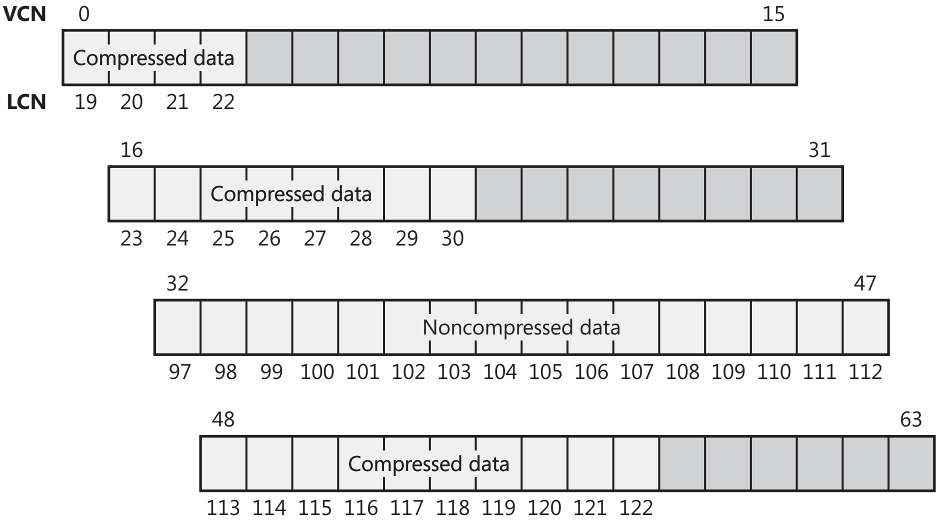
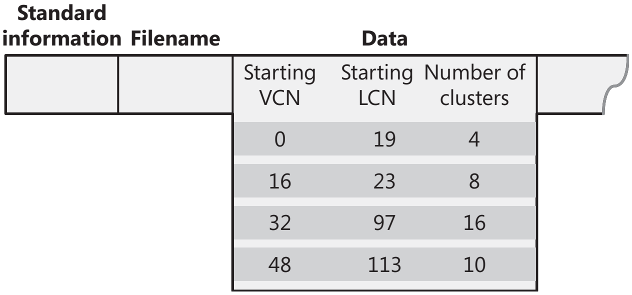


Figure 12-39 Data runs of a compressed file

Note Although the diagrams in this chapter show contiguous LCNs, a compression unit need not be stored in physically contiguous clusters. Runs that occupy noncontiguous clusters produce slightly more complicated MFT records than the one shown in Figure 12-39.

When it writes data to a compressed file, NTFS ensures that each run begins on a virtual 16-cluster boundary. Thus the starting VCN of each run is a multiple of 16, and the runs are no longer than 16 clusters. NTFS reads and writes at least one compression unit at a time when it accesses compressed files. When it writes compressed data, however, NTFS tries to store compression units in physically contiguous locations so that it can read them all in a single I/O operation. The 16-cluster size of the NTFS compression unit was chosen to reduce internal fragmentation: the larger the compression unit, the less the overall disk space needed to store the data. This 16-cluster compression unit size represents a trade-off between producing smaller compressed files and slowing read operations for programs that randomly access files. The equivalent of 16 clusters must be decompressed for each cache miss. (A cache miss is more likely to occur during random file access.) Figure 12-40 shows the MFT record for the compressed file shown in Figure 12-39.



Starting VCN	Starting LCN	Number of clusters
0	19	4
16	23	8
32	97	16
48	113	10

Figure 12-40 MFT record for a compressed file

One difference between this compressed file and the earlier example of a compressed file containing sparse data is that three of the compressed runs in this file are less than 16 clusters long. Reading this information from a file's MFT file record enables NTFS to know whether data in the file is compressed. Any run shorter than 16 clusters contains compressed data that NTFS must decompress when it first reads the data into the cache. A run that is exactly 16 clusters long doesn't contain compressed data and therefore requires no decompression.

If the data in a run has been compressed, NTFS decompresses the data into a scratch buffer and then copies it to the caller's buffer. NTFS also loads the decompressed data into the cache, which makes subsequent reads from the same run as fast as any other cached read. NTFS writes any updates to the file to the cache, leaving the lazy writer to compress and write the modified data to disk asynchronously. This strategy ensures that writing to a compressed file produces no more significant delay than writing to a noncompressed file would.

NTFS keeps disk allocations for a compressed file contiguous whenever possible. As the LCNs indicate, the first two runs of the compressed file shown in Figure 12-38 are physically contiguous, as are the last two. When two or more runs are contiguous, NTFS performs disk read-ahead, as it does with the data in other files. Because the reading and decompression of contiguous file data take place asynchronously before the program requests the data, subsequent read operations obtain the data directly from the cache, which greatly enhances read performance.

Sparse Files

Sparse files (the NTFS file type, as opposed to files that consist of sparse data, described earlier) are essentially compressed files for which NTFS doesn't apply compression to the file's nonsparse data. However, NTFS manages the run data of a sparse file's MFT record the same way it does for compressed files that consist of sparse and nonsparse data.

The Change Journal File

The change journal file, \$Extend\$UsnJrnl, is a sparse file that NTFS creates only when an application enables change logging. The journal stores change entries in the $J data stream. Entries include the following information about a file or directory change:

- The time of the change
- The change type (delete, rename, size extend, and so on)
- The file or directory's attributes
- The file or directory's name
- The file or directory's file reference number
- The file reference number of the file's parent directory

The journal is sparse so that it never overflows; when the journal's on-disk size exceeds the maximum defined for the file, NTFS simply begins zeroing the file data that precedes the window of change information having a size equal to the maximum journal size, as shown in Figure 12-41. To prevent constant resizing when an application is continuously exceeding the journal's size, NTFS shrinks the journal only when its size is twice an application-defined value over the maximum configured size.

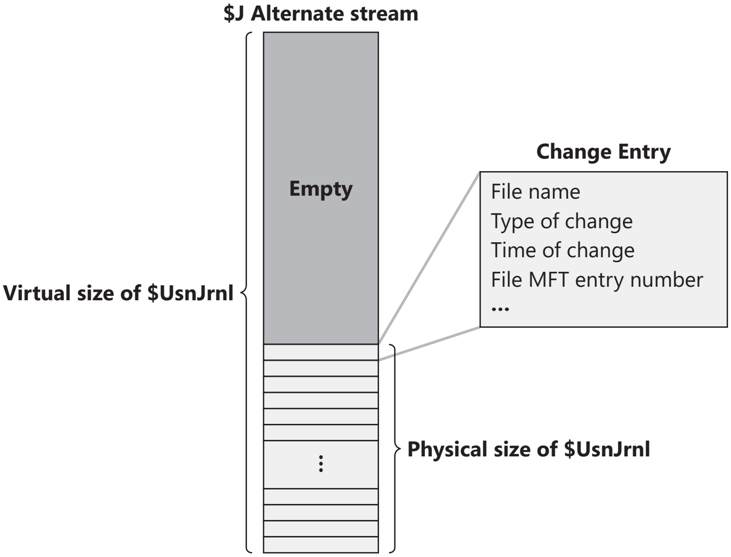


Figure 12-41 Change journal ($UsnJrnl) space allocation

Indexing

In NTFS, a file directory is simply an index of filenames—that is, a collection of filenames (along with their file references) organized in a particular way for quick access. To create a directory, NTFS indexes the filename attributes of the files in the directory. The MFT record for the root directory of a volume is shown in Figure 12-42.

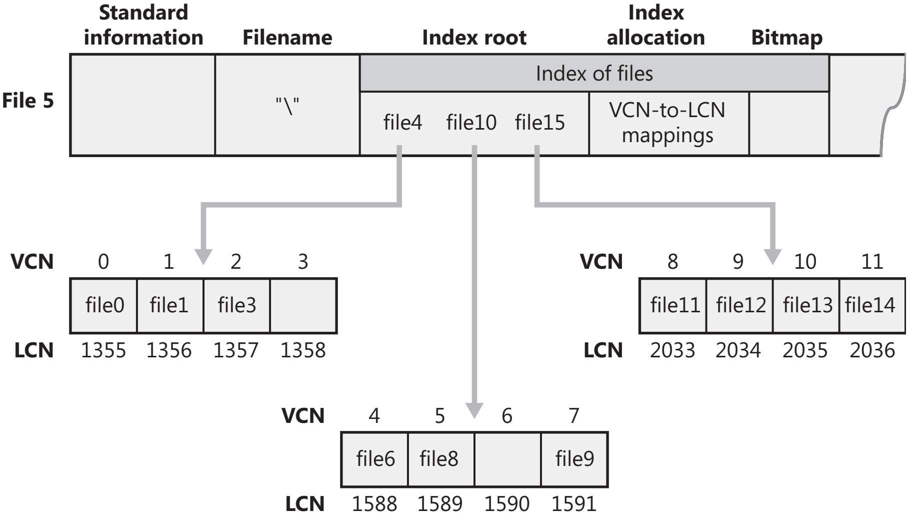


Figure 12-42 Filename index for a volume's root directory

Conceptually, an MFT entry for a directory contains in its index root attribute a sorted list of the files in the directory. For large directories, however, the filenames are actually stored in 4-KB fixed-size index buffers that contain and organize the filenames. Index buffers implement a *b+ tree* data structure, which minimizes the number of disk accesses needed to find a particular file, especially for large directories. The index root attribute contains the first level of the b+ tree (root subdirectories) and points to index buffers containing the next level (more subdirectories, perhaps, or files).

Figure 12-42 shows only filenames in the index root attribute and the index buffers (*file6*, for example), but each entry in an index also contains the file reference in the MFT where the file is described and time stamp and file size information for the file. NTFS duplicates the time stamp and file size information from the file's MFT record. This technique, which is used by FAT and NTFS, requires updated information to be written in two places. Even so, it's a significant speed optimization for directory browsing because it enables the file system to display each file's time stamps and size without opening every file in the directory.

The index allocation attribute maps the VCNs of the index buffer runs to the LCNs that indicate where the index buffers reside on the disk, and the bitmap attribute keeps track of which VCNs in the index buffers are in use and which are free. Figure 12-42 shows one file entry per VCN (that is, per cluster), but filename entries are actually packed into each cluster. Each 4-KB index buffer can contain about 20 to 30 filename entries.

The b+ tree data structure is a type of balanced tree that is ideal for organizing sorted data stored on a disk because it minimizes the number of disk accesses needed to find an entry. In the MFT, a directory's index root attribute contains several filenames that act as indexes into the second level of the b+ tree. Each filename in the index root attribute has an optional pointer associated with it that points to an index buffer. The index buffer it points to contains filenames with lexicographic values less than its own. In Figure 12-42, for example, *file4* is a first-level entry in the b+ tree. It points to an index buffer containing filenames that are (lexicographically) less than itself—the filenames *file0*, *file1*, and *file3*. Note that the names *file1*, *file2*, and so on that are used in this example are not literal filenames but names intended to show the relative placement of files that are lexicographically ordered according to the displayed sequence.

Storing the filenames in b+ trees provides several benefits. Directory lookups are fast because the filenames are stored in a sorted order. And when higher-level software enumerates the files in a directory, NTFS returns already-sorted names. Finally, because b+ trees tend to grow wide rather than deep, NTFS's fast lookup times don't degrade as directories grow.

NTFS also provides general support for indexing data besides filenames, and several NTFS features—including object IDs, quota tracking, and consolidated security—use indexing to manage internal data.

Object IDs

In addition to storing the object ID assigned to a file or directory in the $OBJECT_ID attribute of its MFT record, NTFS also keeps the correspondence between object IDs and their file reference numbers in the $O index of the \$Extend\$ObjId metadata file. The index collates entries by object ID, making it easy for NTFS to quickly locate a file based on its ID. This feature allows applications, using undocumented native API functionality, to open a file or directory using its object ID. Figure 12-43 demonstrates the correspondence of the $Objid metadata file and $OBJECT_ID attributes in MFT records.

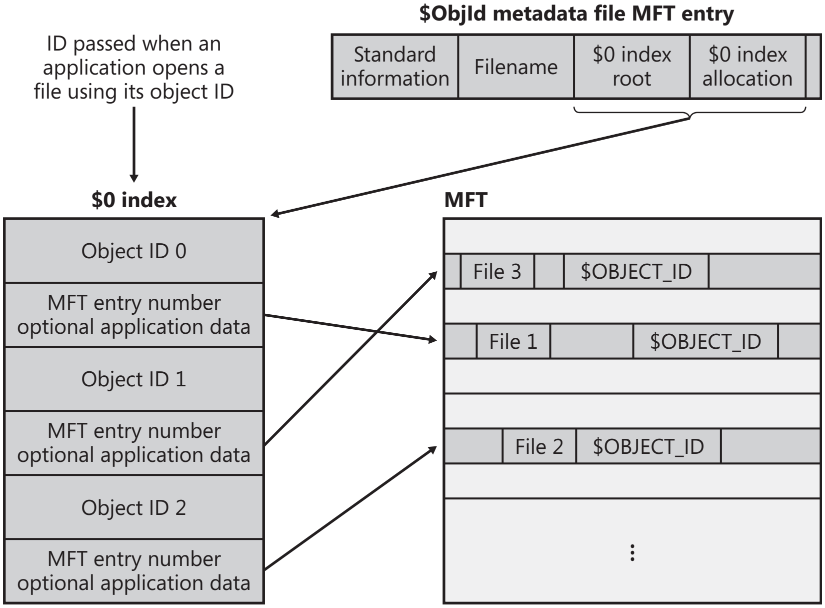

$ObjId metadata file MFT entry
Standard information
Filename
$0 index root
$0 index allocation
ID passed when an application opens a file using its object ID
$0 index
Object ID 0
MFT entry number optional application data
Object ID 1
MFT entry number optional application data
Object ID 2
MFT entry number optional application data
MFT
File 3
$OBJECT_ID
File 1
$OBJECT_ID
File 2
$OBJECT_ID
⋮

Figure 12-43 $Objid and $OBJECT_ID relationships

Quota Tracking

NTFS stores quota information in the \$Extend\$Quota metadata file, which consists of the indexes $O and $Q. Figure 12-44 shows the organization of these indexes. Just as NTFS assigns each security descriptor a unique internal security ID, NTFS assigns each user a unique user ID. When an administrator defines quota information for a user, NTFS allocates a user ID that corresponds to the user's SID. In the $O index, NTFS creates an entry that maps a SID to a user ID and sorts the index by user ID; in the $Q index, NTFS creates a quota control entry. A quota control entry contains the value of the user's quota limits, as well as the amount of disk space the user consumes on the volume.

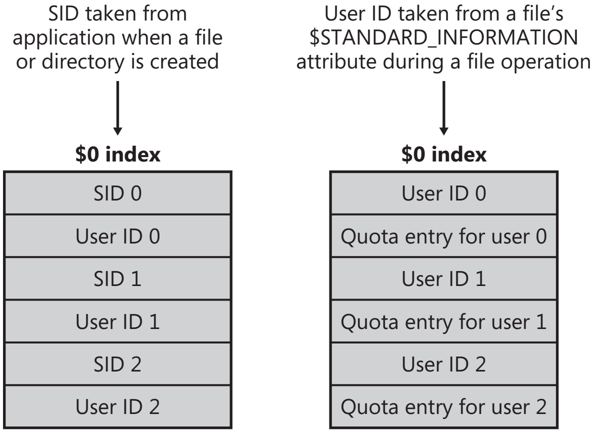


Figure 12-44 $Quota indexing

When an application creates a file or directory, NTFS obtains the application user's SID and looks up the associated user ID in the $O index. NTFS records the user ID in the new file or directory's $STANDARD_INFORMATION attribute, which counts all disk space allocated to the file or directory against that user's quota. Then NTFS looks up the quota entry in the $Q index and determines whether the new allocation causes the user to exceed his or her warning or limit threshold. When a new allocation causes the user to exceed a threshold, NTFS takes appropriate steps, such as logging an event to the System event log or not letting the user create the file or directory. As a file or directory changes size, NTFS updates the quota control entry associated with the user ID stored in the $STANDARD_INFORMATION attribute. NTFS uses general indexing to efficiently correlate user IDs with account SIDs, and, given a user ID, to efficiently look up a user's quota control information.

Consolidated Security

NTFS has always supported security, which lets an administrator specify which users can and can't access individual files and directories. In pre–Windows 2000 NTFS, every file and directory stores its security descriptor in its own security attribute. In most cases, administrators apply the same security settings to an entire directory tree, which results in duplication of security descriptors across all the files and subdirectories to which the settings apply. This duplication can intensively utilize disk space in multiuser environments, such as Windows 2000 Server Terminal Services, in which security descriptors might contain entries for multiple accounts. NTFS in Windows 2000 and later optimizes disk utilization for security descriptors by using a central metadata file named $Secure to store only one instance of each security descriptor on a volume.

The $Secure file contains two index attributes—$SDH and $SII—and a data-stream attribute named $SDS, as Figure 12-45 shows. NTFS assigns every unique security descriptor on a volume an internal NTFS security ID (not to be confused with a SID, which uniquely identifies computers and user accounts) and hashes the security descriptor according to a simple hash algorithm. A hash is a potentially nonunique shorthand representation of a descriptor. Entries in the $SDH index map the security descriptor hashes to the security descriptor's storage location within the $SDS data attribute, and the $SII index entries map NTFS security IDs to the security descriptor's location in the $SDS data attribute.

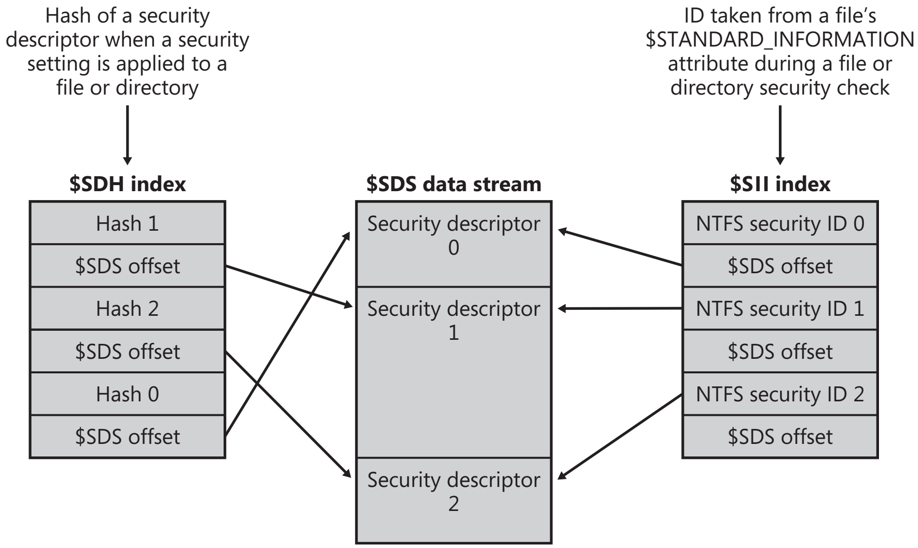


Figure 12-45 $Secure indexing

When you apply a security descriptor to a file or directory, NTFS obtains a hash of the descriptor and looks through the $SDH index for a match. NTFS sorts the $SDH index entries according to the hash of their corresponding security descriptor and stores the entries in a b+ tree. If NTFS finds a match for the descriptor in the $SDH index, NTFS locates the offset of the entry's security descriptor from the entry's offset value and reads the security descriptor from the $SDS attribute. If the hashes match but the security descriptors don't, NTFS looks for another matching entry in the $SDH index. When NTFS finds a precise match, the file or directory to which you're applying the security descriptor can reference the existing security descriptor in the $SDS attribute. NTFS makes the reference by reading the NTFS security identifier from the $SDH entry and storing it in the file or directory's $STANDARD_INFORMATION attribute. The NTFS $STANDARD_INFORMATION attribute, which all files and directories have, stores basic information about a file, including its attributes, timestamp information, and file's security identifier.

If NTFS doesn't find in the $SDH index an entry that has a security descriptor that matches the descriptor you're applying, the descriptor you're applying is unique to the volume and NTFS assigns the descriptor a new internal security ID. NTFS internal security IDs are 32-bit values, whereas SIDs are typically several times larger, so representing SIDs with NTFS security IDs saves space in the $STANDARD_ INFORMATION attribute. NTFS then adds the security descriptor to the $SDS attribute, which is sorted in a b+ tree by the NTFS security ID, and it adds to the $SDH and $SII indexes entries that reference the descriptor's offset in the $SDS data.

When an application attempts to open a file or directory, NTFS uses the $SII index to look up the file or directory's security descriptor. NTFS reads the file or directory's internal security ID from the MFT entry's $STANDARD_INFORMATION attribute. It then uses the $Secure file's

$SII index to locate the ID's entry in the $SDS attribute. The offset into the $SDS attribute lets NTFS read the security descriptor and complete the security check. NTFS stores the 32 most recently accessed security descriptors with their $SII indices in a cache so that it will access only the $Secure file when the $SII isn't cached.

NTFS doesn't delete entries in the $Secure file, even if no file or directory on a volume references the entry. Not deleting these entries doesn't significantly decrease disk space because most volumes, even those used for long periods, have relatively few unique security descriptors.

NTFS's use of general indexing lets files and directories that have the same security settings efficiently share security descriptors. The $SII index lets NTFS quickly look up a security descriptor in the $Secure file while performing security checks, and the $SDH index lets NTFS quickly determine whether a security descriptor being applied to a file or directory is already stored in the $Secure file and can be shared.

Reparse Points

As described earlier in the chapter, a *reparse point* is a block of up to 16 KB of application-defined reparse data and a 32-bit reparse tag that are stored in the $REPARSE_POINT attribute of a file or directory. Whenever an application creates or deletes a reparse point, NTFS updates the \$Extend\$Reparse metadata file, in which NTFS stores entries that identify the file record numbers of files and directories that contain reparse points. Storing the records in a central location enables NTFS to provide interfaces for applications to enumerate all a volume's reparse points or just specific types of reparse points, such as mount points. (See Chapter 10 for more information on mount points.) The \$Extend\$Reparse file uses the general indexing facility of NTFS by collating the file's entries (in an index named $R) by reparse point tags.

NTFS Recovery Support

NTFS recovery support ensures that if a power failure or a system failure occurs, no file system operations (transactions) will be left incomplete and the structure of the disk volume will remain intact without the need to run a disk repair utility. The NTFS Chkdsk utility is used to repair catastrophic disk corruption caused by I/O errors (bad disk sectors, electrical anomalies, or disk failures, for example) or software bugs. But with the NTFS recovery capabilities in place, Chkdsk is rarely needed.

As mentioned earlier (in the section "Recoverability"), NTFS uses a transaction-processing scheme to implement recoverability. This strategy ensures a full disk recovery that is also extremely fast (on the order of seconds) for even the largest disks. NTFS limits its recovery procedures to file system data to ensure that at the very least the user will never lose a volume because of a corrupted file system; however, unless an application takes specific action (such as flushing cached files to disk), NTFS doesn't guarantee user data to be fully updated if a crash occurs. Transaction-based protection of user data is available in most of the database products available for Windows, such as Microsoft SQL Server. Microsoft chose not to implement user-data recovery at the file system level because applications often implement their own data-recovery schemes that are tailored for the type of data they store.

The following sections describe the evolution of file system reliability as a context for an introduction to recoverable file systems, detail the transaction-logging scheme NTFS uses to record modifications to file system data structures, and explain how NTFS recovers a volume if the system fails.

Evolution of File System Design

The development of a recoverable file system was a step forward in the evolution of file system design. In the past, two techniques were common for constructing a file system's I/O and caching support: *careful write* and *lazy write*. The file systems developed for Digital Equipment Corporation (now Compaq) VAX/VMS and for some other proprietary operating systems employed a careful write algorithm, while OS/2 HPFS and most older UNIX file systems used a lazy write file system scheme.

The next two subsections briefly review these two types of file systems and their intrinsic trade-offs between safety and performance. The third subsection describes NTFS's recoverable approach and explains how it differs from the other two strategies.

Careful Write File Systems

When an operating system crashes or loses power, I/O operations in progress are immediately, and often prematurely, interrupted. Depending on what I/O operation or operations were in progress and how far along they were, such an abrupt halt can produce inconsistencies in a file system. An inconsistency in this context is a file system corruption—a filename appears in a directory listing, for example, but the file system doesn't know the file is there or can't access the file. The worst file system corruptions can leave an entire volume inaccessible.

A careful write file system doesn't try to prevent file system inconsistencies. Rather, it orders its write operations so that, at worst, a system crash will produce predictable, noncritical inconsistencies, which the file system can fix at its leisure.

When any kind of file system receives a request to update the disk, it must perform several suboperations before the update will be complete. In a file system that uses the careful write strategy, the suboperations are always written to disk serially. When allocating disk space for a file, for example, the file system first sets some bits in its bitmap and then allocates the space to the file. If the power fails immediately after the bits are set, the careful write file system loses access to some disk space—to the space represented by the set bits—but existing data isn't corrupted.

Serializing write operations also means that I/O requests are filled in the order in which they are received. If one process allocates disk space and shortly thereafter another process creates a file, a careful write file system completes the disk allocation before it starts to create the file because interleaving the suboperations of the two I/O requests could result in an inconsistent state.

> **Note** The FAT file system uses a *write-through* algorithm that causes disk modifications to be immediately written to the disk. Unlike the careful write approach, the write-through technique doesn't require the file system to order its writes to prevent inconsistencies.

The main advantage of a careful write file system is that in the event of a failure the volume stays consistent and usable without the need to immediately run a slow volume-repair utility. Such a utility is needed to correct the predictable, nondestructive disk inconsistencies that occur as the result of a system failure, but the utility can be run at a convenient time, typically when the system is rebooted.

Lazy Write File Systems

A careful write file system sacrifices speed for the safety it provides. A lazy write file system improves performance by using a *write-back* caching strategy; that is, it writes file modifications to the cache and flushes the contents of the cache to disk in an optimized way, usually as a background activity.

The performance improvements associated with the lazy write caching technique take several forms. First, the overall number of disk writes is reduced. Because serialized, immediate disk writes aren't required, the contents of a buffer can be modified several times before they are written to disk. Second, the speed of servicing application requests is greatly increased because the file system can return control to the caller without waiting for disk writes to be completed. Finally, the lazy write strategy ignores the inconsistent intermediate states on a file volume that can result when the suboperations of two or more I/O requests are interleaved. It is thus easier to make the file system multithreaded, allowing more than one I/O operation to be in progress at a time.

The disadvantage of the lazy write technique is that it creates intervals during which a volume is in too inconsistent a state to be corrected by the file system. Consequently, lazy write file systems must keep track of the volume's state at all times. In general, lazy write file systems gain a performance advantage over careful write systems—at the expense of greater risk and user inconvenience if the system fails.

Recoverable File Systems

A recoverable file system such as NTFS tries to exceed the safety of a careful write file system while achieving the performance of a lazy write file system. A recoverable file system ensures volume consistency by using logging techniques (sometimes called *journaling*) originally developed for transaction processing. If the operating system crashes, the recoverable file system restores consistency by executing a recovery procedure that accesses information that has been stored in a log file. Because the file system has logged its disk writes, the recovery procedure takes only seconds, regardless of the size of the volume.

The recovery procedure for a recoverable file system is exact, guaranteeing that the volume will be restored to a consistent state. In NTFS, none of the inadequate restorations associated with lazy write file systems can happen.

A recoverable file system incurs some costs for the safety it provides. Every transaction that alters the volume structure requires that one record be written to the log file for each of the transaction's suboperations. This logging overhead is ameliorated by the file system's *batching* of log records—writing many records to the log file in a single I/O operation. In addition, the recoverable file system can employ the optimization techniques of a lazy write file system. It can even increase the length of the intervals between cache flushes because the file system can be recovered if the system crashes before the cache changes have been flushed to disk. This gain over the caching performance of lazy write file systems makes up for, and often exceeds, the overhead of the recoverable file system's logging activity.

Neither careful write nor lazy write file systems guarantee protection of user file data. If the system crashes while an application is writing a file, the file can be lost or corrupted. Worse, the crash can corrupt a lazy write file system, destroying existing files or even rendering an entire volume inaccessible.

The NTFS recoverable file system implements several strategies that improve its reliability over that of the traditional file systems. First, NTFS recoverability guarantees that the volume structure won't be corrupted, so all files will remain accessible after a system failure.

Second, although NTFS doesn't guarantee protection of user data in the event of a system crash—some changes can be lost from the cache—applications can take advantage of the NTFS write-through and cache-flushing capabilities to ensure that file modifications are recorded on disk at appropriate intervals. Both *cache write-through*—forcing write operations to be immediately recorded on disk—and *cache flushing*—forcing cache contents to be written to disk—are efficient operations. NTFS doesn't have to do extra disk I/O to flush modifications to several different file system data structures because changes to the data structures are recorded—in a single write operation—in the log file; if a failure occurs and cache contents are lost, the file system modifications can be recovered from the log. Furthermore, unlike the FAT file system, NTFS guarantees that user data will be consistent and available immediately after a write-through operation or a cache flush, even if the system subsequently fails.

Logging

NTFS provides file system recoverability by means of a transaction-processing technique called *logging*. In NTFS logging, the suboperations of any transaction that alters important file system data structures are recorded in a log file before they are carried through on the disk. That way, if the system crashes, partially completed transactions can be redone or undone when the system comes back on line. In transaction processing, this technique is known as *write-ahead logging*. In NTFS, transactions include writing to the disk or deleting a file and can be made up of several suboperations.

Log File Service (LFS)

The log file service (LFS) is a series of kernel-mode routines inside the NTFS driver that NTFS uses to access the log file. Although originally designed to provide logging and recovery services for more than one client, the LFS is used only by NTFS. The caller—NTFS in this case—passes the LFS a pointer to an open file object, which specifies a log file to be accessed. The LFS either initializes a new log file or calls the Windows cache manager to access the existing log file through the cache, as shown in Figure 12-46.

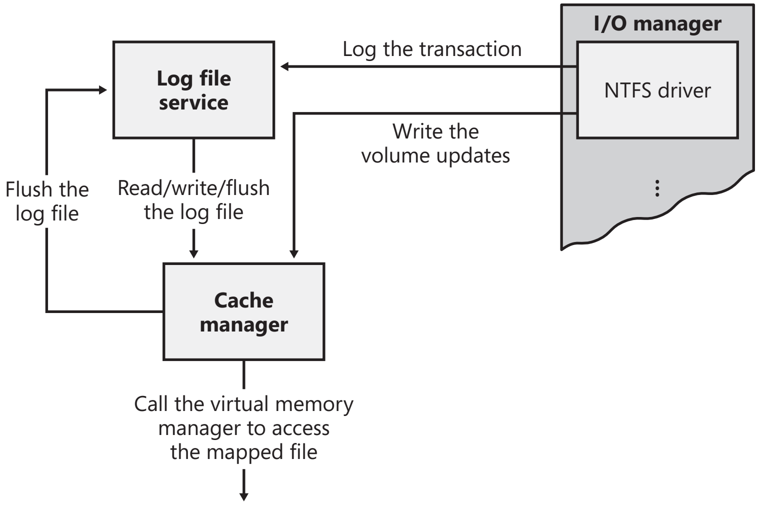


Figure 12-46 Log file service (LFS)

The LFS divides the log file into two regions: a *restart area* and an "infinite" *logging area*, as shown in Figure 12-47.

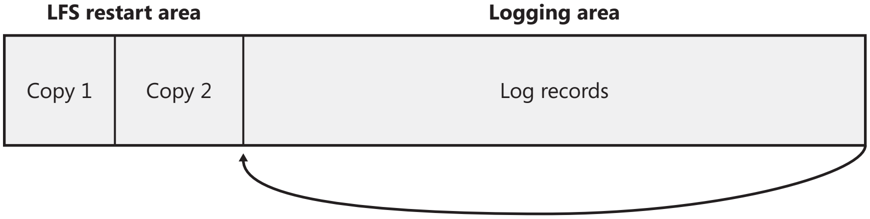


Figure 12-47 Log file regions

NTFS calls the LFS to read and write the restart area. NTFS uses the restart area to store context information such as the location in the logging area at which NTFS will begin to read during recovery after a system failure. The LFS maintains a second copy of the restart data in case the first becomes corrupted or otherwise inaccessible. The remainder of the log file is the logging area, which contains transaction records NTFS writes to recover a volume in the event of a system failure. The LFS makes the log file appear infinite by reusing it circularly (while guaranteeing that it doesn't overwrite information it needs). The LFS uses *logical sequence numbers*

(LSNs) to identify records written to the log file. As the LFS cycles through the file, it increases the values of the LSNs. NTFS uses 64 bits to represent LSNs, so the number of possible LSNs is so large as to be virtually infinite.

NTFS never reads transactions from or writes transactions to the log file directly. The LFS provides services NTFS calls to open the log file, write log records, read log records in forward or backward order, flush log records up to a particular LSN, or set the beginning of the log file to a higher LSN. During recovery, NTFS calls the LFS to perform the following actions: read forward through the log records to redo any transactions that were recorded in the log file but weren't flushed to disk at the time of the system failure; read backward through the log records to undo, or roll back, any transactions that weren't completely logged before the crash; and set the beginning of the log file to a record with a higher LSN when NTFS no longer needs the older transaction records in the log file.

Here's how the system guarantees that the volume can be recovered:

1. NTFS first calls the LFS to record in the (cached) log file any transactions that will modify the volume structure.
2. NTFS modifies the volume (also in the cache).
3. The cache manager prompts the LFS to flush the log file to disk. (The LFS implements the flush by calling the cache manager back, telling it which pages of memory to flush. Refer back to the calling sequence shown in Figure 12-46.)
4. After the cache manager flushes the log file to disk, it flushes the volume changes (the metadata operations themselves) to disk.

These steps ensure that if the file system modifications are ultimately unsuccessful, the corresponding transactions can be retrieved from the log file and can be either redone or undone as part of the file system recovery procedure.

File system recovery begins automatically the first time the volume is used after the system is rebooted. NTFS checks whether the transactions that were recorded in the log file before the crash were applied to the volume, and if they weren't, it redoes them. NTFS also guarantees that transactions not completely logged before the crash are undone so that they don't appear on the volume.

Log Record Types

The LFS allows its clients to write any kind of record to their log files. NTFS writes several types of records. Two types, *update records* and *checkpoint records*, are described here.

Update records Update records are the most common type of record NTFS writes to the log file. Each update record contains two kinds of information:

- **Redo information** How to reapply one suboperation of a fully logged ("committed") transaction to the volume if a system failure occurs before the transaction is flushed from the cache
- **Undo information** How to reverse one suboperation of a transaction that was only partially logged ("not committed") at the time of a system failure

Figure 12-48 shows three update records in the log file. Each record represents one suboperation of a transaction, creating a new file. The redo entry in each update record tells NTFS how to reapply the suboperation to the volume, and the undo entry tells NTFS how to roll back (undo) the suboperation.

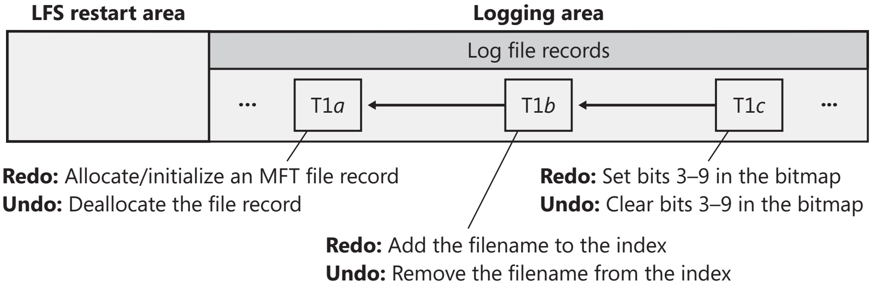

LFS restart area
Logging area
Log file records
...
T1*a*
T1*b*
T1*c*
...
Redo: Allocate/initialize an MFT file record
Undo: Deallocate the file record
Redo: Set bits 3–9 in the bitmap
Undo: Clear bits 3–9 in the bitmap
Redo: Add the filename to the index
Undo: Remove the filename from the index

Figure 12-48 Update records in the log file

After logging a transaction (in this example, by calling the LFS to write the three update records to the log file), NTFS performs the suboperations on the volume itself, in the cache. When it has finished updating the cache, NTFS writes another record to the log file, recording the entire transaction as complete—a suboperation known as *committing* a transaction. Once a transaction is committed, NTFS guarantees that the entire transaction will appear on the volume, even if the operating system subsequently fails.

When recovering after a system failure, NTFS reads through the log file and redoes each committed transaction. Although NTFS completed the committed transactions before the system failure, it doesn't know whether the cache manager flushed the volume modifications to disk in time. The updates might have been lost from the cache when the system failed. Therefore, NTFS executes the committed transactions again just to be sure that the disk is up to date.

After redoing the committed transactions during a file system recovery, NTFS locates all the transactions in the log file that weren't committed at failure and rolls back (undoes) each suboperation that had been logged. In Figure 12-47, NTFS would first undo the T1*c* suboperation and then follow the backward pointer to T1*b* and undo that suboperation. It would continue to follow the backward pointers, undoing suboperations, until it reached the first suboperation in the transaction. By following the pointers, NTFS knows how many and which update records it must undo to roll back a transaction.

Redo and undo information can be expressed either physically or logically. Physical descriptions specify volume updates in terms of particular byte ranges on the disk that are to be changed, moved, and so on. Logical descriptions express updates in terms of operations such as "delete file A.dat." As the lowest layer of software maintaining the file system structure, NTFS writes update records with physical descriptions. Transaction-processing or other application-level software might benefit from writing update records in logical terms, however, because logically expressed updates are more compact than physically expressed ones. Logical descriptions necessarily depend on NTFS to understand what operations such as deleting a file involve.

NTFS writes update records (usually several) for each of the following transactions:

- Creating a file
- Deleting a file
- Extending a file
- Truncating a file
- Setting file information
- Renaming a file
- Changing the security applied to a file

The redo and undo information in an update record must be carefully designed because although NTFS undoes a transaction, recovers from a system failure, or even operates normally, it might try to redo a transaction that has already been done or, conversely, to undo a transaction that never occurred or that has already been undone. Similarly, NTFS might try to redo or undo a transaction consisting of several update records, only some of which are complete on disk. The format of the update records must ensure that executing redundant redo or undo operations is *idempotent*, that is, has a neutral effect. For example, setting a bit that is already set has no effect, but toggling a bit that has already been toggled does. The file system must also handle intermediate volume states correctly.

Checkpoint records In addition to update records, NTFS periodically writes a checkpoint record to the log file, as illustrated in Figure 12-49.

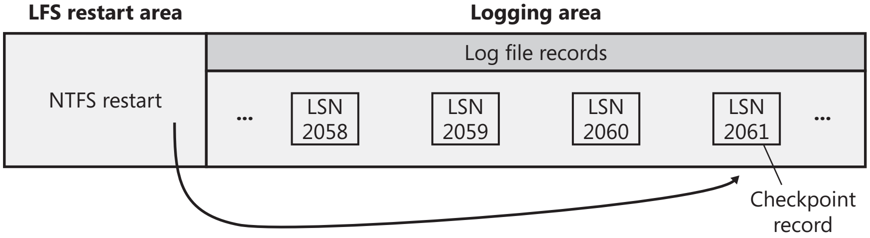


Figure 12-49 Checkpoint record in the log file

A checkpoint record helps NTFS determine what processing would be needed to recover a volume if a crash were to occur immediately. Using information stored in the checkpoint record, NTFS knows, for example, how far back in the log file it must go to begin its recovery. After writing a checkpoint record, NTFS stores the LSN of the record in the restart area so that it can quickly find its most recently written checkpoint record when it begins file system recovery after a crash occurs.

Although the LFS presents the log file to NTFS as if it were infinitely large, it isn't. The generous size of the log file and the frequent writing of checkpoint records (an operation that usually frees up space in the log file) make the possibility of the log file's filling up a remote one. Nevertheless, the LFS accounts for this possibility by tracking several numbers:

- The available log space
- The amount of space needed to write an incoming log record and to undo the write, should that be necessary
- The amount of space needed to roll back all active (noncommitted) transactions, should that be necessary

If the log file doesn't contain enough available space to accommodate the total of the last two items, the LFS returns a "log file full" error and NTFS raises an exception. The NTFS exception handler rolls back the current transaction and places it in a queue to be restarted later.

To free up space in the log file, NTFS must momentarily prevent further transactions on files. To do so, NTFS blocks file creation and deletion and then requests exclusive access to all system files and shared access to all user files. Gradually, active transactions either are completed successfully or receive the "log file full" exception. NTFS rolls back and queues the transactions that receive the exception.

Once it has blocked transaction activity on files as just described, NTFS calls the cache manager to flush unwritten data to disk, including unwritten log file data. After everything is safely flushed to disk, NTFS no longer needs the data in the log file. It resets the beginning of the log file to the current position, making the log file "empty." Then it restarts the queued transactions. Beyond the short pause in I/O processing, the "log file full" error has no effect on executing programs.

This scenario is one example of how NTFS uses the log file not only for file system recovery but also for error recovery during normal operation. You'll find out more about error recovery in the following section.

Recovery

NTFS automatically performs a disk recovery the first time a program accesses an NTFS volume after the system has been booted. (If no recovery is needed, the process is trivial.) Recovery depends on two tables NTFS maintains in memory:

- The *transaction table* keeps track of transactions that have been started but aren't yet committed. The suboperations of these active transactions must be removed from the disk during recovery.
- The *dirty page table* records which pages in the cache contain modifications to the file system structure that haven't yet been written to disk. This data must be flushed to disk during recovery.

NTFS writes a checkpoint record to the log file once every 5 seconds. Just before it does, it calls the LFS to store a current copy of the transaction table and of the dirty page table in the log file. NTFS then records in the checkpoint record the LSNs of the log records containing the copied tables. When recovery begins after a system failure, NTFS calls the LFS to locate the log records containing the most recent checkpoint record and the most recent copies of the transaction and dirty page tables. It then copies the tables to memory.

The log file usually contains more update records following the last checkpoint record. These update records represent volume modifications that occurred after the last checkpoint record was written. NTFS must update the transaction and dirty page tables to include these operations. After updating the tables, NTFS uses the tables and the contents of the log file to update the volume itself.

To effect its volume recovery, NTFS scans the log file three times, loading the file into memory during the first pass to minimize disk I/O. Each pass has a particular purpose:

1. Analysis
2. Redoing transactions
3. Undoing transactions

Analysis Pass

During the *analysis pass*, as shown in Figure 12-50, NTFS scans forward in the log file from the beginning of the last checkpoint operation to find update records and use them to update the transaction and dirty page tables it copied to memory. Notice in the figure that the checkpoint operation stores three records in the log file and that update records might be interspersed among these records. NTFS therefore must start its scan at the beginning of the checkpoint operation.

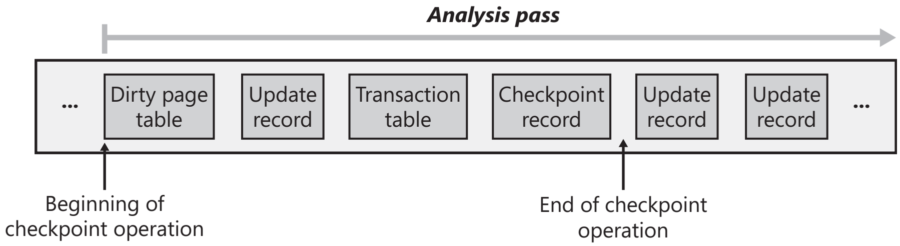


Figure 12-50 Analysis pass

Most update records that appear in the log file after the checkpoint operation begins represent a modification to either the transaction table or the dirty page table. If an update record is a "transaction committed" record, for example, the transaction the record represents must be removed from the transaction table. Similarly, if the update record is a "page update" record that modifies a file system data structure, the dirty page table must be updated to reflect that change.

Once the tables are up to date in memory, NTFS scans the tables to determine the LSN of the oldest update record that logs an operation that hasn't been carried out on disk. The transaction table contains the LSNs of the noncommitted (incomplete) transactions, and the dirty page table contains the LSNs of records in the cache that haven't been flushed to disk. The LSN of the oldest update record that NTFS finds in these two tables determines where the redo pass will begin. If the last checkpoint record is older, however, NTFS will start the redo pass there instead.

Redo Pass

During the *redo pass*, as shown in Figure 12-51, NTFS scans forward in the log file from the LSN of the oldest update record, which it found during the analysis pass. It looks for "page update" records, which contain volume modifications that were written before the system failure but that might not have been flushed to disk. NTFS redoes these updates in the cache.

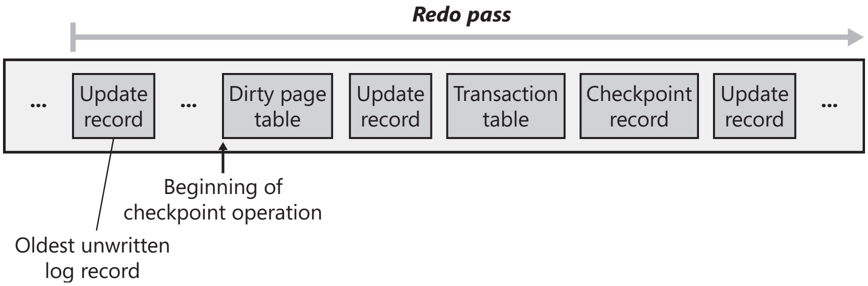


Figure 12-51 Redo pass

When NTFS reaches the end of the log file, it has updated the cache with the necessary volume modifications and the cache manager's lazy writer can begin writing cache contents to disk in the background.

Undo Pass

After it completes the redo pass, NTFS begins its *undo pass*, in which it rolls back any transactions that weren't committed when the system failed. Figure 12-52 shows two transactions in the log file; transaction 1 was committed before the power failure, but transaction 2 wasn't. NTFS must undo transaction 2.

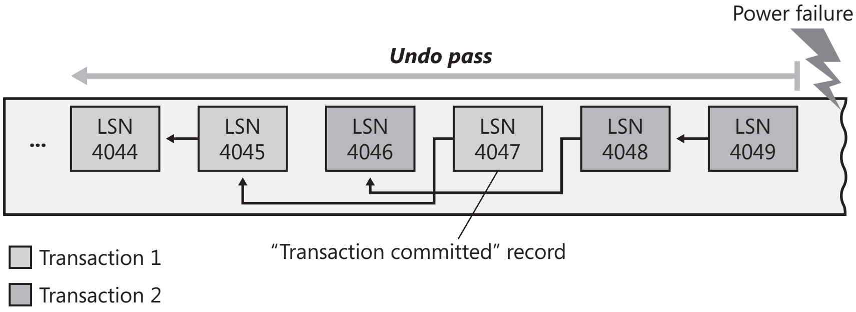


Figure 12-52 Undo pass

Suppose that transaction 2 created a file, an operation that comprises three suboperations, each with its own update record. The update records of a transaction are linked by backward pointers in the log file because they are usually not contiguous.

The NTFS transaction table lists the LSN of the last-logged update record for each noncommitted transaction. In this example, the transaction table identifies LSN 4049 as the last update record logged for transaction 2. As shown from right to left in Figure 12-53, NTFS rolls back transaction 2.

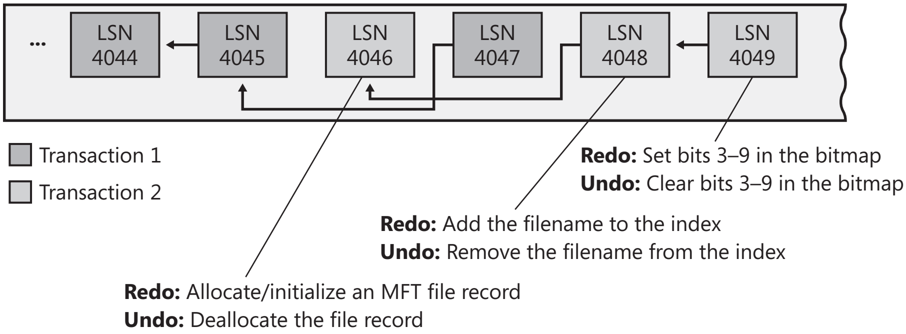

...
LSN 4044
LSN 4045
LSN 4046
LSN 4047
LSN 4048
LSN 4049
Transaction 1
Transaction 2
Redo: Set bits 3–9 in the bitmap
Undo: Clear bits 3–9 in the bitmap
Redo: Add the filename to the index
Undo: Remove the filename from the index
Redo: Allocate/initialize an MFT file record
Undo: Deallocate the file record

Figure 12-53 Undoing a transaction

After locating LSN 4049, NTFS finds the undo information and executes it, clearing bits 3 through 9 in its allocation bitmap. NTFS then follows the backward pointer to LSN 4048, which directs it to remove the new filename from the appropriate filename index. Finally, it follows the last backward pointer and deallocates the MFT file record reserved for the file, as the update record with LSN 4046 specifies. Transaction 2 is now rolled back. If there are other noncommitted transactions to undo, NTFS follows the same procedure to roll them back. Because undoing transactions affects the volume's file system structure, NTFS must log the undo operations in the log file. After all, the power might fail again during the recovery, and NTFS would have to redo its undo operations!

When the undo pass of the recovery is finished, the volume has been restored to a consistent state. At this point, NTFS flushes the cache changes to disk to ensure that the volume is up to date. NTFS then writes an "empty" LFS restart area to indicate that the volume is consistent and that no recovery need be done if the system should fail again immediately. Recovery is complete.

NTFS guarantees that recovery will return the volume to some preexisting consistent state, but not necessarily to the state that existed just before the system crash. NTFS can't make that guarantee because, for performance, it uses a "lazy commit" algorithm, which means that the log file isn't immediately flushed to disk each time a "transaction committed" record is written. Instead, numerous "transaction committed" records are batched and written together, either when the cache manager calls the LFS to flush the log file to disk or when the LFS writes a checkpoint record (once every 5 seconds) to the log file. Another reason the recovered volume might not be completely up to date is that several parallel transactions might be active when the system crashes and some of their "transaction committed" records might make it to disk whereas others might not. The consistent volume that recovery produces includes all the volume updates whose "transaction committed" records made it to disk and none of the updates whose "transaction committed" records didn't make it to disk.

NTFS uses the log file to recover a volume after the system fails, but it also takes advantage of an important "freebie" it gets from logging transactions. File systems necessarily contain a lot of code devoted to recovering from file system errors that occur during the course of normal file I/O. Because NTFS logs each transaction that modifies the volume structure, it can use the log file to recover when a file system error occurs and thus can greatly simplify its error handling code. The "log file full" error described earlier is one example of using the log file for error recovery.

Most I/O errors a program receives aren't file system errors and therefore can't be resolved entirely by NTFS. When called to create a file, for example, NTFS might begin by creating a file record in the MFT and then enter the new file's name in a directory index. When it tries to allocate space for the file in its bitmap, however, it could discover that the disk is full and the create request can't be completed. In such a case, NTFS uses the information in the log file to undo the part of the operation it has already completed and to deallocate the data structures it reserved for the file. Then it returns a "disk full" error to the caller, which in turn must respond appropriately to the error.

NTFS Bad-Cluster Recovery

The volume managers included with Windows, FtDisk (for basic disks) and Logical Disk Manager (LDM, for dynamic disks), can recover data from a bad sector on a fault-tolerant volume, but if the hard disk doesn't use the SCSI protocol or runs out of spare sectors, a volume manager can't perform sector sparing to replace the bad sector. (See Chapter 10 for more information on the volume managers.) When the file system reads from the sector, the volume manager instead recovers the data and returns the warning to the file system that there is only one copy of the data.

The FAT file system doesn't respond to this volume manager warning. Moreover, neither these file systems nor the volume managers keep track of the bad sectors, so a user must run the Chkdsk or Format utility to prevent a volume manager from repeatedly recovering data for the file system. Both Chkdsk and Format are less than ideal for removing bad sectors from use. Chkdsk can take a long time to find and remove bad sectors, and Format wipes all the data off the partition it's formatting.

In the file system equivalent of a volume manager's sector sparing, NTFS dynamically replaces the cluster containing a bad sector and keeps track of the bad cluster so that it won't be reused. (Recall that NTFS maintains portability by addressing logical clusters rather than physical sectors.) NTFS performs these functions when the volume manager can't perform sector sparing. When a volume manager returns a bad-sector warning or when the hard disk driver returns a bad-sector error, NTFS allocates a new cluster to replace the one containing the bad sector. NTFS copies the data that the volume manager has recovered into the new cluster to reestablish data redundancy.

Figure 12-54 shows an MFT record for a user file with a bad cluster in one of its data runs as it exited before the cluster went bad. When it receives a bad-sector error, NTFS reassigns the cluster containing the sector to its bad-cluster file. This prevents the bad cluster from being allocated to another file. NTFS then allocates a new cluster for the file and changes the file's VCN-to-LCN mappings to point to the new cluster. This bad-cluster remapping (introduced earlier in this chapter) is illustrated in Figure 12-55. Cluster number 1357, which contains the bad sector, is replaced by a new cluster, number 1049.

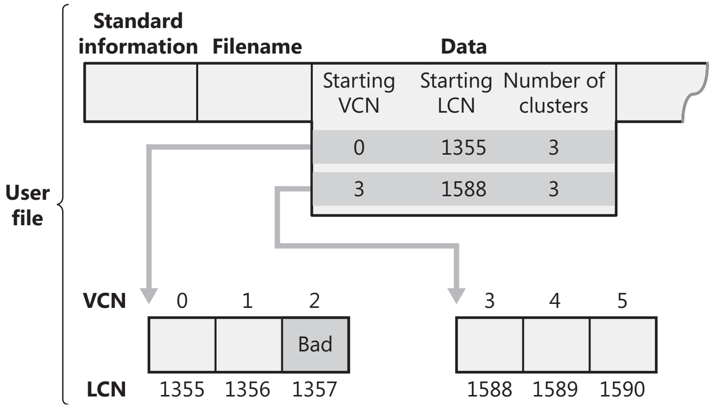


Figure 12-54 MFT record for a user file with a bad cluster

Bad-sector errors are undesirable, but when they do occur, the combination of NTFS and volume managers provides the best possible solution. If the bad sector is on a redundant volume, the volume manager recovers the data and replaces the sector if it can. If it can't replace the sector, it returns a warning to NTFS and NTFS replaces the cluster containing the bad sector.

If the volume isn't configured as a redundant volume, the data in the bad sector can't be recovered. When the volume is formatted as a FAT volume and the volume manager can't recover the data, reading from the bad sector yields indeterminate results. If some of the file system's control structures reside in the bad sector, an entire file or group of files (or potentially, the whole disk) can be lost. At best, some data in the affected file (often, all the data in the file beyond the bad sector) is lost. Moreover, the FAT file system is likely to reallocate the bad sector to the same or another file on the volume, causing the problem to resurface.

Like the other file systems, NTFS can't recover data from a bad sector without help from a volume manager. However, NTFS greatly contains the damage a bad sector can cause. If NTFS discovers the bad sector during a read operation, it remaps the cluster the sector is in, as shown in Figure 12-55. If the volume isn't configured as a redundant volume, NTFS returns a "data read" error to the calling program. Although the data that was in that cluster is lost, the rest of the file—and the file system—remains intact; the calling program can respond appropriately to the data loss, and the bad cluster won't be reused in future allocations. If NTFS discovers the bad cluster on a write operation rather than a read, NTFS remaps the cluster before writing and thus loses no data and generates no error.

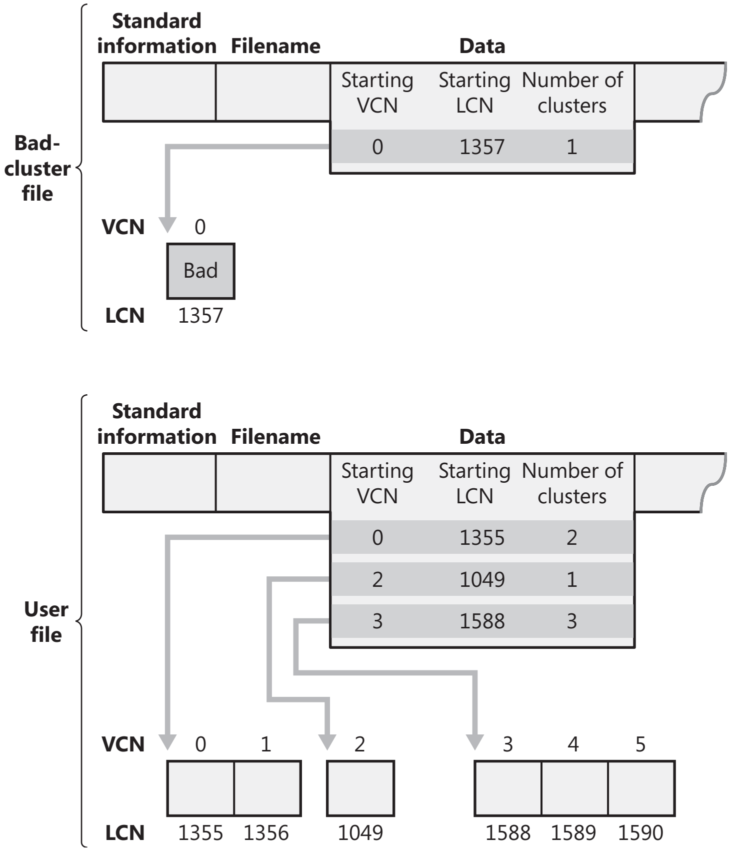


Figure 12-55 Bad-cluster remapping

The same recovery procedures are followed if file system data is stored in a sector that goes bad. If the bad sector is on a redundant volume, NTFS replaces the cluster dynamically, using the data recovered by the volume manager. If the volume isn't redundant, the data can't be recovered and NTFS sets a bit in the volume file that indicates corruption on the volume. The NTFS Chkdsk utility checks this bit when the system is next rebooted, and if the bit is set, Chkdsk executes, fixing the file system corruption by reconstructing the NTFS metadata.

In rare instances, file system corruption can occur even on a fault-tolerant disk configuration. A double error can destroy both file system data and the means to reconstruct it. If the system crashes while NTFS is writing the mirror copy of an MFT file record—of a filename index or of the log file, for example—the mirror copy of such file system data might not be fully updated. If the system were rebooted and a bad-sector error occurred on the primary disk at exactly the same location as the incomplete write on the disk mirror, NTFS would be unable to recover the correct data from the disk mirror. NTFS implements a special scheme for detecting such corruptions in file system data. If it ever finds an inconsistency, it sets the corruption bit in the volume file, which causes Chkdsk to reconstruct the NTFS metadata when the system is next rebooted. Because file system corruption is rare on a fault-tolerant disk configuration, Chkdsk is seldom needed. It is supplied as a safety precaution rather than as a first-line data recovery strategy.

The use of Chkdsk on NTFS is vastly different from its use on the FAT file system. Before writing anything to disk, FAT sets the volume's dirty bit and then resets the bit after the modification is complete. If any I/O operation is in progress when the system crashes, the dirty bit is left set and Chkdsk runs when the system is rebooted. On NTFS, Chkdsk runs only when unexpected or unreadable file system data is found and NTFS can't recover the data from a redundant volume or from redundant file system structures on a single volume. (The system boot sector is duplicated, as are the parts of the MFT required for booting the system and running the NTFS recovery procedure. This redundancy ensures that NTFS will always be able to boot and recover itself.)

Table 12-6 summarizes what happens when a sector goes bad on a disk volume formatted for one of the Windows-supported file systems according to various conditions we've described in this section.

Table 12-6 Summary of NTFS Data Recovery Scenarios

Scenario	With a SCSI disk that has spare sectors	With a non-SCSI disk or a disk with no spare sectors*
Fault tolerant volume†	1. Volume manager recovers the data. 2. Volume manager performs sector sparing (replaces the bad sector). 3. File system remains unaware of the error.	1. Volume manager recovers the data. 2. Volume manager sends the data and a bad-sector error to the file system. 3. NTFS performs cluster remapping.
Non-fault-tolerant volume	1. Volume manager can't recover the data. 2. Volume manager sends a bad-sector error to the file system. 3. NTFS performs cluster remapping. Data is lost‡.	1. Volume manager can't recover the data. 2. Volume manager sends a bad-sector error to the file system. 3. NTFS performs cluster remapping. Data is lost.‡

* In neither of these cases can a volume manager perform sector sparing: (1) hard disks that don't use the SCSI protocol have no standard interface for providing sector sparing; (2) some hard disks don't provide hardware support for sector sparing, and SCSI hard disks that do provide sector sparing can eventually run out of spare sectors if a lot of sectors go bad.

† A fault-tolerant volume is one of the following: a mirror set or a RAID-5 set.

‡ In a write operation, no data is lost: NTFS remaps the cluster before the write.

If the volume on which the bad sector appears is a fault-tolerant volume (a mirrored or RAID-5 volume) and if the hard disk is one that supports sector sparing (and that hasn't run out of spare sectors), it doesn't matter which file system you're using (FAT or NTFS). The volume manager replaces the bad sector without the need for user or file system intervention.

If a bad sector is located on a hard disk that doesn't support sector sparing, the file system is responsible for replacing (remapping) the bad sector or—in the case of NTFS—the cluster in which the bad sector resides. The FAT file system doesn't provide sector or cluster remapping. The benefits of NTFS cluster remapping are that bad spots in a file can be fixed without harm to the file (or harm to the file system, as the case may be) and that the bad cluster won't be reallocated to the same or another file.

Encrypting File System Security

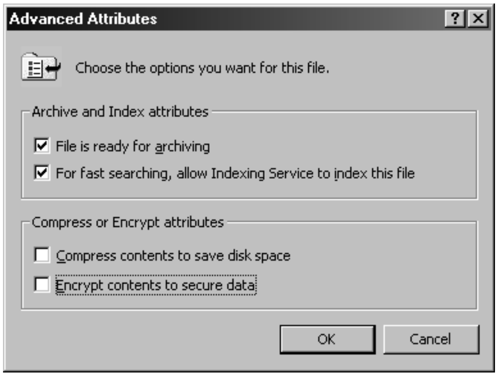
EFS security relies on cryptography support. The first time a file is encrypted, EFS assigns the account of the user performing the encryption a private/public key pair for use in file encryption. Users can encrypt files via Windows Explorer by opening a file's Properties dialog box, pressing Advanced, and selecting the Encrypt Contents To Secure Data option, as shown in Figure 12-56. Users can also encrypt files via a command-line utility named *cipher*. Windows automatically encrypts files that reside in directories that are designated as encrypted directories. When a file is encrypted, EFS generates a random number for the file that EFS calls the file's file encryption key (FEK). EFS uses the FEK to encrypt the file's contents with a stronger variant of the Data Encryption Standard (DES) algorithm—DESX on Windows 2000 and DESX, Triple-DES (3DES) or Advanced Encryption Standard (AES) on Windows XP (Service Pack 1 and later) and Windows Server 2003. EFS stores the file's FEK with the file but encrypts the FEK with the user's EFS public key by using the RSA public key–based encryption algorithm. After EFS completes these steps, the file is secure: other users can't decrypt the data without the file's decrypted FEK, and they can't decrypt the FEK without the private key.

EFS FEK Key Strength

The default FEK encryption algorithm on Windows 2000 and Windows XP is DESX, and on Windows XP with Service Pack 1 and higher and Windows Server 2003 it is AES. On Windows versions approved for export outside the United States, the EFS driver implements a 56-bit key DESX encryption. For the U.S.-only version of Windows and export versions with the 128-bit encryption pack, the key is 128 bits long. The Windows AES algorithm uses 256-bit keys. Use of 3DES allows access to larger sized keys, so if you require greater key strength you can enable 3DES encryption in one of two ways: either as the algorithm for all system cryptographic services or just for EFS.

To have 3DES be the encryption algorithm for all system cryptographic services, open the Local Security Policy Editor by entering secpol.msc in the Run dialog box from the Start menu and open the Security Options node under Local Policies. View the properties of System Cryptography: Use FIPS Compliant Algorithms For Encryption, Hashing And Signing, and enable it.

To enable 3DES for EFS only, create the DWORD value HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\EFS\AlgorithmID, set it to 0x6603, and reboot.



Advanced Attributes

Choose the options you want for this file.

Archive and Index attributes

☑ File is ready for archiving

☑ For fast searching, allow Indexing Service to index this file

Compress or Encrypt attributes

□ Compress contents to save disk space

□ Encrypt contents to secure data

Figure 12-56 Encrypt files by using the Advanced Attributes dialog box

EFS uses a private/public key algorithm to encrypt FEKs. To encrypt file data, EFS uses DESX, AES, or 3DES because all are *symmetric encryption algorithms*, which means that they use the same key to encrypt and decrypt data. Symmetric encryption algorithms are typically very fast, which makes them suitable for encrypting large amounts of data, such as file data. However, symmetric encryption algorithms have a weakness: you can bypass their security if you obtain the key. If multiple users want to share one encrypted file protected only by DESX, AES, or 3DES, each user would require access to the file's FEK. Leaving the FEK unencrypted would obviously be a security problem, but encrypting the FEK once would require all the users to share the same FEK decryption key—another potential security problem.

Keeping the FEK secure is a difficult problem, which EFS addresses with the public key–based half of its encryption architecture. Encrypting a file's FEK for individual users who access the file lets multiple users share an encrypted file. EFS can encrypt a file's FEK with each user's public key and can store each user's encrypted FEK with the file. Anyone can access a user's public key, but no one can use a public key to decrypt the data that the public key encrypted. The only way users can decrypt a file is with their private key, which the operating system must access. A user's private key decrypts the user's encrypted copy of a file's FEK. Public key–based algorithms are usually slow, but EFS uses these algorithms only to encrypt FEKs. Splitting key management between a publicly available key and a private key makes key management a little easier than symmetric encryption algorithms do and solves the dilemma of keeping the FEK secure.

Windows stores a user's private keys in their user profile directory (typically under \Documents and Settings) within the Application Data\Microsoft\Crypto\RSA subdirectory. To protect private keys, Windows encrypts all files within the RSA folder with a random symmetric key called the user's master key. The master key is 64-bytes in length and is generated by a strong random number generator. The master key is also stored in the user's profile under the Application Data\Microsoft\Protect directory and is 3DES-encrypted with a key that's in part based on the user's password. When a user changes his or her password, master keys are automatically unencrypted and re-encrypted using the new password.

Several components work together to make EFS work, as the diagram of EFS architecture in Figure 12-57 shows. On Windows 2000, EFS is implemented as a device driver that runs in kernel mode and is tightly connected with the NTFS file system driver, but on Windows XP and Windows Server 2003 EFS support is merged into the NTFS driver. Whenever NTFS encounters an encrypted file, NTFS executes EFS functions that the EFS kernel-mode code registered with NTFS when the EFS kernel-mode code initialized. The EFS functions encrypt and decrypt file data as applications access encrypted files. Although EFS stores an FEK with a file's data, users' public keys encrypt the FEK. To encrypt or decrypt file data, EFS must decrypt the file's FEK with the aid of cryptography services that reside in user mode.

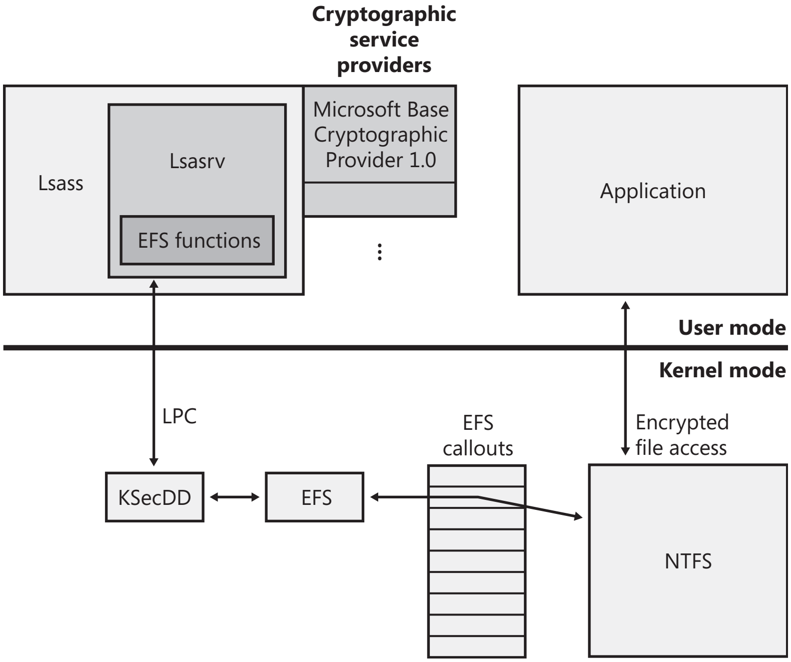


Figure 12-57 EFS architecture

The Local Security Authority Subsystem (LSASS - \Windows\System32\LSASS.exe) manages logon sessions but also handles EFS key management chores. For example, when the EFS driver needs to decrypt an FEK to decrypt file data a user wants to access, the EFS driver sends a request to LSASS. EFS sends the request via a local procedure call (LPC). The KSecDD (\Windows\System32\Drivers\Ksecdd.sys) device driver exports functions for other drivers that need to send LPC messages to LSASS. The Local Security Authority Server (Lsasrv - \Windows\System32\Lsasrv.dll) component of LSASS that listens for remote procedure call (RPC) requests passes requests to decrypt an FEK to the appropriate EFS-related decryption function, which also resides in Lsasrv. Lsasrv uses functions in Microsoft CryptoAPI (also referred to as CAPI) to decrypt the FEK, which the EFS driver sent to LSASS in encrypted form.

CryptoAPI comprises cryptographic service provider (CSP) DLLs that make various cryptography services (such as encryption/decryption and hashing) available to applications. The CSP DLLs manage retrieval of user private and public keys, for example, so that Lsasrv doesn't need to concern itself with the details of how keys are protected or even with the details of the encryption algorithms. EFS uses the RSA encryption algorithms provided by the Microsoft Enhanced Cryptographic provider (\Windows\System32\Rsaenh.dll). After Lsasrv decrypts an FEK, Lsasrv returns the FEK to the EFS driver via an LPC reply message. After EFS receives the decrypted FEK, EFS can use DESX to decrypt the file data for NTFS. Let's look at the details of how EFS integrates with NTFS and how Lsasrv uses CryptoAPI to manage keys.

Encrypting a File for the First Time

The NTFS driver calls EFS functions when it encounters an encrypted file. A file's attributes record that the file is encrypted in the same way that a file records that it is compressed (discussed earlier in this chapter). NTFS and EFS have specific interfaces for converting a file from nonencrypted to encrypted form, but user-mode components primarily drive the process. As described earlier, Windows lets you encrypt a file in two ways: by using the *cipher* command-line utility or by checking the Encrypt Contents To Secure Data check box in the Advanced Attributes dialog box for a file in Windows Explorer. Both Windows Explorer and the *cipher* command rely on the *EncryptFile* Windows API that the Advapi32.dll (Advanced Windows APIs DLL) exports. Advapi32 loads another DLL, Feclient.dll (File Encryption Client DLL), to obtain APIs that Advapi32 can use to invoke EFS interfaces in Lsasrv via LPCs.

When Lsasrv receives an RPC message from Feclient to encrypt a file, Lsasrv uses the Windows impersonation facility to impersonate the user that ran the application (either *cipher* or Windows Explorer) that is encrypting the file. (Impersonation is described in Chapter 8.) This procedure lets Windows treat the file operations that Lsasrv performs as if the user who wants to encrypt the file is performing them. Lsasrv usually runs in the System account. (The System account is described in Chapter 8.) In fact, if it doesn't impersonate the user, Lsasrv usually won't have permission to access the file in question.

Lsasrv next creates a log file in the volume's System Volume Information directory into which Lsasrv records the progress of the encryption process. The log file usually has the name Efs0.log, but if other files are undergoing encryption, increasing numbers replace the 0 until a unique log file name for the current encryption is created.

CryptoAPI relies on information that a user's registry profile stores, so if the profile is not already loaded, Lsasrv next uses the *LoadUserProfile* API function of Userenv.dll (User Environment DLL) to load the profile into the registry of the user it is impersonating. Typically, the user profile is already loaded because Winlogon loads a user's profile when a user logs on. However, if a user uses the Windows *RunAs* command to log on to a different account, when the user tries to access encrypted files from that account, the account's profile might not load.

Lsasrv then generates an FEK for the file by using the RSA encryption facilities of the Microsoft Base Cryptographic Provider 1.0 CSP.

Constructing Key Rings

At this point, Lsasrv has an FEK and can construct EFS information to store with the file, including an encrypted version of the FEK. Lsasrv reads the HKEY_ CURRENT_USER\Software\Microsoft\Windows NT\CurrentVersion\EFS\CurrentKeys\CertificateHash value of the user performing the encryption to obtain the user's public key signature. (Note that this key doesn't appear in the registry until a file or folder is encrypted.) Lsasrv uses the signature to access the user's public key and encrypt FEKs.

Lsasrv can now construct the information that EFS stores with the file. EFS stores only one block of information in an encrypted file, and that block contains an entry for each user sharing the file. These entries are called *key entries*, and EFS stores them in the Data Decryption Field (DDF) portion of the file's EFS data. A collection of multiple key entries is called a *key ring* because, as mentioned earlier, EFS lets multiple users share encrypted files.

Figure 12-58 shows a file's EFS information format and key entry format. EFS stores enough information in the first part of a key entry to precisely describe a user's public key. This data includes the user's security ID (SID) (note that the SID is not guaranteed to be present), the container name in which the key is stored, the cryptographic provider name, and the private/public key pair certificate hash. Only the private/public key pair certificate hash is used by the description process. The second part of the key entry contains an encrypted version of the FEK. Lsasrv uses the CryptoAPI to encrypt the FEK with the RSA algorithm and the user's public key.

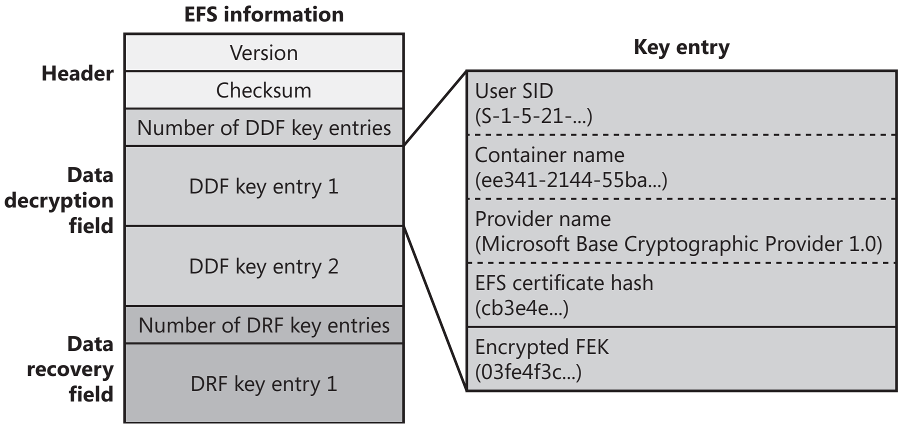
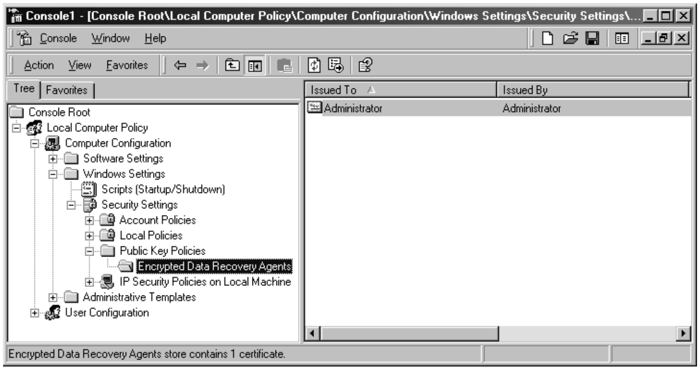


Figure 12-58 Format of EFS information and key entries

Next, Lsasrv creates another key ring that contains recovery key entries. EFS stores information about recovery key entries in a file's Data Recovery Field (DRF). The format of DRF entries is identical to the format of DDF entries. The DRF's purpose is to let designated accounts, or Recovery Agents, decrypt a user's file when administrative authority must have access to the user's data. For example, suppose a company employee forgot his or her logon password. An administrator can reset the user's password, but without Recovery Agents, no one can recover the user's encrypted data.

Recovery Agents are defined with the Encrypted Data Recovery Agents security policy of the local computer or domain. This policy is available from the Group Policy MMC snap-in, as shown in Figure 12-59. When you use the Recovery Agent Wizard (by right-clicking on Encrypted Data Recovery Agents and selecting Encrypted Recovery Agent from the New option), you can add Recovery Agents and specify which private/public key pairs (designated by their certificates) the Recovery Agents use for EFS recovery. Lsasrv interprets the recovery policy when it initializes and when it receives notification that the recovery policy has changed. EFS creates a DRF key entry for each Recovery Agent by using the cryptographic provider registered for EFS recovery. The default Recovery Agent provider is the RSA encryption facility of Base Cryptographic Provider 1.0—the same provider Lsasrv uses for user keys.



Issued To	Issued By
Administrator	Administrator

Encrypted Data Recovery Agents store contains 1 certificate.

Figure 12-59 Encrypted Data Recovery Agents group policy

In the final step in creating EFS information for a file, Lsasrv calculates a checksum for the DDF and DRF by using the MD5 hash facility of Base Cryptographic Provider 1.0. Lsasrv stores the checksum's result in the EFS information header. EFS references this checksum during decryption to ensure that the contents of a file's EFS information haven't become corrupted or been tampered with.

Encrypting File Data

Figure 12-60 illustrates the flow of the encryption process. After Lsasrv constructs the necessary information for a file a user wants to encrypt, it can begin encrypting the file. Lsasrv creates a backup file, Efs0.tmp, for the file undergoing encryption. (Lsasrv uses higher numbers in the backup filename if other backup files exist.) Lsasrv creates the backup file in the directory that contains the file undergoing encryption. Lsasrv applies a restrictive security descriptor to the backup file so that only the System account can access the file's contents. Lsasrv next initializes the log file that it created in the first phase of the encryption process. Finally, Lsasrv records in the log file that the backup file has been created. Lsasrv encrypts the original file only after the file is completely backed up.

Lsasrv next sends the EFS kernel-mode code, through NTFS, a command to add to the original file the EFS information that it just created. On Windows 2000, NTFS receives this command, but because NTFS doesn't understand EFS commands, NTFS calls the EFS driver. The EFS kernel-mode code takes the EFS information that Lsasrv sent and uses exported NTFS functions to apply the information to the file. The exported NTFS functions let EFS add the $EFS attribute to NTFS files. Execution returns to Lsasrv, which copies the contents of the file undergoing encryption to the backup file. When the backup copy is complete, including backups of all alternate data streams, Lsasrv records in the log file that the backup file is up to date. Lsasrv then sends another command to NTFS to tell NTFS to encrypt the contents of the original file.

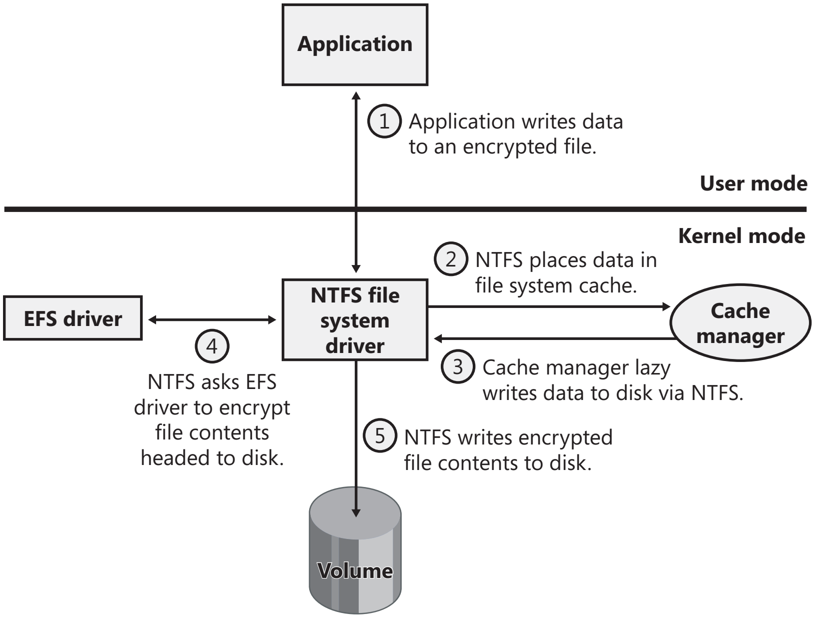

Application
1 Application writes data to an encrypted file.
User mode
Kernel mode
2 NTFS places data in file system cache.
EFS driver
NTFS file system driver
Cache manager
4
NTFS asks EFS driver to encrypt file contents headed to disk.
3 Cache manager lazy writes data to disk via NTFS.
5 NTFS writes encrypted file contents to disk.
Volume

Figure 12-60 Flow of EFS

When NTFS receives the EFS command to encrypt the file, NTFS deletes the contents of the original file and copies the backup data to the file. After NTFS copies each section of the file, NTFS flushes the section's data from the file system cache, which prompts the cache manager to tell NTFS to write the file's data to disk. Because the file is marked as encrypted, at this point in the file-writing process, NTFS calls EFS to encrypt the data before NTFS writes the data to disk. EFS uses the unencrypted FEK that NTFS passes it to perform DESX, AES, or 3DES encryption of the file as appropriate, one sector (512 bytes) at a time.

After EFS encrypts the file, Lsasrv records in the log file that the encryption was successful and deletes the file's backup copy. Finally, Lsasrv deletes the log file and returns control to the application that requested the file's encryption.

Encryption Process Summary

The following list summarizes the steps EFS performs to encrypt a file:

1. The user profile is loaded if necessary.
2. A log file is created in the System Volume Information directory with the name Efs*x*.log, where *x* is a unique number (for example, Efs0.log). As subsequent steps are performed, records are written to the log so that the file can be recovered in case the system fails during the encryption process.
3. Base Cryptographic Provider 1.0 generates a random 128-bit FEK for the file.
4. A user EFS private/public key pair is generated or obtained. HKEY_CURRENT_USER\Software\Microsoft\Windows NT\CurrentVersion\EFS\CurrentKeys\CertificateHash identifies the user's key pair.
5. A DDF key ring is created for the file that has an entry for the user. The entry contains a copy of the FEK that has been encrypted with the user's EFS public key.
6. A DRF key ring is created for the file. It has an entry for each Recovery Agent on the system, with each entry containing a copy of the FEK encrypted with the agent's EFS public key.
7. A backup file with a name in the form Efs0.tmp is created in the same directory as the file to be encrypted
8. The DDF and DRF key rings are added to a header and augment the file as its EFS attribute.
9. The backup file is marked as encrypted, and the original file is copied to the backup.
10. The original file's contents are destroyed, and the backup is copied to the original. This copy operation results in the data in the original file being encrypted because the file is now marked as encrypted.
11. The backup file is deleted.
12. The log file is deleted.
13. The user profile is unloaded (if it was loaded in step 1).

If the system crashes during the encryption process, either the original file remains intact or the backup file contains a consistent copy. When Lsasrv initializes after a system crash, it looks for log files under the System Volume Information subdirectory on each NTFS volume on the system. If Lsasrv finds one or more log files, it examines their contents and determines how recovery should take place. Lsasrv deletes the log file and the corresponding backup file if the original file wasn't modified at the time of the crash; otherwise, Lsasrv copies the backup file over the original, partially encrypted file and then deletes the log and backup. After Lsasrv processes log files, the file system will be in a consistent state with respect to encryption, with no loss of user data.

The Decryption Process

The decryption process begins when a user opens an encrypted file. NTFS examines the file's attributes when opening the file and then executes a callback function in the EFS driver. The EFS driver reads the $EFS attribute associated with the encrypted file. To read the attribute, the driver calls EFS support functions that NTFS exports for EFS's use. NTFS completes the necessary steps to open the file. The EFS driver ensures that the user opening the file has access privileges to the file's encrypted data (that is, that an encrypted FEK in either the DDF or DRF key rings corresponds to a private/public key pair associated with the user). As EFS performs this validation, EFS obtains the file's decrypted FEK to use in subsequent data operations the user might perform on the file.

EFS can't decrypt an FEK and relies on Lsasrv (which can use CryptoAPI) to perform FEK decryption. EFS sends an LPC message by way of the Ksecdd.sys driver to Lsasrv that asks Lsasrv to obtain the decrypted form of the encrypted FEK in the $EFS attribute data (the EFS data) that corresponds to the user who is opening the file.

When Lsasrv receives the LPC message, Lsasrv executes the Userenv.dll (User Environment DLL) *LoadUserProfile* API function to bring the user's profile into the registry, if the profile isn't already loaded. Lsasrv proceeds through each key field in the EFS data, using the user's private key to try to decrypt each FEK. For each key, Lsasrv attempts to decrypt a DDF or DRF key entry's FEK. If the certificate hash in a key field doesn't refer to a key the user owns, Lsasrv moves on to the next key field. If Lsasrv can't decrypt any DDF or DRF key field's FEK, the user can't obtain the file's FEK. Consequently, EFS denies access to the application opening the file. However, if Lsasrv identifies a hash as corresponding to a key the user owns, it decrypts the FEK with the user's private key using CryptoAPI.

Because Lsasrv processes both DDF and DRF key rings when decrypting an FEK, it automatically performs file recovery operations. If a Recovery Agent that isn't registered to access an encrypted file (that is, it doesn't have a corresponding field in the DDF key ring) tries to access a file, EFS will let the Recovery Agent gain access because the agent has access to a key pair for a key field in the DRF key ring.

Decrypted FEK Caching

Traveling the path from the EFS driver to Lsasrv and back can take a relatively long time—in the process of decrypting an FEK, CryptoAPI uses results in more than 2000 registry API calls and 400 file system accesses on a typical system. The EFS driver, with the aid of NTFS, uses a cache to try to avoid this expense.

Decrypting File Data

After an application opens an encrypted file, the application can read from and write to the file. NTFS calls the EFS driver to decrypt file data as NTFS reads the data from the disk and before NTFS places the data in the file system cache. Similarly, when an application writes data to a file, the data remains in unencrypted form in the file system cache until the application or the cache manager uses NTFS to flush the data back to disk. When an encrypted file's data writes back from the cache to the disk, NTFS calls the EFS driver to encrypt the data.

As stated earlier, the EFS driver performs encryption and decryption in 512-byte units. The 512-byte size is the most convenient for the driver because disk reads and writes occur in multiples of the 512-byte sector.

Backing Up Encrypted Files

An important aspect of any file encryption facility's design is that file data is never available in unencrypted form except to applications that access the file via the encryption facility. This restriction particularly affects backup utilities, in which archival media store files. EFS addresses this problem by providing a facility for backup utilities so that the utilities can back up and restore files in their encrypted states. Thus, backup utilities don't have to be able to decrypt file data, nor do they need to decrypt file data in their backup procedures.

Backup utilities use the EFS API functions *OpenEncryptedFileRaw*, *ReadEncryptedFileRaw*, *WriteEncryptedFileRaw*, and *CloseEncryptedFileRaw* in Windows to access a file's encrypted contents. The Advapi32.dll library provides these API functions, which all use LPCs to invoke corresponding functions in Lsasrv. For example, after a backup utility opens a file for raw access during a backup operation, the utility calls *ReadEncryptedFileRaw* to obtain the file data. The Lsasrv function *EfsReadFileRaw* issues control commands (which the EFS session key encrypts with DESX, AES, or 3DES) to the NTFS driver to read the file's EFS attribute first and then the encrypted contents.

EfsReadFileRaw might have to perform multiple read operations to read a large file. As *EfsReadFileRaw* reads each portion of such a file, Lsasrv sends an RPC message to Advapi32.dll that executes a callback function that the backup program specified when it issued the *ReadEncryptedFileRaw* API function. *EfsReadFileRaw* hands the encrypted data it just read to the callback function, which can write the data to the backup media. Backup utilities restore encrypted files in a similar manner. The utilities call the *WriteEncryptedFileRaw* API function, which invokes a callback function in the backup program to obtain the unencrypted data from the backup media while Lsasrv's *EfsWriteFileRaw* function is restoring the file's contents.

EXPERIMENT: Viewing EFS Information

EFS has a handful of other API functions that applications can use to manipulate encrypted files. For example, applications use the *AddUsersToEncryptedFile* API function to give additional users access to an encrypted file and *RemoveUsersFromEncryptedFile* to revoke users' access to an encrypted file. Applications use the *QueryUsersOnEncryptedFile* function to obtain information about a file's associated DDF and DRF key fields. *QueryUsersOnEncryptedFile* returns the SID, certificate hash value, and display information that each DDF and DRF key field contains. The following output is from the EFSDump utility, from *www.sysinternals.com*, when an encrypted file is specified as a command-line argument:

```
C:\>efsdump test.txt
EFS Information Dumper v1.02
Copyright (C) 1999 Mark Russinovich
Systems Internals - http://www.sysinternals.com

test.txt:
DDF Entry:
    DARYL\Mark:
         CN=Mark,L=EFS,OU=EFS File Encryption Certificate
DRF Entry:
    DARYL\Administrator
         OU=EFS File Encryption Certificate, L=EFS, CN=Administrator
```

You can see that the file test.txt has one DDF entry for user Mark and one DRF entry for Administrator, which is the only Recovery Agent currently registered on the system.

You can use the Nfi utility from the OEM Support Tools to see the presence of the $EFS attribute in an encrypted file:

```
C:\>nfi test.txt
NTFS File Sector Information Utility.Copyright (C) Microsoft Corporation 1999. All rig
hts reserved.
\Temp\README.TXT    $STANDARD_INFORMATION (resident)    $FILE_NAME (resident)    $DATA
 (nonresident)        logical sectors 2728124-2728135 (0x29a0bc-
0x29a0c7)    Attribute Type 0x100 $EFS (nonresident)       logical sectors 2156984-
2156987 (0x20e9b8-0x20e9bb)
```

Conclusion

Windows support a wide variety of file system formats accessible to both the local system and remote clients. The file system filter driver architecture provides a clean way to extend and augment file system access. And NTFS provides a reliable, secure, scalable file system format for local file system storage. In the next chapter, we'll look at networking on Windows.

Chapter 13

Networking

Microsoft Windows was designed with networking in mind, and it includes broad networking support that is integrated with the I/O system and the Windows API. The four basic types of networking software are services, APIs, protocols, and network adapter device drivers, and each is layered on the next to form a network stack. Windows has well-defined interfaces for each layer, so in addition to using the wide variety of APIs, protocols, and adapter device drivers that ship with Windows, third parties can extend the operating system's networking capabilities by developing their own.

In this chapter, we take you from the top of the Windows networking stack to the bottom. First, we present the mapping between the Windows networking software components and the Open Systems Interconnection (OSI) reference model. Then we briefly describe the networking APIs available on Windows and explain how they are implemented. You'll learn how multiple redirector support and name resolution works and how protocol drivers are implemented. After looking at the implementation of network adapter device drivers, we examine *binding*, which is the glue that connects services, protocol stacks, and network adapters.

Windows Networking Architecture

The goal of network software is to take a request (usually an I/O request) from an application on one machine, pass it to another machine, execute the request on the remote machine, and return the results to the first machine. In the course of this process, the request must be transformed several times along the way. A high-level request, such as "read *x* number of bytes from file *y* on machine *z*," requires software that can determine how to get to machine *z* and what communication software that machine understands. Then the request must be altered for transmission across a network—for example, divided into short packets of information. When the request reaches the other side, it must be checked for completeness, decoded, and sent to the correct operating system component for execution. Finally, the reply must be encoded for sending back across the network.

The OSI Reference Model

To help different computer manufacturers standardize and integrate their networking software, in 1983 the International Organization for Standardization (ISO) defined a software model for sending messages between machines. The result is the *Open Systems Interconnection* (OSI) *reference model*. The model defines seven layers of software, as shown in Figure 13-1.

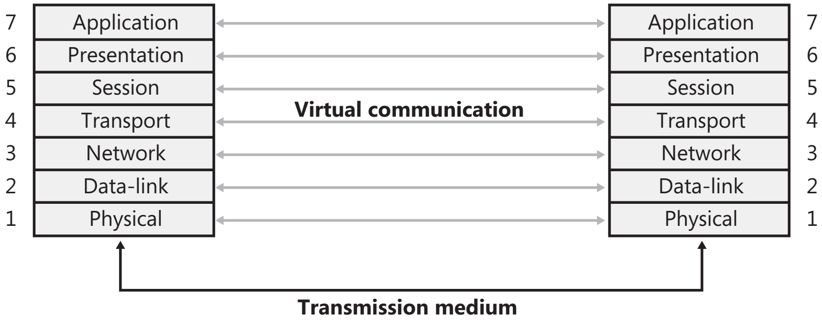


Figure 13-1 OSI reference model

The OSI reference model is an idealized scheme that few systems implement precisely, but it's often used to frame discussions of networking principles. Each layer on one machine assumes that it is "talking to" the same layer on the other machine. Both machines "speak" the same language, or protocol, at the same level. In reality, however, a network transmission must pass down each layer on the client machine, be transmitted across the network, and then pass up the layers on the destination machine until it reaches a layer that can understand and implement the request.

The purpose of each layer in the OSI model is to provide services to higher layers and to abstract how the services are implemented at lower layers. Detailing the purpose of each layer is beyond the scope of this book, but here are some brief descriptions of the various layers:

- **Application layer** Handles information transfer between two network applications, including functions such as security checks, identification of the participating machines, and initiation of the data exchange.
- **Presentation layer** Handles data formatting, including issues such as whether lines end in carriage return/line feed (CR/LF) or just carriage return (CR), whether data is to be compressed or encrypted, and so forth.
- **Session layer** Manages the connection between cooperating applications, including high-level synchronization and monitoring of which application is "talking" and which is "listening."
- **Transport layer** On the client, this layer divides messages into packets and assigns them sequence numbers to ensure that they are all received in the proper order. On the destination, it assembles packets that have been received. It also shields the session layer from the effects of changes in hardware.
- **Network layer** Creates packet headers and handles routing, congestion control, and internetworking. It is the highest layer that understands the network's *topology*—that is, the physical configuration of the machines in the network, any limitations in bandwidth, and so on.

- **Data-link layer** Transmits low-level data frames, waits for acknowledgment that they were received, and retransmits frames that were lost over unreliable lines.
- **Physical layer** Passes bits to the network cable or other physical transmission medium.

The dashed lines in Figure 13-1 represent protocols used in transmitting a request to a remote machine. As stated earlier, each layer of the hierarchy assumes that it is speaking to the same layer on another machine and uses a common protocol. The collection of protocols through which a request passes on its way down and back up the layers of the network is called a *protocol stack*.

Windows Networking Components

Figure 13-2 provides an overview of the components of Windows networking, showing how each component fits into the OSI reference model and which protocols are used between layers. The mapping between OSI layers and networking components isn't precise, which is the reason that some components cross layers. The various components include the following:

- *Networking APIs* provide a protocol-independent way for applications to communicate across a network. Networking APIs can be implemented in user mode or in both user mode and kernel mode, and in some cases are wrappers around another networking API that implements a specific programming model or provides additional services. (Note that the term *networking API* also describes any programming interfaces provided by networking-related software.)
- *Transport Driver Interface* (TDI) *clients* are kernel-mode device drivers that usually implement the kernel-mode portion of a networking API's implementation. TDI clients get their name from the fact that the I/O request packets (IRPs) they send to protocol drivers are formatted according to the Windows Transport Driver Interface standard (documented in the DDK). This standard specifies a common programming interface for kernel-mode device drivers. (See Chapter 9 for more information about IRPs.)
- *TDI transports*, also known as *transports*, *Network Driver Interface Specification* (NDIS) *protocol drivers*, and *protocol drivers*, are kernel-mode protocol drivers. They accept IRPs from TDI clients and process the requests these IRPs represent. This processing might require network communications with a peer, prompting the TDI transport to add protocol-specific headers (such as TCP, UDP, or IP) to data passed in the IRP and to communicate with adapter drivers using NDIS functions (also documented in the DDK). TDI transports generally facilitate application network communications by transparently performing message operations such as segmentation and reassembly, sequencing, acknowledgment, and retransmission.
- The *NDIS library* (Ndis.sys) provides encapsulation for adapter drivers, hiding from them specifics of the Windows kernel-mode environment. The NDIS library exports functions for use by TDI transports as well as support functions for adapter drivers.

- *NDIS miniport drivers* are kernel-mode drivers that are responsible for interfacing TDI transports to particular network adapters. NDIS miniport drivers are written so that they are wrapped by the Windows NDIS library. The encapsulation provides cross-platform compatibility with Microsoft Consumer Windows. NDIS miniport drivers don't process IRPs; rather, they register a call-table interface to the NDIS library that contains pointers to functions corresponding to ones that the NDIS library exports to TDI transports. NDIS miniport drivers communicate with network adapters by using NDIS library functions that resolve to hardware abstraction layer (HAL) functions.

As the figure shows, the OSI layers don't correspond to actual software. TDI transports, for example, frequently cross several boundaries. In fact, the bottom four layers of software are often referred to collectively as "the transport." Software components residing in the upper three layers are referred to as "users of the transport."

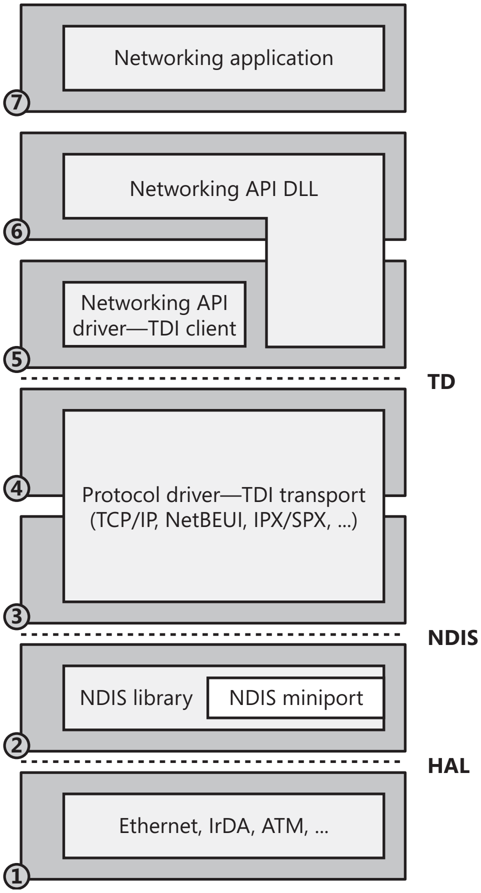


Figure 13-2 OSI model and Windows networking components

In the remainder of this chapter, we'll examine the networking components shown in Figure 13-2 (as well as others not shown in the figure), looking at how they fit together and how they relate to Windows as a whole.

Networking APIs

Windows implements multiple networking APIs to provide support for legacy applications and compatibility with industry standards. In this section, we'll briefly look at the networking APIs and describe how applications use them. It's important to keep in mind that the decision about which API an application uses depends on characteristics of the API, such as which protocols the API can layer over, whether the API supports reliable or bidirectional communication, and the API's portability to other Windows platforms the application might run on. We'll discuss the following networking APIs:

- Windows Sockets (Winsock)
- Remote procedure call (RPC)
- Web access APIs
- Named pipes and mailslots
- NetBIOS
- Other networking APIs

Windows Sockets

The original Windows Sockets (Winsock) (version 1.0) was Microsoft's implementation of BSD (Berkeley Software Distribution) Sockets, a programming API that became the standard by which UNIX systems have communicated over the Internet since the 1980s. Support for sockets on Windows makes the task of porting UNIX networking applications to Windows relatively straightforward. The modern versions of Winsock include most of the functionality of BSD Sockets but also include Microsoft-specific enhancements, which continue to evolve. Winsock supports reliable-connection-oriented communication as well as unreliable-connectionless communication. Windows provides Winsock 2.2, which is also either included with or available as an add-on for all current versions of Windows. Winsock 2.2 adds numerous features beyond the BSD Sockets specification, such as adding functions that take advantage of Windows asynchronous I/O that offer far better performance and scalability over straight BSD Socket programming.

Winsock includes the following features:

- Support for scatter-gather and asynchronous application I/O.
- Quality of service (QoS) conventions so that applications can negotiate latency and bandwidth requirements when the underlying network supports QoS.

- Extensibility so that Winsock can be used with protocols other than those Windows requires it to support.
- Support for integrated namespaces other than those defined by a protocol an application is using with Winsock. A server can publish its name in Active Directory, for example, and using namespace extensions, a client can look up the server's address in Active Directory.
- Support for multipoint messages where messages transmit from a single source to multiple receivers simultaneously.

We'll examine typical Winsock operation and then describe ways that Winsock can be extended.

Winsock Client Operation

The first step a Winsock application takes is to initialize the Winsock API with a call to an initialization function. On a Windows 2000 system, a Winsock application's next step is to create a *socket* that will represent a communications endpoint. The application obtains the address of the server to which it wants to connect by calling *gethostbyname*. Winsock is a protocol-independent API, so an address can be specified for any protocol installed on the system over which Winsock operates (TCP/IP, TCP/IP with IP version 6, IPX). After obtaining the server address, a connection-oriented client attempts to connect to the server by using *connect* and specifying the server address.

On Windows XP and Windows Server 2003, an application should use *getaddrinfo* to obtain a server's address instead of using *gethostbyname*. *getaddrinfo* returns the list of addresses assigned to the server, and the client attempts to connect to each one in turn until it is able to establish a connection with one of them. This ensures that a client that only supports IP version 4 (IPv4), for instance, connects to the appropriate IPv4 address on a server that might have both IPv4 and IP version 6 (IPv6) addresses assigned to it.

When a connection is established, the client can send and receive data over its socket using *recv* and *send*, for example. A connectionless client specifies the remote address with connectionless APIs, such as the connectionless equivalents of *send* and *recv*, and *sendto* and *recvfrom*.

Winsock Server Operation

The sequence of steps for a server application differs from that of a client. After initializing the Winsock API, the server creates a socket and then binds it to a local address by using *bind*. Again, the address type specified—whether TCP/IP, TCP/IP with IP version 6, or some other address type—is up to the server application.

If the server is connection-oriented, it performs a *listen* operation on the socket, indicating the number of connections that it can support for the socket. Then it performs an *accept* operation to allow a client to connect to the socket. If there is a pending connection request, the accept call completes immediately; otherwise, it completes when a connection request arrives. When a connection is made, the *accept* function returns a new socket that represents the server's end of the connection. The server can perform receive and send operations by using functions such as *recv* and *send*. Figure 13-3 shows connection-oriented communication between a Winsock client and server.

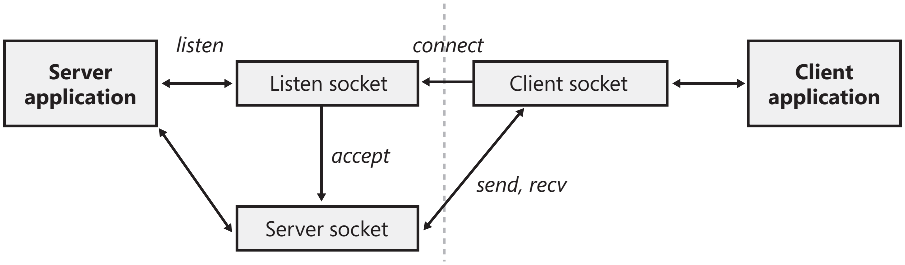


Figure 13-3 Connection-oriented Winsock operation

After binding an address, a connectionless server is no different from a connectionless client: it can send and receive data over the socket simply by specifying the remote address with each operation. Most connectionless protocols are unreliable and in general will not know whether the destination actually received the sent data packets, known as *datagrams*. Datagram protocols are ideal for quick message passing, where the overhead of establishing a connection is not too much and reliability is not required (although an application can build reliability on top of the protocol).

Winsock Extensions

A powerful feature from a Windows programming point of view is that the Winsock API is integrated with Windows messages. A Winsock application can take advantage of this feature to perform asynchronous socket operations and receive notification of an operation's completion via a standard Windows message or through the execution of a callback function. This capability simplifies the design of a Windows application because the application doesn't need to be multithreaded or manage synchronization objects to both perform network I/O and respond to user input or requests from the window manager to update the application windows.

In addition to supporting functions that correspond directly to those implemented in BSD Sockets, Microsoft has added a handful of functions that aren't part of the Winsock standard. Two of these functions, *AcceptEx* and *TransmitFile*, are worth describing because many Web servers on Windows use them to achieve high performance. *AcceptEx* is a version of the *accept* function that, in the process of establishing a connection with a client, returns the client's address and the client's first message. *AcceptEx* allows the server application to prepost multiple accept operations so that high volumes of incoming connection requests can be handled. With this function, a Web server avoids executing multiple Winsock functions that would otherwise be required.

After establishing a connection with a client, a Web server usually sends a file, such as a Web page, to the client. The *TransmitFile* function's implementation is integrated with the Windows cache manager so that a client can send a file directly from the file system cache. Sending data in this way is called *zero-copy* because the server doesn't have to touch the file data to send it; it simply specifies a handle to a file and the ranges of the file to send. In addition, *TransmitFile* allows a server to prepend or append data to the file's data so that the server can send header information, which might include the name of the Web server and a field that indicates to the client the size of the message the server is sending. Internet Information Services (IIS), which is bundled with Windows, uses both *AcceptEx* and *TransmitFile*.

Windows XP and Windows Server 2003 add a handful of other, multifunction APIs, including *ConnectEx*, *DisconnectEx*, and *TransmitPackets*. *ConnectEx* establishes a connection and sends the first message on the connection. *DisconnectEx* closes a connection and allows the socket handle representing the connection to be reused in a call to *AcceptEx* or *ConnectEx*. Finally, *TransmitPackets* is similar to *TransmitFile*, except that it allows for the sending of in-memory data in addition to or in lieu of file data.

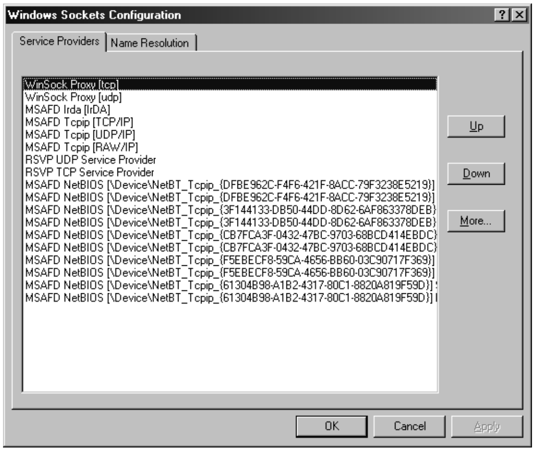
Extending Winsock

Winsock is an extensible API on Windows because third parties can add a *transport service provider* that interfaces Winsock with other protocols or layers on top of existing protocols to provide functionality such as proxying. Third parties can also add a *namespace service provider* to augment Winsock's name-resolution facilities. Service providers plug in to Winsock using the Winsock *service provider interface* (SPI). When a transport service provider is registered with Winsock, Winsock uses the transport service provider to implement socket functions, such as *connect* and *accept*, for the address types that the provider indicates it implements. There are no restrictions on how the transport service provider implements the functions, but the implementation usually involves communicating with a transport driver in kernel mode.

A requirement of any Winsock client/server application is for the server to make its address available to clients so that the clients can connect to the server. Standard services that execute on the TCP/IP protocol use "well-known addresses" to make their addresses available. As long as a browser knows the name of the computer a Web server is running on, it can connect to the Web server by specifying the well-known Web server address (the IP address of the server concatenated with :80, the port number used for HTTP). Namespace service providers make it possible for servers to register their presence in other ways. For example, one namespace service provider might on the server side register the server's address in Active Directory, and on the client side look up the server's address in Active Directory. Namespace service providers supply this functionality to Winsock by implementing standard Winsock name-resolution functions such as *getaddrinfo* (a replacement for *gethostbyname*) and *getnameinfo*.

EXPERIMENT: Looking at Winsock Service Providers

The Windows Sockets Configuration utility (Sporder.exe) included with the Platform SDK shows the registered Winsock transport and namespace providers and allows you to change the order in which transport service providers are enumerated. For example, if there are two TCP/IP transport service providers, the first one listed is the default provider for Winsock applications using the TCP/IP protocol. Here's a screen shot from Sporder showing the registered transport service providers:



```
WinSock Proxy [tcp]
WinSock Proxy [udp]
MSAFD Irda [IrDA]
MSAFD Tcpip [TCP/IP]
MSAFD Tcpip [UDP/IP]
MSAFD Tcpip [RAW/IP]
RSVP UDP Service Provider
RSVP TCP Service Provider
MSAFD NetBIOS [\Device\NetBT_Tcpip_{DFBE962C-F4F6-421F-8ACC-79F3238E5219}]
MSAFD NetBIOS [\Device\NetBT_Tcpip_{DFBE962C-F4F6-421F-8ACC-79F3238E5219}]
MSAFD NetBIOS [\Device\NetBT_Tcpip_{3F144133-DB50-44DD-8D62-6AF863378DEB}
MSAFD NetBIOS [\Device\NetBT_Tcpip_{3F144133-DB50-44DD-8D62-6AF863378DEB}
MSAFD NetBIOS [\Device\NetBT_Tcpip_{CB7FCA3F-0432-47BC-9703-68BCD414EBDC}
MSAFD NetBIOS [\Device\NetBT_Tcpip_{CB7FCA3F-0432-47BC-9703-68BCD414EBDC}
MSAFD NetBIOS [\Device\NetBT_Tcpip_{F5EBECF8-59CA-4656-BB60-03C90717F369}]
MSAFD NetBIOS [\Device\NetBT_Tcpip_{F5EBECF8-59CA-4656-BB60-03C90717F369}]
MSAFD NetBIOS [\Device\NetBT_Tcpip_{61304B98-A1B2-4317-80C1-8820A819F59D}]
MSAFD NetBIOS [\Device\NetBT_Tcpip_{61304B98-A1B2-4317-80C1-8820A819F59D}]
```

Winsock Implementation

Winsock's implementation is shown in Figure 13-4. Its application interface consists of an API DLL, Ws2_32.dll (\Windows\System32\Ws2_32.dll), which provides applications access to Winsock functions. Ws2_32.dll calls on the services of namespace and transport service providers to carry out name and message operations. The Mswsock.dll library acts as a transport service provider for the protocols Microsoft provides support for in Winsock, and uses *Winsock Helper* libraries that are protocol specific to communicate with kernel-mode protocol drivers. For example, Wshtcpip.dll is the TCP/IP helper. Mswsock.dll (\Windows\System32\Mswsock.dll) implements the Microsoft Winsock extension functions, such as *TransmitFile*, *AcceptEx*, and *WSARecvEx*. Windows ships with helper DLLs for TCP/IP, TCP/IP with IPv6, AppleTalk, IPX/SPX, ATM, and IrDA (Infrared Data Association) and namespace service providers for DNS (TCP/IP), Active Directory, and IPX/SPX.

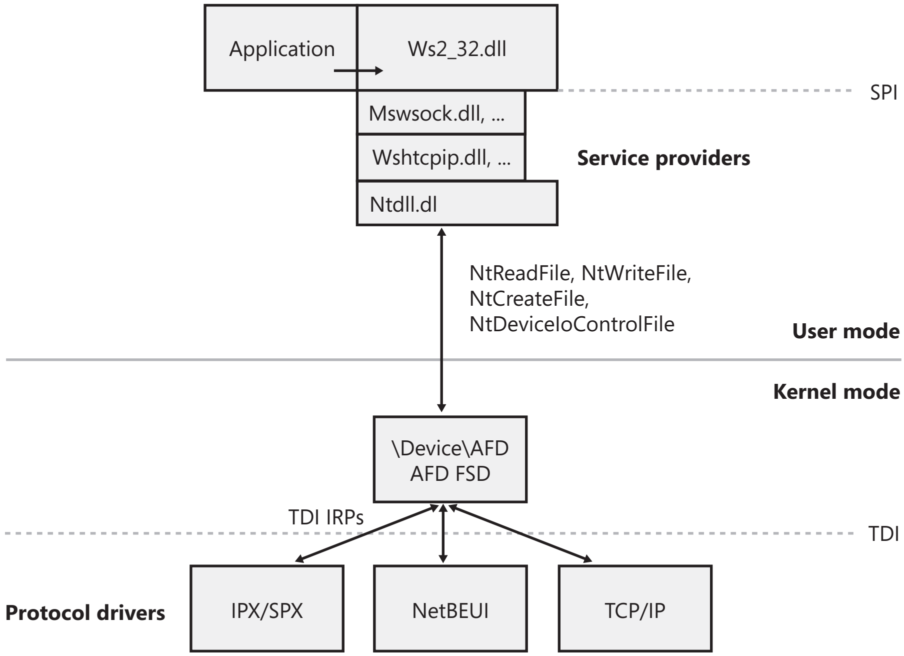


Figure 13-4 Winsock implementation

Like the named pipe and mailslot APIs (described later in this chapter), Winsock integrates with the Windows I/O model and uses file handles to represent sockets. This support requires the aid of a kernel-mode file system driver, so Msafd.dll uses the services of the Ancillary Function Driver (AFD - \Windows\System32\Drivers\Afd.sys) to implement socket-based functions. AFD is a TDI client and executes network socket operations, such as sending and receiving messages, by sending TDI IRPs to protocol drivers. AFD isn't coded to use particular protocol drivers; instead, Msafd.dll informs AFD of the name of the protocol used for each socket so that AFD can open the device object representing the protocol.

Windows Sockets Direct

Windows Sockets Direct (WSD) is an interface that allows Winsock applications to take advantage of System Area Networks (SANs), which are also known as Remote Direct Memory Access (RDMA) fabric, without application modification. The high-performance characteristics of SANs make them ideal for applications ranging from distributed computation to three-tier e-commerce architectures like the one shown in Figure 13-5. In the system depicted, SANs connect the front-end Web servers with business logic servers, and the business logic servers with backend database servers to provide high-speed data movement through the data-processing layers. WSD is available on Windows 2003 and Windows 2000 Data Center Server or Windows 2000 Advanced Server with Service Pack (SP) 2 or higher.

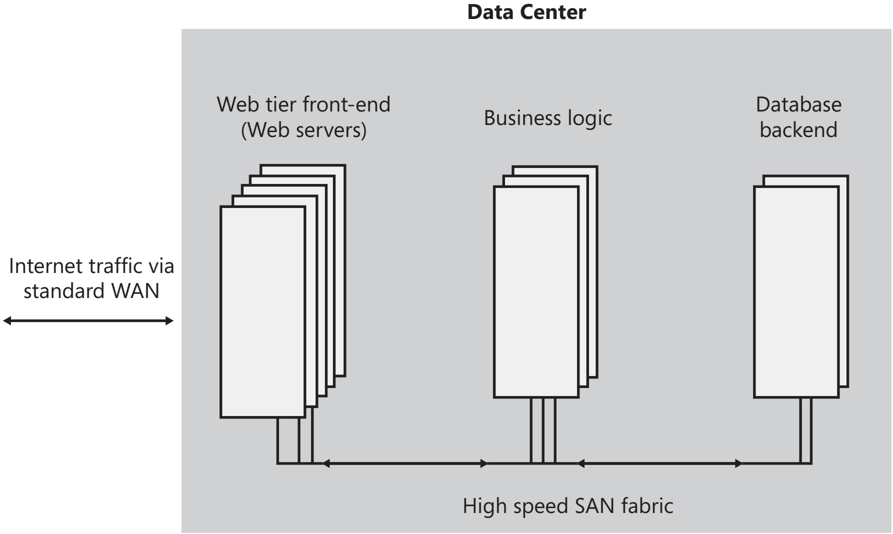

Data Center
Web tier front-end (Web servers)
Business logic
Database backend
Internet traffic via standard WAN
High speed SAN fabric

Figure 13-5 Use of a SAN in a three-tier e-commerce architecture

SAN Interconnect SANs usually achieve their high performance characteristics by relying on specialized network interconnect and switching hardware. Common SAN interconnect types include InfiniBand, Gigabit Ethernet, FiberChannel, and proprietary solutions. Physical memory shared between two computers can also serve as a SAN interconnect.

SAN switching hardware implements a nonroutable protocol that provides TCP-like guarantees such as reliable, in-order message delivery. They also support a SAN feature called Remote Direct Memory Access (RDMA) that allows messages to be transmitted directly from the physical memory of a source computer to that of a destination computer without the intermediate memory copy operation that normally takes place on the receive end of a message transmission. RDMA thus frees both CPU and memory bus bandwidth that a copy operation consumes.

SAN implementations also permit applications to bypass kernel-mode components altogether, sending and receiving data directly to or from user applications. This minimizes the number of system calls applications make and further reduces time spent in operating system networking code.

WSD Architecture Most SAN implementations require application changes for the applications to interface with SAN network protocols and take advantage of SAN hardware-implemented network protocols and SAN features such as RDMA, but WSD allows any Winsock-based application that uses the TCP protocol to take advantage of SANs without modification. WSD gets its name from the fact that it gives applications direct access to SAN hardware, bypassing the TCP/IP stack. This shortcut to the network hardware gives applications a performance improvement of 2 to 2.5.

WSD achieves its shortcut to SAN hardware by using a software switch beneath the Winsock DLL, as shown in Figure 13-6. The switch routes SAN network activity to a vendor-supplied SAN Winsock service provider (WSP). The WSP acts as the user-mode equivalent of an NDIS driver, with the capability to map SAN hardware registers into user-mode memory and then manipulate the hardware without going through kernel-mode components. Some network activity does require the aid of a kernel-mode component (for instance, to map hardware into user-mode memory), which is also provided by the SAN vendor. Finally, the SAN vendor supplies a NDIS miniport driver to interface the TCP/IP stack to SAN hardware for applications that use Winsock networking features not supported natively by the SAN.

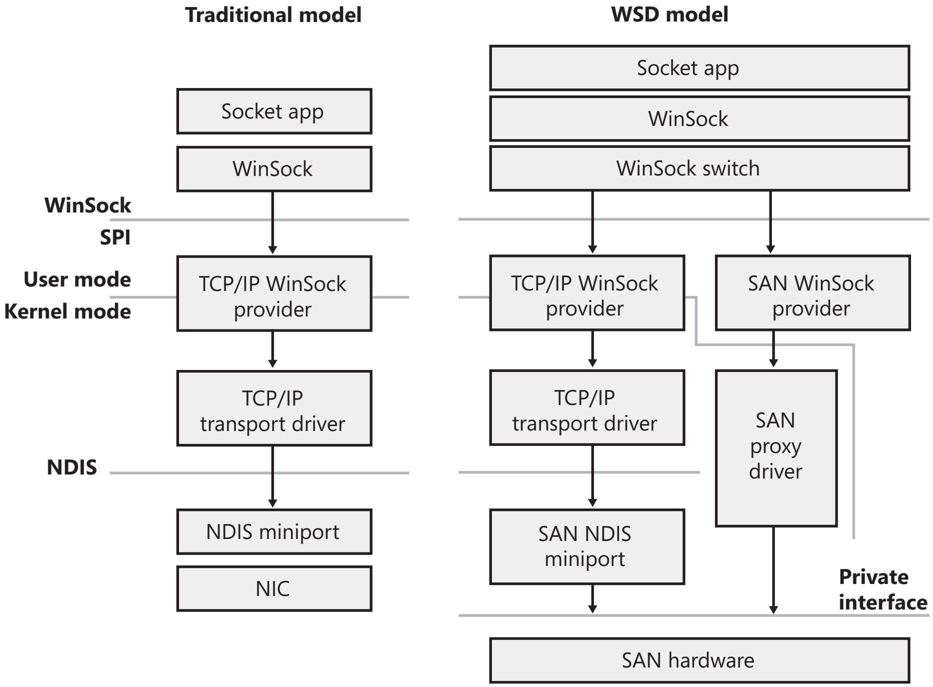


Figure 13-6 Traditional Winsock model vs. WSD model

Remote Procedure Call

Remote procedure call (RPC) is a network programming standard originally developed in the early 1980s. The Open Software Foundation (now The Open Group) made RPC part of the distributed computing environment (DCE) distributed computing standard. Although there is a second RPC standard, SunRPC, the Microsoft RPC implementation is compatible with the OSF/DCE standard. RPC builds on other networking APIs, such as named pipes or Winsock, to provide an alternate programming model that in some sense hides the details of networking programming from an application developer.

RPC Operation

An RPC facility is one that allows a programmer to create an application consisting of any number of procedures, some that execute locally and others that execute on remote computers via a network. It provides a procedural view of networked operations rather than a transport-centered view, thus simplifying the development of distributed applications.

Networking software is traditionally structured around an I/O model of processing. In Windows, for example, a network operation is initiated when an application issues a remote I/O request. The operating system processes it accordingly by forwarding it to a *redirector*, which acts as a remote file system by making the client interaction with the remote file system invisible to the client. The redirector passes the operation to the remote file system, and after the remote system fills the request and returns the results, the local network card interrupts. The kernel handles the interrupt, and the original I/O operation completes, returning results to the caller.

RPC takes a different approach altogether. RPC applications are like other structured applications, with a main program that calls procedures or procedure libraries to perform specific tasks. The difference between RPC applications and regular applications, however, is that some of the procedure libraries in an RPC application execute on remote computers, as shown in Figure 13-7, whereas others execute locally.

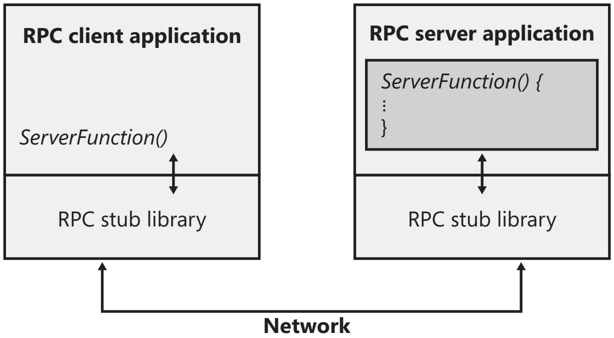


Figure 13-7 RPC operation

To the RPC application, all the procedures appear to execute locally. In other words, instead of making a programmer actively write code to transmit computational or I/O-related requests across a network, handle network protocols, deal with network errors, wait for results, and so forth, RPC software handles these tasks automatically. And the Windows RPC facility can operate over any available transports loaded into the system.

To write an RPC application, the programmer decides which procedures will execute locally and which will execute remotely. For example, suppose an ordinary workstation has a network connection to a Cray supercomputer or to a machine designed specifically for high-speed vector operations. If the programmer were writing an application that manipulated large matrices, it would make sense from a performance point of view to offload the mathematical calculations to the remote computer by writing the program as an RPC application.

RPC applications work like this: As an application runs, it calls local procedures as well as procedures that aren't present on the local machine. To handle the latter case, the application is linked to a local static-link library or DLL that contains *stub procedures*, one for each remote procedure. For simple applications, the stub procedures are statically linked with the application, but for bigger components the stubs are included in separate DLLs. In DCOM, covered later in the chapter, the latter method is typically used. The stub procedures have the same name and use the same interface as the remote procedures, but instead of performing the required operations, the stub takes the parameters passed to it and *marshals* them for transmission across the network. Marshaling parameters means ordering and packaging them in a particular way to suit a network link, such as resolving references and picking up a copy of any data structures that a pointer refers to.

The stub then calls RPC run-time procedures that locate the computer where the remote procedure resides, determine which transport mechanisms that computer uses, and send the request to it using local transport software. When the remote server receives the RPC request, it *unmarshals* the parameters (the reverse of marshaling them), reconstructs the original procedure call, and calls the procedure. When the server finishes, it performs the reverse sequence to return results to the caller.

In addition to the synchronous function-call-based interface described here, Windows RPC also supports *asynchronous RPC*. Asynchronous RPC lets an RPC application execute a function but not wait until the function completes to continue processing. Instead, the application can execute other code and later, when a response has arrived from the server, the RPC run time notifies the client that the operation has completed. The RPC run time uses the notification mechanism requested by the client. If the client uses an event synchronization object for notification, it waits for the signaling of the event object by calling the *WaitForSingleObject* or *WaitForMultipleObject* function. If the client provides an Asynchronous Procedure Call (APC), the run time queues the execution of the APC to the thread that executed the RPC function. If the client program uses I/O completion port as its notification mechanism, it must call *GetQueuedCompletionStatus* to learn of the function's completion. Alternatively, a client can poll for completion by calling *RcpAsyncGetCallStatus*.

In addition to the RPC run time, Microsoft's RPC facility includes a compiler, called the *Microsoft Interface Definition Language* (MIDL) compiler. The MIDL compiler simplifies the creation of an RPC application. The programmer writes a series of ordinary function prototypes (assuming a C or C++ application) that describe the remote routines and then places the routines in a file. The programmer then adds some additional information to these prototypes, such as a network-unique identifier for the package of routines and a version number, plus attributes that specify whether the parameters are input, output, or both. The embellished prototypes form the developer's Interface Definition Language (IDL) file.

Once the IDL file is created, the programmer compiles it with the MIDL compiler, which produces both client-side and server-side stub routines, mentioned previously, as well as header files to be included in the application. When the client-side application is linked to the stub routines file, all remote procedure references are resolved. The remote procedures are then installed, using a similar process, on the server machine. A programmer who wants to call an existing RPC application need only write the client side of the software and link the application to the local RPC run-time facility.

The RPC run time uses a generic RPC *transport provider interface* to talk to a transport protocol. The provider interface acts as a thin layer between the RPC facility and the transport, mapping RPC operations onto the functions provided by the transport. The Windows RPC facility implements transport provider DLLs for named pipes, NetBIOS, SPX, TCP/IP, and UDP. Windows Server 2003 removes the NetBIOS transport provider and adds a provider for HTTP.. In a similar fashion, the RPC facility is designed to work with different network security facilities.

Note In Windows 2000, you can write new provider DLLs to support additional transports, but support for add-on provider DLLs was removed as of Windows XP.

Most of the Windows networking services are RPC applications, which means that both local processes and processes on remote computers can call them. Thus, a remote client computer can call the server service to list shares, open files, write to print queues, or activate users on your server, or it can call the messenger service to direct messages to you (all subject to security constraints, of course).

Server name publishing, which is the ability of a server to register its name in a location accessible for client lookup, is in RPC and is integrated with Active Directory. If Active Directory isn't installed, the RPC name locator services fall back on NetBIOS broadcast. This behavior ensures interoperability with Windows NT 4 systems and allows RPC to function on standalone servers and workstations.

RPC Security

Windows RPC includes integration with security support providers (SSPs) so that RPC clients and servers can use authenticated or encrypted communications. When an RPC server wants secure communication, it tells the RPC run time what authentication service to add to the list of available *authentication services*. When a client wants to use secure communication, it binds to the server. At that time, it must tell the RPC run time the authentication service and *authentication level* it wants. Various authentication levels exist to ensure that only authorized clients connect to a server, verify that each message a server receives originates at an authorized client, check the integrity of RPC messages to detect manipulation, and even encrypt RPC message data. Obviously, higher authentication levels require more processing. The client can also optionally specify the server *principal name*. A principal is an entity that the RPC security system recognizes. The server must register its SSP-specific *principal name* with an SSP.

An SSP handles the details of performing network communication authentication and encryption, not only for RPC but also for Winsock. Windows includes a number of built-in SSPs, including a Kerberos SSP to implement Kerberos version 5 authentication, Secure Channel (SChannel), which implements Secure Sockets Layer (SSL) and the Transport Layer Security (TLS) protocols. In the absence of a specified SSP, RPC software uses the built-in security of the underlying transport. Some transports, such as named pipes or Local RPC, have built-in security. Others, like TCP do not, and in this case RPC makes insecure calls in the absence of a specified SSP.

Another feature of RPC security is the ability of a server to impersonate the security identity of a client with the *RpcImpersonateClient* function. After a server has finished performing impersonated operations on behalf of a client, it returns to its own security identity by calling *RpcRevertToSelf* or *RpcRevertToSelfEx*. (See Chapter 8 for more information on impersonation.)

RPC Implementation

RPC implementation is depicted in Figure 13-8, which shows that an RPC-based application links with the RPC run-time DLL (\Windows\System32\Rpcrt4.dll). The RPC run-time DLL provides marshaling and unmarshaling functions for use by an application's RPC function stubs as well as functions for sending and receiving marshaled data. The RPC run-time DLL includes support routines to handle RPC over a network as well as a form of RPC called *local RPC*. Local RPC can be used for communication between two processes located on the same system, and the RPC run-time DLL uses the local procedure call (LPC) facilities in kernel mode as the local networking API. (See Chapter 3 for more information on LPCs.) When RPC is based on nonlocal communication mechanisms, the RPC run-time DLL uses the Winsock, named pipe, or Message Queuing (described shortly) APIs.

Note Message Queuing is not supported as a transport mechanism in Windows Server 2003.

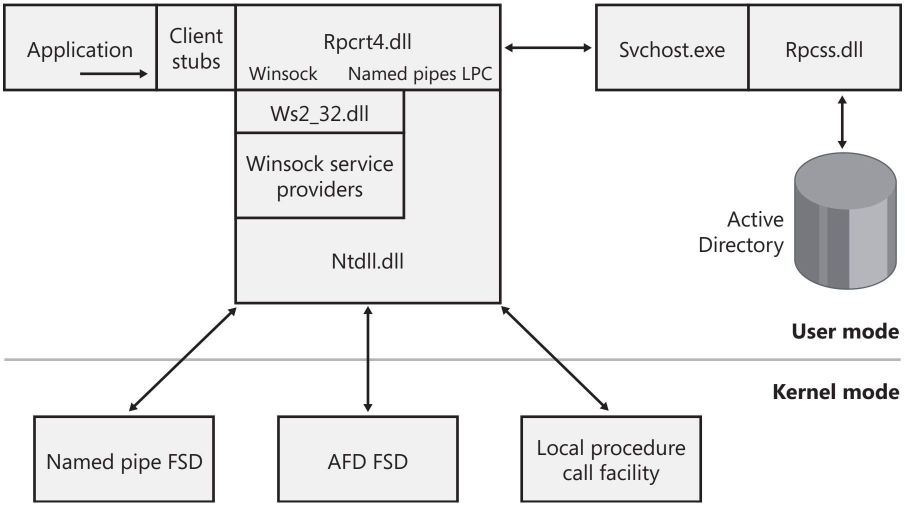


Figure 13-8 RPC implementation

The RPC Subsystem (RPCSS - \Windows\System32\Rpcss.dll) is implemented as a Windows service. RPCSS is itself an RPC application that communicates with instances of itself on other systems to perform name lookup, registration, and dynamic endpoint mapping. (For clarity, Figure 13-8 doesn't show RPCSS linked with the RPC run-time DLL.)

Web Access APIs

To ease the development of Internet applications, Windows provides both client and server Internet APIs. By using the APIs, applications can provide and use Gopher, FTP, and HTTP services without knowledge of the intricacies of the corresponding protocols. The client APIs include Windows Internet, also known as WinInet—which enables applications to interact with the Gopher, FTP, and HTTP protocols—and WinHTTP, which enables applications to interact with the HTTP protocol and is more suitable than WinInet in certain situations. HTTP is a server-side API that was introduced in Windows Server 2003 to enable the development of Web server applications.

WinInet

WinInet supports the Gopher, FTP, and HTTP 1.0 and 1.1 protocols. The APIs break down into sub-API sets specific to each protocol. Using the FTP-related APIs—such as *InternetConnect* to connect to an FTP server, *FtpFindFirstFile* and *FtpFindNextFile* to enumerate the contents of an FTP directory, and *FtpGetFile* and *FtpPutFile* to receive and send files—an application developer avoids the details of establishing a connection and formatting TCP/IP messages to the FTP protocol. The Gopher-related and HTTP-related APIs provide similar levels of abstraction with the HTTP APIs providing cookie persistence, automatic dialup services, client-side file caching, and automatic credential dialog handling. WinInet is used by core Windows components such as Explorer and Internet Explorer.

WinHTTP

The current version of the WinHTTP API, 5.1, is available on Windows 2000 with Service Pack 3 installed, Windows XP, and Windows Server 2003. It provides an abstraction of the HTTP/1.1 protocol for HTTP client applications similar to what the WinInet HTTP-related APIs provide. However, whereas the WinInet HTTP APIs are intended for user-interactive client-side applications, the WinHTTP API is designed for server applications that communicate with HTTP servers. Server applications are often implemented as Windows services that do not provide a user interface and so do not desire the dialog boxes that WinInet APIs display. In addition, the WinHTTP APIs are more scalable and offer security functionality, such as thread impersonation, not available from the WinInet API.

HTTP

Using the HTTP API, which Windows Server 2003 implements, server applications can register to receive HTTP requests for particular URLs, receive HTTP requests, and send HTTP responses. The HTTP API includes Secure Sockets Layer (SSL) support so that applications can exchange data over secure HTTP connections. The API includes server-side caching capabilities, synchronous and asynchronous I/O models, and both IPv4 and IPv6 addressing. Internet Information Services (IIS) version 6, the version that ships with Windows Server 2003, uses the HTTP API.

The HTTP API, which applications access through the Httpapi.dll library, relies on the kernel-mode Http.sys driver. Http.sys starts on demand the first time any application on the system calls *HttpInitialize*. Applications use *HttpCreateHttpHandle* to create a private request queue and specify the URLs that it wants to handle requests for with *HttpAddUrl*. Using the request queues and their registered URLs, Http.sys allows more than one application to service HTTP requests on a given port (port 80 for example), with each servicing HTTP requests to different parts of the URL namespace.

HttpReceiveHttpRequest receives incoming requests directed at registered URLs, and *HttpSendHttpResponse* sends HTTP responses. Both functions offer asynchronous operation so that an application can use *GetOverlappedResult* or I/O completion ports to determine when an operation is completed.

Applications can use Http.sys to cache data in nonpaged physical memory by calling *HttpAddToFragmentCache* and associating a *fragment name* with the cached data. Http.sys invokes the Memory Manager function *MmAllocatePagesForMdl* to allocate unmapped physical pages. When Http.sys requires a virtual address mapping for the physical memory described by an entry in the cache, for instance when it copies data to the cache or sends data from the cache, it uses *MmMapLockedPagesSpecifyCache* and then *MmUnmapLockedPages* after it completes its access. Http.sys maintains cached data until an application invalidates it or an optional application-specified timeout associated with the data expires. Http.sys also trims cached data in a worker thread that wakes up when the low memory notification event is signaled. (See Chapter 7 for information on the low memory notification event.) When an application specifies one or more fragment names in a call to *HttpSendHttpResponse*, Http.sys passes a pointer to the cached data in physical memory to the TCP/IP driver and avoids a copy operation.

Named Pipes and Mailslots

Named pipes and mailslots are programming APIs that Microsoft originally developed for OS/2 LAN Manager and then ported to Windows NT. Named pipes provide for reliable bidirectional communications, whereas mailslots provide unreliable unidirectional data transmission. An advantage of mailslots is that they support broadcast capability. In Windows, both APIs take advantage of Windows security, which allows a server to control precisely which clients can connect to it.

The names servers assign to named pipes and clients conform to the Windows Universal Naming Convention (UNC), which is a protocol-independent way to identify resources on a Windows network. The implementation of UNC names is described later in the chapter.

Named Pipe Operation

Named pipe communication consists of a named pipe server and a named pipe client. A named pipe server is an application that creates a named pipe to which clients can connect. A named pipe's name has the format \\Server\Pipe\PipeName. The *Server* component of the name specifies the computer on which the named pipe server is executing. (A named pipe server can't create a named pipe on a remote system.) The name can be a DNS name (for example, *mspress.microsoft.com*), a NetBIOS name (*mspress*), or an IP address (131.107.0.1). The *Pipe* component of the name must be the string "Pipe", and *PipeName* is the unique name assigned to a named pipe. The unique portion of the named pipe's name can include subdirectories; an example of a named pipe name with a subdirectory is \\MyComputer\Pipe\MyServerApp\ConnectionPipe.

A named pipe server uses the *CreateNamedPipe* Windows function to create a named pipe. One of the function's input parameters is a pointer to the named pipe name, in the form\\.\Pipe\PipeName. The "\\.\" is a Windows-defined alias for "this computer." Other parameters the function accepts include an optional security descriptor that protects access to the named pipe, a flag that specifies whether the pipe should be bidirectional or unidirectional, a value indicating the maximum number of simultaneous connections the pipe supports, and a flag specifying whether the pipe should operate in *byte mode* or *message mode*.

Most networking APIs operate only in byte mode, which means that a message sent with one send function might require the receiver to perform multiple receives, building up the complete message from fragments. Named pipes operating in message mode simplify the implementation of a receiver because there is a one-to-one correspondence between sends and receives. A receiver therefore obtains an entire message each time it completes a receive and doesn't have to concern itself with keeping track of message fragments.

The first call to *CreateNamedPipe* for a particular name creates the first instance of that name and establishes the behavior of all named pipe instances having that name. A server creates additional instances, up to the maximum specified in the first call, with additional calls to *CreateNamedPipe*. After creating at least one named pipe instance, a server executes the *ConnectNamedPipe* Windows function, which enables the named pipe the server created to establish connections with clients. *ConnectNamedPipe* can be executed synchronously or asynchronously, and it doesn't complete until a client establishes a connection with the instance (or an error occurs).

A named pipe client uses the Windows *CreateFile* or *CallNamedPipe* function, specifying the name of the pipe a server has created, to connect to a server. If the server has performed a *ConnectNamedPipe* call, the client's security profile and the access it requests to the pipe (read, write) are validated against the named pipe's security descriptor. (See Chapter 8 for more information on the security-check algorithms Windows uses.) If the client is granted access to a named pipe, it receives a handle representing the client side of a named pipe connection and the server's call to *ConnectNamedPipe* completes.

After a named pipe connection is established, the client and server can use the *ReadFile* and *WriteFile* Windows functions to read from and write to the pipe. Named pipes support both synchronous and asynchronous operation for message transmittal. Figure 13-9 shows a server and client communicating through a named pipe instance.

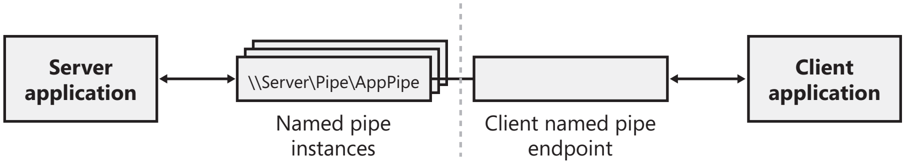


Figure 13-9 Named pipe communications

A unique characteristic of the named pipe networking API is that it allows a server to impersonate a client by using the *ImpersonateNamedPipeClient* function. See the "Impersonation" section in Chapter 8 for a discussion of how impersonation is used in client/server applications.

Mailslot Operation

Mailslots provide an unreliable unidirectional broadcast mechanism. One example of an application that can use this type of communication is a time-synchronization service, which might broadcast a source time across the domain every few seconds. Receiving the source-time message isn't crucial for every computer on the network and is therefore a good candidate for the use of mailslots.

Like named pipes, mailslots are integrated with the Windows API. A mailslot server creates a mailslot by using the *CreateMailslot* function. *CreateMailslot* accepts a name of the form "\\.\Mailslot\MailslotName" as an input parameter. Again, like named pipes, a mailslot server can create mailslots only on the machine it's executing on, and the name it assigns to a mailslot can include subdirectories. *CreateMailslot* also takes a security descriptor that controls client access to the mailslot. The handles returned by *CreateMailslot* are *overlapped*, which means that operations performed on the handles, such as sending and receiving messages, are asynchronous.

Because mailslots are unidirectional and unreliable, *CreateMailslot* doesn't take many of the parameters that *CreateNamedPipe* does. After it creates a mailslot, a server simply listens for incoming client messages by executing the *ReadFile* function on the handle representing the mailslot.

Mailslot clients use a naming format similar to that used by named pipe clients but with variations that make it possible to broadcast messages to all the mailslots of a given name within the client's domain or a specified domain. To send a message to a particular instance of a mailslot, the client calls *CreateFile*, specifying the computer-specific name. An example of such a name is "\\Server\Mailslot\MailslotName". (The client can specify "\\.\" to represent the local computer.) If the client wants to obtain a handle representing all the mailslots of a given name on the domain it's a member of, it specifies the name in the format "\\*\Mailslot\MailslotName", and if the client wants to broadcast to all the mailslots of a given name within a different domain, the format it uses is "\\DomainName\Mailslot\MailslotName".

After obtaining a handle representing the client side of a mailslot, the client sends messages by calling *WriteFile*. Because of the way mailslots are implemented, only messages smaller than 425 bytes can be broadcast. If a message is larger than 425 bytes, the mailslot implementation uses a reliable communications mechanism that requires a one-to-one client/server connection, which precludes broadcast capability. Also, a quirk of the mailslot implementation causes any messages of 425 or 426 bytes to be truncated to 424 bytes. These limitations make mailslots generally unsuitable for messages larger than 424 bytes. Figure 13-10 shows an example of a client broadcasting to multiple mailslot servers within a domain.

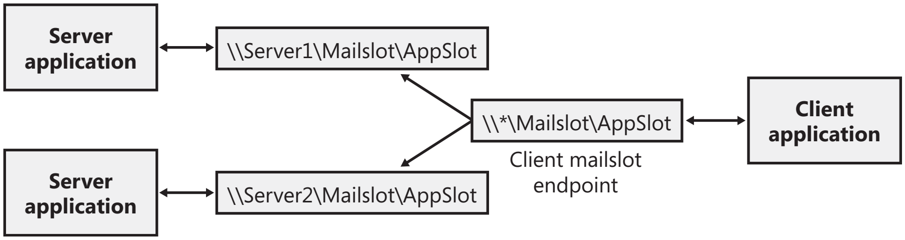


Figure 13-10 Mailslot broadcast

Named Pipe and Mailslot Implementation

As evidence of their tight integration with Windows, named pipe and mailslot functions are all implemented in the Kernel32.dll Windows client-side DLL. *ReadFile* and *WriteFile*, which are the functions applications use to send and receive messages using named pipes and mailslots, are the primary Windows I/O routines. The *CreateFile* function, which a client uses to open either a named pipe or a mailslot, is also a standard Windows I/O routine. However, the names specified by named pipe and mailslot applications specify file system namespaces managed by the named pipe file system driver (\Windows\System32\Drivers\Npfs.sys) and the mailslot file system driver (\Windows\System32\Drivers\Msfs.sys), as shown in Figure 13-11. The named pipe file system driver creates a device object named \Device\NamedPipe and a symbolic link to that object named \Global??\Pipe (\??\Pipe on Windows 2000), and the mailslot file system driver creates a device object named \Device\Mailslot and a symbolic link named \Global??\Mailslot that points to that object. (See Chapter 3 for an explanation of the \Global?? object manager directory.) Names passed to *CreateFile* of the form \\.\Pipe\... and \\.\Mailslot\... have their prefix of \\.\ translated to \Global??\ so that the names resolve through a symbolic link to a device object. The special functions *CreateNamedPipe* and *CreateMailslot* use the corresponding native functions *NtCreateNamedPipeFile* and *NtCreateMailslotFile*.

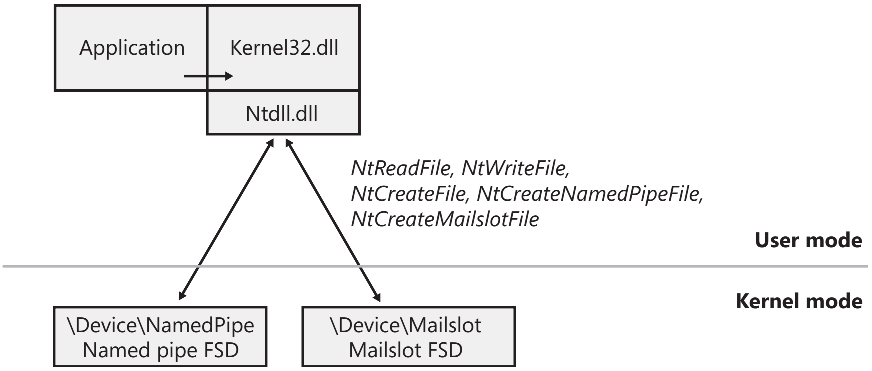


Figure 13-11 Named pipe and mailslot implementation

Later in the chapter, we'll discuss how the redirector file system driver is involved when a name that specifies a remote named pipe or mailslot resolves to a remote system. However, when a named pipe or mailslot is created by a server or opened by a client, the appropriate file system driver (FSD) on the machine where the named pipe or mailslot is located is eventually invoked. There are several reasons why FSDs in kernel mode implement named pipes and mailslots, the main one being that they integrate with the object manager namespace and can use file objects to represent opened named pipes and mailslots. This integration results in several benefits:

- The FSDs use kernel-mode security functions to implement standard Windows security for named pipes and mailslots.
- Applications can use *CreateFile* to open a named pipe or mailslot because FSDs integrate with the object manager namespace.
- Applications can use Windows functions such as *ReadFile* and *WriteFile* to interact with named pipes and mailslots.
- The FSDs rely on the object manager to track handle and reference counts for file objects representing named pipes and mailslots.
- The FSDs can implement their own named pipe and mailslot namespaces, complete with subdirectories.

Because named pipes and mailslot name resolution uses the redirector FSD to communicate across the network, they indirectly rely on the Common Internet File System (CIFS) protocol. CIFS works by using the TCP/IP, TCP/IP with IPv6, and IPX protocols, so applications running on systems that have at least one of these in common can use named pipes and mailslots. For more information about CIFS, see Chapter 12.

EXPERIMENT: Listing the Named Pipe Namespace and Watching Named Pipe Activity

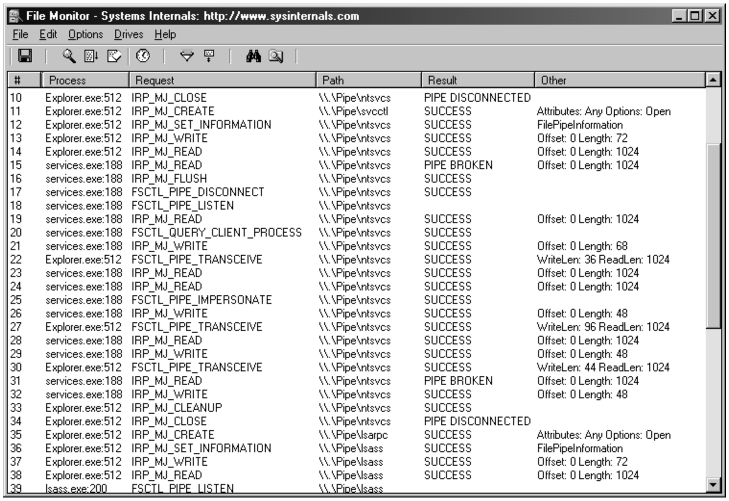
It’s not possible to use the Windows API to open the root of the named pipe FSD and perform a directory listing, but you can do this by using native API services. The PipeList tool from *www.sysinternals.com* shows you the names of the named pipes defined on a computer as well as the number of instances that have been created for a name and the maximum number of instances as defined by a server’s call to *CreateNamedPipe*. Here’s an example of PipeList output:

```
C:\>pipelist

PipeList v1.01
by Mark Russinovich
http://www.sysinternals.com

Pipe Name                          Instances       Max Instances
---------                          ---------       -------------
TerminalServer\AutoReconnect            1                1
InitShutdown                            2               -1
lsass                                   4               -1
protected_storage                       2               -1
ntsvcs                                 58               -1
scerpc                                  2               -1
net\NtControlPipe1                      1                1
net\NtControlPipe2                      1                1
ExtEventPipe_Service                    1               30
net\NtControlPipe3                      1                1
Winsock2\CatalogChangeListener-37c-0    1                1
net\NtControlPipe4                      1                1
net\NtControlPipe0                      1                1
DhcpClient                              1               -1
ProfMapApi                              2               -1
winlogonrpc                             2               -1
net\NtControlPipe5                      1                1
SfcApi                                  2               -1
net\NtControlPipe6                      1                1
Winsock2\CatalogChangeListener-4c4-0    1                1
atsvc                                   2               -1
epmapper                                2               -1
net\NtControlPipe7                      1                1
spoolss                                 2               -1
wkssvc                                  3               -1
DAV RPC SERVICE                         2               -1
net\NtControlPipe8                      1                1
keysvc                                  2               -1
net\NtControlPipe9                      1                1
```

```
PCHHangRepExecPipe                  1              8
PCHFaultRepExecPipe                 1              8
net\NtControlPipe10                 1              1
000000ac.000                        2             -1
Winsock2\CatalogChangeListener-ac-0 1              1
net\NtControlPipe11                 1              1
net\NtControlPipe12                 1              1
winreg                              2             -1
SECLOGON                            2             -1
srvsvc                              3             -1
ipsec                               2             -1
trkwks                              2             -1
Winsock2\CatalogChangeListener-ac-1 1              1
net\NtControlPipe13                 1              1
vmware-authdpipe                    1             -1
W32TIME                             2             -1
net\NtControlPipe14                 1              1
net\NtControlPipe15                 1              1
Winsock2\CatalogChangeListener-e8-0 1              1
INETINFO                            2             -1
SMTPSVC                             2             -1
browser                             3             -1
net\NtControlPipe16                 1              1
Ctx_WinStation_API_service          2             -1
ExtEventPipe_s0                     1             30
ExtEventPipe_s0_poller              1             30
net\NtControlPipe17                 1              1
interbas\server\gds_db              1             -1
ROUTER                              8             -1
PIPE_EVENTROOT\CIMV2SCM EVENT PROVIDER  2         -1
net\NtControlPipe20                 1              1
```

It's clear from this output that several system components use named pipes as their communications mechanism. For example, the *InitShutdown* pipe is created by Winlogon to accept remote shutdown commands, and the *SecLogon* pipe is created by the SecLogon service to perform logon operations on behalf of the Runas utility. You can determine what process has each of these pipes open by using the object search facility in Process Explorer from *www.sysinternals.com*. Note that a Max Instances value of –1 means that there is no upper limit on the number of instances for the given name.

The Filemon file system filter driver from *www.sysinternals.com* (see Chapter 12 for more information on Filemon) is able to attach to either the Npfs.sys or Msfs.sys file system drivers and to therefore see all named pipe or mailslot activity occurring on a system. Select the Named Pipes or Mail Slots menu entries from Filemon's Drives menu to have Filemon attach to the corresponding driver. The following screen shot shows Filemon capturing the named pipe activity generated when the My Network Places icon on the desktop was opened. Notice the messages transmitted through the LSASS and workstation service named pipes.


File Monitor - Systems Internals: http://www.sysinternals.com
File Edit Options Drives Help

#	Process	Request	Path	Result	Other
10	Explorer.exe:512	IRP_MJ_CLOSE	\\.\Pipe\ntsvcs	PIPE DISCONNECTED	
11	Explorer.exe:512	IRP_MJ_CREATE	\\.\Pipe\svcctl	SUCCESS	Attributes: Any Options: Open
12	Explorer.exe:512	IRP_MJ_SET_INFORMATION	\\.\Pipe\ntsvcs	SUCCESS	FilePipeInformation
13	Explorer.exe:512	IRP_MJ_WRITE	\\.\Pipe\ntsvcs	SUCCESS	Offset: 0 Length: 72
14	Explorer.exe:512	IRP_MJ_READ	\\.\Pipe\ntsvcs	SUCCESS	Offset: 0 Length: 1024
15	services.exe:188	IRP_MJ_READ	\\.\Pipe\ntsvcs	PIPE BROKEN	Offset: 0 Length: 1024
16	services.exe:188	IRP_MJ_FLUSH	\\.\Pipe\ntsvcs	SUCCESS	
17	services.exe:188	FSCTL_PIPE_DISCONNECT	\\.\Pipe\ntsvcs	SUCCESS	
18	services.exe:188	FSCTL_PIPE_LISTEN	\\.\Pipe\ntsvcs		
19	services.exe:188	IRP_MJ_READ	\\.\Pipe\ntsvcs	SUCCESS	Offset: 0 Length: 1024
20	services.exe:188	FSCTL_QUERY_CLIENT_PROCESS	\\.\Pipe\ntsvcs	SUCCESS	
21	services.exe:188	IRP_MJ_WRITE	\\.\Pipe\ntsvcs	SUCCESS	Offset: 0 Length: 68
22	Explorer.exe:512	FSCTL_PIPE_TRANSCEIVE	\\.\Pipe\ntsvcs	SUCCESS	WriteLen: 36 ReadLen: 1024
23	services.exe:188	IRP_MJ_READ	\\.\Pipe\ntsvcs	SUCCESS	Offset: 0 Length: 1024
24	services.exe:188	IRP_MJ_READ	\\.\Pipe\ntsvcs	SUCCESS	Offset: 0 Length: 1024
25	services.exe:188	FSCTL_PIPE_IMPERSONATE	\\.\Pipe\ntsvcs	SUCCESS	
26	services.exe:188	IRP_MJ_WRITE	\\.\Pipe\ntsvcs	SUCCESS	Offset: 0 Length: 48
27	Explorer.exe:512	FSCTL_PIPE_TRANSCEIVE	\\.\Pipe\ntsvcs	SUCCESS	WriteLen: 96 ReadLen: 1024
28	services.exe:188	IRP_MJ_READ	\\.\Pipe\ntsvcs	SUCCESS	Offset: 0 Length: 1024
29	services.exe:188	IRP_MJ_WRITE	\\.\Pipe\ntsvcs	SUCCESS	Offset: 0 Length: 48
30	Explorer.exe:512	FSCTL_PIPE_TRANSCEIVE	\\.\Pipe\ntsvcs	SUCCESS	WriteLen: 44 ReadLen: 1024
31	services.exe:188	IRP_MJ_READ	\\.\Pipe\ntsvcs	PIPE BROKEN	Offset: 0 Length: 1024
32	services.exe:188	IRP_MJ_WRITE	\\.\Pipe\ntsvcs	SUCCESS	Offset: 0 Length: 48
33	Explorer.exe:512	IRP_MJ_CLEANUP	\\.\Pipe\ntsvcs	SUCCESS	
34	Explorer.exe:512	IRP_MJ_CLOSE	\\.\Pipe\ntsvcs	PIPE DISCONNECTED	
35	Explorer.exe:512	IRP_MJ_CREATE	\\.\Pipe\lsarpc	SUCCESS	Attributes: Any Options: Open
36	Explorer.exe:512	IRP_MJ_SET_INFORMATION	\\.\Pipe\lsass	SUCCESS	FilePipeInformation
37	Explorer.exe:512	IRP_MJ_WRITE	\\.\Pipe\lsass	SUCCESS	Offset: 0 Length: 72
38	Explorer.exe:512	IRP_MJ_READ	\\.\Pipe\lsass	SUCCESS	Offset: 0 Length: 1024
39	lsass.exe:200	FSCTL_PIPE_LISTEN	\\.\Pipe\lsass		

NetBIOS

Until the 1990s, the Network Basic Input/Output System (NetBIOS) programming API had been the most widely used programming API on PCs. NetBIOS allows for both reliable-connection-oriented and unreliable-connectionless communication. Windows supports NetBIOS for its legacy applications. Microsoft discourages application developers from using NetBIOS because other APIs, such as named pipes and Winsock, are much more flexible and portable. NetBIOS is supported by the TCP/IP and IPX/SPX protocols on Windows.

NetBIOS Names

NetBIOS relies on a naming convention whereby computers and network services are assigned a 16-byte name called a NetBIOS name. The 16th byte of a NetBIOS name is treated as a modifier that can specify a name as unique or as part of a group. Only one instance of a unique NetBIOS name can be assigned to a network, but multiple applications can assign the same group name. A client can broadcast messages by sending them to a group name.

To support interoperability with Windows NT 4 systems as well as Consumer Windows, Windows automatically defines a NetBIOS name for a domain that is up to the first 15 bytes of the left-most Domain Name System (DNS) name that an administrator assigns to the domain. For example, if a domain were named *mspress.microsoft.com*, the NetBIOS name of the domain would be *mspress*. Similarly, Windows requires an administrator to assign each computer a NetBIOS name at the time of installation.

Another concept used by NetBIOS is that of LAN adapter (LANA) numbers. A LANA number is assigned to every NetBIOS-compatible protocol that layers above a network adapter. For example, if a computer has two network adapters and TCP/IP and NWLink can use either adapter, there would be four LANA numbers. LANA numbers are important because a NetBIOS application must explicitly assign its service name to each LANA through which it's willing to accept client connections. If the application listens for client connections on a particular name, clients can access the name only via protocols on the network adapters for which the name is registered.

The "Windows Internet Name Service" section later in this chapter describes the resolution of NetBIOS names to TCP/IP addresses.

NetBIOS Operation

A NetBIOS server application uses the NetBIOS API to enumerate the LANAs present on a system and assign a NetBIOS name representing the application's service to each LANA. If the server is connection oriented, it performs a NetBIOS listen command to wait for client connection attempts. After a client is connected, the server executes NetBIOS functions to send and receive data. Connectionless communication is similar, but the server simply reads messages without establishing connections.

A connection-oriented client uses NetBIOS functions to establish a connection with a NetBIOS server and then executes further NetBIOS functions to send and receive data. An established NetBIOS connection is also known as a *session*. If the client wants to send connectionless messages, it simply specifies the NetBIOS name of the server with the send function.

NetBIOS consists of a number of functions, but they all route through the same interface: *Netbios*. This routing scheme is the result of a legacy left over from the time when NetBIOS was implemented on MS-DOS as an MS-DOS interrupt service. A NetBIOS application would execute an MS-DOS interrupt and pass a data structure to the NetBIOS implementation that specified every aspect of the command being executed. As a result, the *Netbios* function in Windows takes a single parameter, which is a data structure that contains the parameters specific to the service the application requests.

EXPERIMENT: Using *Nbtstat* to See NetBIOS Names

You can use the *Nbtstat* command, which is included with Windows, to list the active sessions on a system, the NetBIOS-to-TCP/IP name mappings cached on a computer, and the NetBIOS names defined on a computer. Here's an example of the *Nbtstat* command with the *–n* option, which lists the NetBIOS names defined on the computer:

```
C:\>nbtstat -n
Local Area Connection:
Node IpAddress: [10.1.23.147] Scope Id: []                NetBIOS Local Name Table
   Name               Type         Status   ------------------------------------------
---
    MARKLAP        <00>  UNIQUE      Registered     WORKGROUP      <00>  GROUP       R
egistered     MARKLAP        <20>  UNIQUE      Registered     WORKGROUP      <1E>  GRO
UP       Registered     WORKGROUP      <1D>  UNIQUE      Registered     ..__MSBROWSE__
.<01>  GROUP       Registered
Wireless Network Connection:
Node IpAddress: [0.0.0.0] Scope Id: []    No names in cache
```

NetBIOS API Implementation

The components that implement the NetBIOS API are shown in Figure 13-12. The *Netbios* function is exported to applications by \Windows\System32\Netapi32.dll. Netapi32.dll opens a handle to the kernel-mode driver named the *NetBIOS emulator* (\Windows\System32\Drivers\Netbios.sys) and issues Windows *DeviceIoControl* file commands on behalf of an application. The NetBIOS emulator translates NetBIOS commands issued by an application into TDI commands that it sends to protocol drivers.

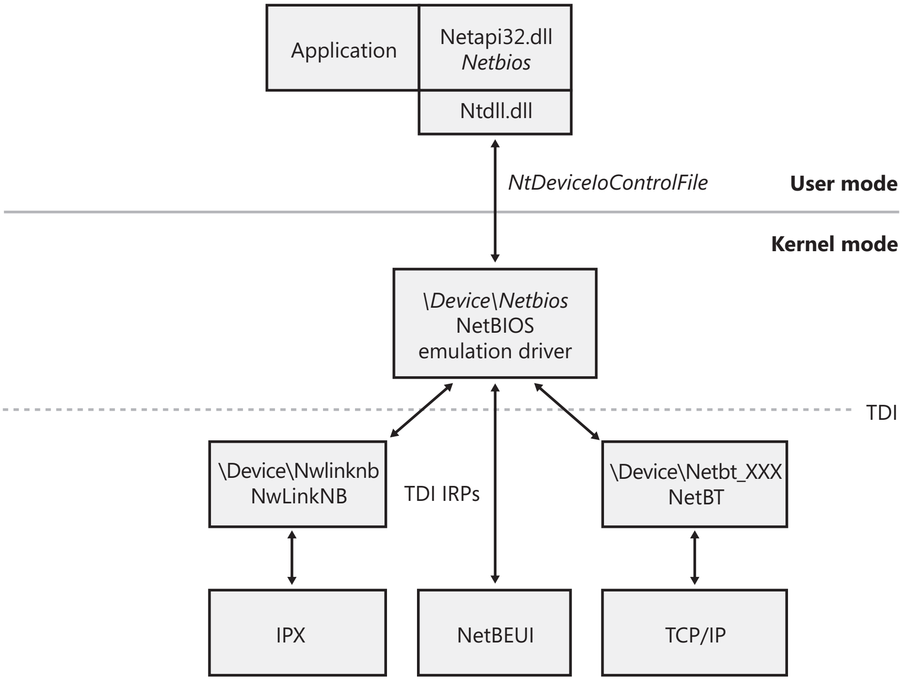


Figure 13-12 NetBIOS API implementation

If an application wants to use NetBIOS over the TCP/IP protocol, the NetBIOS emulator requires the presence of the NetBT driver (\Windows\System32\Drivers\Netbt.sys). NetBT is known as the NetBIOS over TCP/IP driver and is responsible for supporting NetBIOS semantics that are inherent to the NetBIOS Extended User Interface (NetBEUI) protocol (included in previous versions of Windows), but not the TCP/IP protocol. For example, NetBIOS relies on NetBEUI's message-mode transmission and NetBIOS name resolution facilities, so the NetBT driver implements them on top of the TCP/IP protocol. Similarly, the NwLinkNB driver implements NetBIOS semantics over the IPX/SPX protocol.

Other Networking APIs

Windows includes other networking APIs that are used less frequently or are layered on the APIs already described (and outside the scope of this book). Four of these, however, Real-Time

Communications (RTC), Distributed Component Object Model (DCOM), Message Queuing, and Universal Plug and Play (UPnP), are important enough to the operation of a Windows system and many applications to merit brief descriptions.

Real-Time Communications (RTC)

The RTC Client API, available on Windows XP and Windows Server 2003, enables developers to create applications to establish integrated multimodal communications. Applications can be developed to enable the PC to become the center for home or business communications. Audio and video calls as well as Instant Messaging (IM) and collaboration are all integrated into one communications session on the PC. In addition to PC-PC sessions, the API can establish PC-phone calls, phone-phone calls, or text-only IM sessions. Application sharing and whiteboard are also available on PC-PC sessions.

RTC supports presence information to allow clients to call contacts (or buddies) through a registrar server that maintains current location information on contacts. A contact's location can be a PC or a telephone and, in the future, a cell phone, pager, or handheld device. For example, if the application attempts to connect to a contact at his work location and the presence information indicates the contact is available on a PC at home, RTC will automatically redirect the connection to that location. The RTC API also allows users to maintain privacy by blocking specific callers from their presence information.

DCOM

Microsoft's COM API lets applications consist of different components, each component being a replaceable self-contained module. A COM object exports an object-oriented interface to methods for manipulating the data within the object. Because COM objects present well-defined interfaces, developers can implement new objects to extend existing interfaces and dynamically update applications with the new support.

DCOM extends COM by letting an application's components reside on different computers, which means that applications don't need to be concerned that one COM object might be on the local computer and another might be across the LAN. DCOM thus provides location transparency, which simplifies developing distributed applications. DCOM isn't a self-contained API but relies on RPC to carry out its work.

Message Queuing

Message Queuing is a general-purpose platform for developing distributed applications that take advantage of loosely coupled messaging. Message Queuing is therefore an API and a messaging infrastructure. Its flexibility comes from the fact that its queues serve as message repositories in which senders can queue messages for receivers, and receivers can de-queue the messages at their discretion. Senders and receivers do not need to establish connections to use Message Queuing, nor do they even need to be executing at the same time, which allows for disconnected asynchronous message exchange.

A notable feature of Message Queuing is that it is integrated with Microsoft Transaction Server (MTS) and SQL Server, so it can participate in Microsoft Distributed Transaction Coordinator (MS DTC) coordinated transactions. Using MS DTC with Message Queuing allows you to develop reliable transaction functionality to three-tier applications.

UPnP

Universal Plug and Play is an architecture for peer-to-peer network connectivity of intelligent appliances, devices, and *control points*. It is designed to bring easy-to-use, flexible, standards-based connectivity to ad-hoc, managed, or unmanaged networks, whether these networks are in the home, in small businesses, or attached directly to the Internet. Universal Plug and Play is a distributed, open networking architecture that uses existing TCP/IP and Web technologies to enable seamless proximity networking in addition to control and data transfer among networked devices.

Universal Plug and Play supports zero-configuration, invisible networking, and automatic discovery for a range of device categories from a wide range of vendors. This enables a device to dynamically join a network, obtain an IP address, and convey its capabilities upon request. Then other control points can use the Control Point API with UPnP technology to learn about the presence and capabilities of other devices. A device can leave a network smoothly and automatically when it is no longer in use.

Multiple Redirector Support

Applications can examine or access resources on remote systems in two ways. One way is by using the UNC standard with Windows functions to directly address a remote resource; a second way is by using the Windows Networking (WNet) API to enumerate computers and resources that those computers export for sharing. Both these approaches use the capabilities of a redirector to find their way to the network. (CIFS is described in Chapter 12.) As we stated earlier, to access CIFS servers from a client, Microsoft supplies a CIFS redirector, which has a kernel-mode component called the redirector FSD and a user-mode component called the Workstation service. Microsoft also makes available a redirector that can access resources shared by Novell NetWare servers, and third parties can add their own redirectors to Windows. In this section, we'll examine the software that decides which redirector to invoke when remote I/O requests are issued. Here are the responsible components:

- *Multiple provider router* (MPR) is a DLL (\Windows\System32\Mpr.dll) that determines which network to access when an application uses the Windows WNet API for browsing remote file systems.
- *Multiple UNC Provider* (MUP) is a driver (\Windows\System32\Drivers\Mup.sys) that determines which network to access when an application uses the Windows I/O API to open remote files.

Multiple Provider Router

The Windows WNet functions allow applications (including the Windows Explorer My Network Places) to connect to network resources, such as file servers and printers, and to browse the contents of any type of remote file system. Because the WNet API can be called to work across different networks using different transport protocols, software must be present to send the request correctly over the network and to understand the results that the remote server returns. Figure 13-13 shows the redirector software responsible for these tasks.

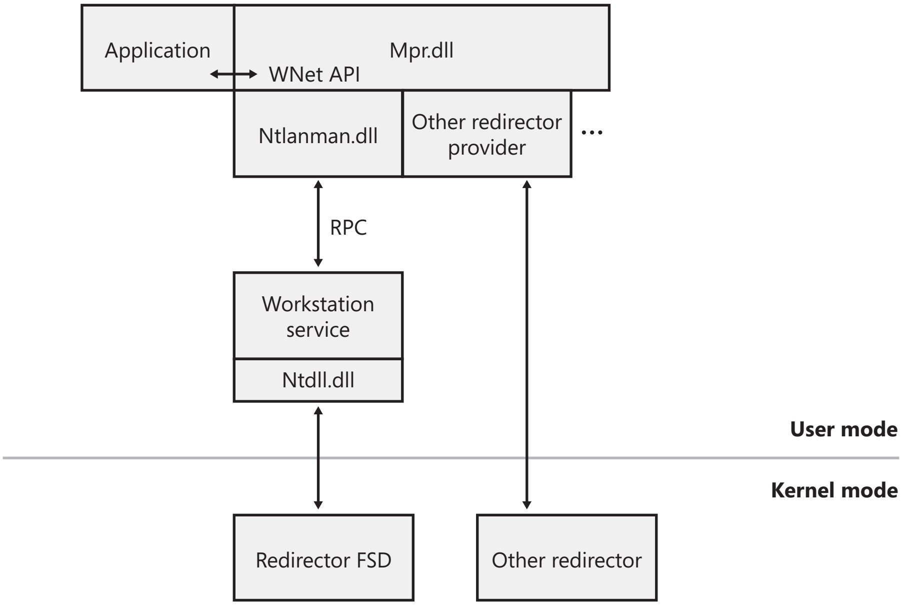
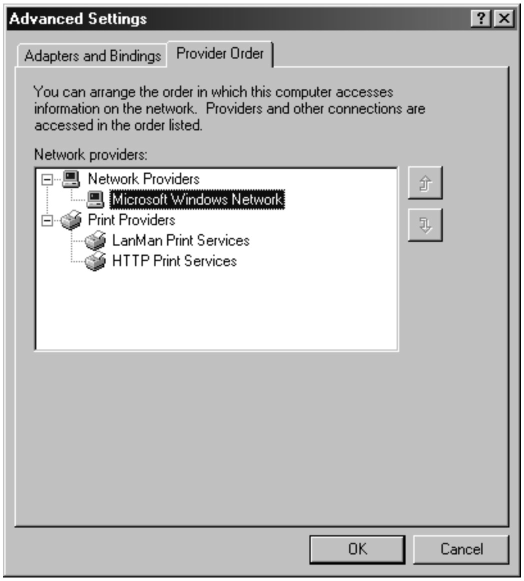


Figure 13-13 MPR components

A *provider* is software that establishes Windows as a client of a remote network server. Some of the operations a WNet provider performs include making and breaking network connections, printing remotely, and transferring data. The built-in WNet provider includes a DLL, the Workstation service, and the redirector. Other network vendors need to supply only a DLL and a redirector.

When an application calls a WNet routine, the call passes directly to the MPR DLL. MPR takes the call and determines which WNet provider recognizes the resource being accessed. Each provider DLL beneath MPR supplies a set of standard functions collectively called the *provider interface*. This interface allows MPR to determine which network the application is trying to access and to direct the request to the appropriate WNet provider software. The Workstation service's provider is \Windows\System32\Ntlanman.dll, as directed by the ProviderPath value under the HKLM\SYSTEM\CurrentControlSet\Services\lanmanworkstation\NetworkProvider registry key.

When called by the *WNetAddConnection* API function to connect to a remote network resource, MPR checks the HKLM\SYSTEM\CurrentControlSet\Control\NetworkProvider\HwOrder\ProviderOrder registry value to determine which network providers are loaded. It polls them one at a time in the order in which they're listed in the registry until a redirector recognizes the resource or until all available providers have been polled. You can change the ProviderOrder by using the Advanced Settings dialog box, shown in Figure 13-14. (Only one provider is installed on the system from which the screen shot was taken.) If you have Windows 2000 or a Start menu configured in its Classic setting, this dialog box is accessible from the Advanced menu of the Network Connections folder. You can access the Network Connections folder by right-clicking the My Network Places icon on the desktop and selecting Properties from the pop-up menu or by selecting it from the Settings option from the Start menu. Another way to access the dialog box is to open Network Connections in the Control Panel, click the connection you want to view, and then click Advanced Settings on the Advanced menu.



You can arrange the order in which this computer accesses information on the network. Providers and other connections are accessed in the order listed.

Network providers:

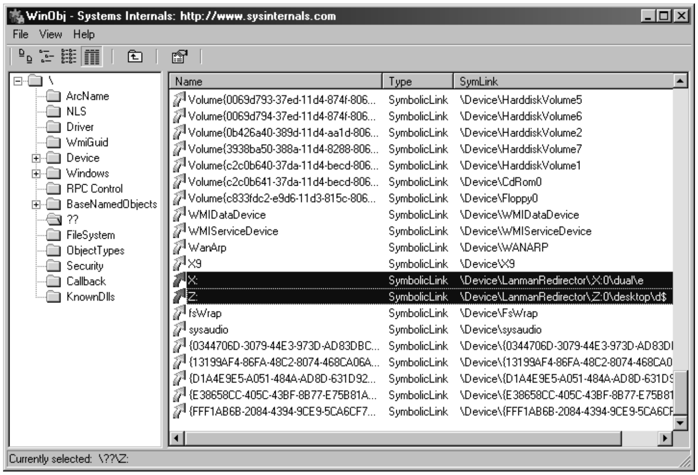
- Network Providers
  - Microsoft Windows Network
- Print Providers
  - LanMan Print Services
  - HTTP Print Services

Figure 13-14 The provider order editor

The *WNetAddConnection* function can also assign a drive letter or device name to a remote resource. When called to do so, *WNetAddConnection* routes the call to the appropriate network provider. The provider, in turn, creates a symbolic-link object in the object manager namespace that maps the drive letter being defined to the redirector (that is, the remote FSD) for that network.

Figure 13-15 shows the \?? directory on a Windows 2000 system (see Chapter 3 for information on the \?? directory), where you can see several driver letters representing connections to remote file shares. The figure shows that the redirector creates a device object named

\Device\LanmanRedirector and that the additional text included in the symbolic link's value indicates to the redirector which remote resource the drive letter corresponds to. When a user opens X:\Book\Chap13.doc, the redirector is passed the unparsed portion of the path that resolves through the symbolic link, which is ";X:0\dual\e\Book\Chap13.doc". The redirector notes that the resource being accessed is located on the E share of server *dual*.



Name	Type	SymLink
Volume{0069d793-37ed-11d4-874f-806...	SymbolicLink	\Device\HarddiskVolume5
Volume{0069d794-37ed-11d4-874f-806...	SymbolicLink	\Device\HarddiskVolume6
Volume{0b426a40-389d-11d4-aa1d-806...	SymbolicLink	\Device\HarddiskVolume2
Volume{3938ba50-388a-11d4-8288-806...	SymbolicLink	\Device\HarddiskVolume7
Volume{c2c0b640-37da-11d4-becd-806...	SymbolicLink	\Device\HarddiskVolume1
Volume{c2c0b641-37da-11d4-becd-806...	SymbolicLink	\Device\CdRom0
Volume{c833fdc2-e9d6-11d3-815c-806...	SymbolicLink	\Device\Floppy0
WMIDataDevice	SymbolicLink	\Device\WMIDataDevice
WMIServiceDevice	SymbolicLink	\Device\WMIServiceDevice
WanArp	SymbolicLink	\Device\WANARP
X9	SymbolicLink	\Device\X9
X:	SymbolicLink	\Device\LanmanRedirector\;X:0\dual\e
Z:	SymbolicLink	\Device\LanmanRedirector\;Z:0\desktop\d$
fsWrap	SymbolicLink	\Device\FsWrap
sysaudio	SymbolicLink	\Device\sysaudio
{0344706D-3079-44E3-973D-AD83DBC...	SymbolicLink	\Device\{0344706D-3079-44E3-973D-AD83D
{13199AF4-86FA-48C2-8074-468CA06A...	SymbolicLink	\Device\{13199AF4-86FA-48C2-8074-468CA0
{D1A4E9E5-A051-484A-AD8D-631D92...	SymbolicLink	\Device\{D1A4E9E5-A051-484A-AD8D-631D
{E38658CC-405C-43BF-8B77-E75B81A...	SymbolicLink	\Device\{E38658CC-405C-43BF-8B77-E75B81
{FFF1AB6B-2084-4394-9CE9-5CA6CF7...	SymbolicLink	\Device\{FFF1AB6B-2084-4394-9CE9-5CA6C

Currently selected: \??\Z:

Figure 13-15 Resolving a network resource name

Like the built-in redirector, other redirectors create a device object in the object manager namespace when they are loaded into the system and initialized. Then, when the WNet or other API calls the object manager to open a resource on a different network, the object manager uses the device object as a jumping-off point into the remote file system. It calls an I/O manager parse method associated with the device object to locate the redirector FSD that can handle the request. (See Chapter 12 for more information on file system drivers.)

Multiple UNC Provider

The Multiple UNC Provider (MUP) is a networking component similar to MPR. It fields I/O requests destined for a file or a device that has a UNC name (names beginning with the characters \\, indicating that the resource exists on the network). MUP takes such requests and, like MPR, determines which local redirector recognizes the remote resource. Unlike MPR, MUP is a device driver (loaded at system boot time) that issues I/O requests to lower-layer drivers, in this case to redirectors, as shown in Figure 13-16. Mup.sys also contains the Distributed File System (DFS is described later in this chapter) client implementation. The DFS client is enabled by default and can be disabled by setting the HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\Mup\DisableDfs DWORD registry value to 1.

When MUP loads during the boot it creates a device object named \Device\Mup. When a network redirector, like CIFS, loads the redirector it creates a named device object (for example \Device\LanmanRedirector) and registers itself with MUP as an UNC provider by calling the function *FsRtlRegisterUncProvider*. If the redirector is the first one to register and MUP's DFS client support is disabled then FsRtlRegisterUncProvider creates a symbolic link name \??\UNC that points to the redirector's device object; otherwise MUP sets up the symbolic link name \Global??\UNC (\??\UNC on Windows 2000) to point to its own device object, \Device\MUP.

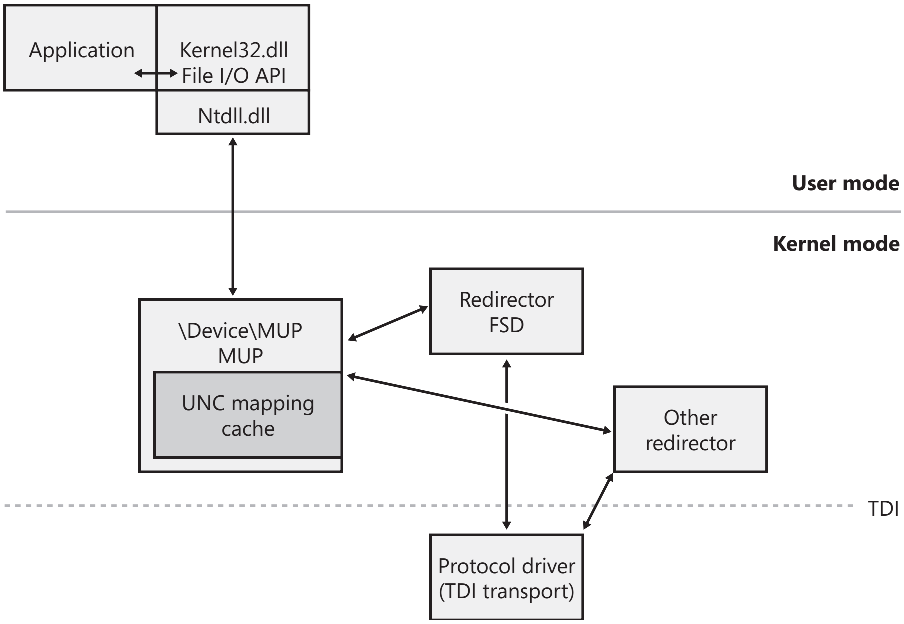


Figure 13-16 Multiple UNC Provider (MUP)

The MUP driver is activated when an application first attempts to open a remote file or device by specifying a UNC name (instead of a redirected drive letter, as described earlier). When the Windows client-side DLL Kernel32.dll (which is the DLL that exports file-I/O-related APIs) receives a file I/O request with UNC paths, the subsystem appends the UNC path to the string \Global??\UNC and then calls the NtCreateFile system service to open the file.

If there is one registered network provider \Global??\UNC resolves to the device object representing the driver and that driver handles the request. If there are multiple registered providers \Global??\UNC resolves to \Device\MUP and MUP must determine which provider should process the request.

When the MUP driver receives an I/O request and the DFS client is enabled MUP first determines if the path references a DFS path (DFS paths are also UNC formatted) and, if so, processes the I/O request itself. If the DFS client is disabled or the path is not a DFS path then MUP uses the HKLM\SYSTEM\CurrentControlSet\Control\NetworkProvider\Order\ProviderOrder registry value to determine priority order of the network providers that registered with it using FsRtlRegisterUncProvider. It polls them one at a time in the order in which they're listed in the value until one of them indicates that it recognizes the path or until all available providers have been polled. MUP ignores those redirectors that are listed in the registry key but not registered. When a redirector recognizes a path the redirector indicates how much of the path is unique to it. For example, if the path is \\WIN2K3SERVER\PUBLIC\Windowsinternals\Chap13.doc, the redirector might recognize it and claim the prefix \\WIN2K3SERVER\PUBLIC as its own. The MUP driver caches this information and thereafter sends requests beginning with that prefix directly to the redirector, skipping the polling operation. The MUP driver's cache has a timeout feature, so after a period, a prefix's association with a particular redirector expires.

Name Resolution

Name Resolution is the process by a character-based name, such as Mycomputer or www.microsoft.com, is translated into a numeric address, such as 192.168.1.1, that the protocol stack can recognize. This section describes the two TCP/IP-related name resolution protocols provided by Windows, Domain Name System (DNS) and Windows Internet Name Service (WINS).

Domain Name System

Domain Name System (DNS) is a standard by which Internet names (such as *www.microsoft.com*) are translated to their corresponding IP addresses. A network application that wants to resolve a DNS name to an IP address sends a DNS lookup request using the TCP/IP protocol to a DNS server. DNS servers implement a distributed database of name/IP address pairs that are used to perform translations, and each server maintains the translations for a particular *zone*. Describing the details of DNS is outside the scope of this book, but DNS is the foundation of naming in Windows and so it is the primary Windows name resolution protocol.

The Windows DNS server is implemented as a Windows service (\Windows\System32\Dns.exe) that is included in server versions of Windows. Standard DNS server implementation relies on a text file as the translation database, but the Windows DNS server can be configured to store zone information in Active Directory.

Windows Internet Name Service

A networking service called *Windows Internet Name Service* (WINS) maintains the mapping between NetBIOS names and IP addresses used for NetBIOS-based TCP/IP applications. If WINS isn't installed, NetBIOS uses local broadcast messages for name operations on a subnet. Note that NetBIOS names are secondary to DNS names for Windows Sockets applications; computer names are registered and resolved first through DNS, with Windows falling back on NetBIOS names only if DNS name resolution fails.

Protocol Drivers

Networking API drivers must take API requests and translate them into low-level network protocol requests for transmission across the network. The API drivers rely on transport protocol drivers in kernel mode to do the actual translation. Separating APIs from underlying protocols gives the networking architecture the flexibility of letting each API use a number of different protocols. The protocol drivers that Windows includes are TCP/IP, TCP/IP with IP version 6 (IPv6), NWLink, and the AppleTalk protocol. Here's a brief description of each protocol:

- The Internet's explosive growth and reliance on the TCP/IP protocol has made TCP/IP the preeminent protocol in Windows. The Defense Advanced Research Projects Agency (DARPA) developed TCP/IP in 1969 specifically as the foundation for the Internet; therefore, TCP/IP has WAN-friendly characteristics such as routability and good WAN performance. TCP/IP is the preferred Windows protocol. It is installed by default and cannot be removed.
- The 4-byte network addresses used by the IPv4 protocol used by the standard TCP/IP stack limits the number of public IP addresses to roughly 4 billion, which is a limit that will be pressed as more and more devices, such as cell phones and PDAs, acquire an Internet presence. For that reason, the IPv6 protocol, which has 16-byte addresses, is gaining adoption. Windows XP (Service Pack 1 and later) and Windows Server 2003 include a TCP/IP stack, \Windows\System32\Drivers\Tcpip6.sys, that implements IPv6. The Windows IPv6 stack also supports coexistence with IPv4 networks, through the use of tunneling.
- NWLink consists of Novell's IPX and SPX protocols. Windows includes NWLink for interoperability with Novell NetWare servers.
- The AppleTalk protocol is used by Apple McIntosh networks, and Windows support for the protocol allows Windows to interoperate with file and print services hosted on AppleTalk-based networks.

TDI transports in Windows generally implement all the protocols associated with their primary protocol. For example, the IPv4 TCP/IP driver (\Windows\System32\Drivers\Tcpip.sys) implements TCP, UDP, IP, ARP, ICMP, and IGMP. A TDI transport generally creates device objects that represent particular protocols so that clients can obtain a file object representing a protocol and issue network I/O to the protocol by using IRPs. The TCP/IP driver creates several device objects that represent various TDI-client accessible protocols: \Device\Tcp, \Device\Udp, \Device\Ip, and on Windows XP and Windows Server 2003, \Device\Rawip and \Device\Ipmulticast.

EXPERIMENT: Looking at TCP/IP's Device Objects

Using the kernel debugger to look at a live system, you can examine TCP/IP's device objects. After performing the *!drvobj* command to see the addresses of each of the driver's device objects, execute *!devobj* to view the name and other details about the device object.

```
lkd> !drvobj tcpip
Driver object (8a01ada0) is for:
 \Driver\Tcpip
Driver Extension List: (id , addr)

Device Object list:
8a0dbc88  8a0dc958  8a0dcd80  8a0eff18
8a0f32a0

lkd> !devobj 8a0dbc88
Device object (8a0dbc88) is for:
 RawIp \Driver\Tcpip DriverObject 8a01ada0
Current Irp 00000000 RefCount 3 Type 00000012 Flags 00000050
Dacl e100d19c DevExt 00000000 DevObjExt 8a0dbd40
ExtensionFlags (0000000000)
Device queue is not busy.

lkd> !devobj 8a0dc958
Device object (8a0dc958) is for:
 Udp \Driver\Tcpip DriverObject 8a01ada0
Current Irp 00000000 RefCount 41 Type 00000012 Flags 00000050
Dacl e100d19c DevExt 00000000 DevObjExt 8a0dca10
ExtensionFlags (0000000000)
Device queue is not busy.

lkd> !devobj 8a0dcd80
Device object (8a0dcd80) is for:
 Tcp \Driver\Tcpip DriverObject 8a01ada0
Current Irp 00000000 RefCount 302 Type 00000012 Flags 00000050
Dacl e1c3a1ac DevExt 00000000 DevObjExt 8a0dce38
ExtensionFlags (0000000000)
Device queue is not busy.

lkd> !devobj 8a0eff18
Device object (8a0eff18) is for:
 IPMULTICAST \Driver\Tcpip DriverObject 8a01ada0
Current Irp 00000000 RefCount 0 Type 00000012 Flags 00000040
Dacl e100d19c DevExt 00000000 DevObjExt 8a0effd0
ExtensionFlags (0000000000)
Device queue is not busy.

lkd> !devobj 8a0f32a0
Device object (8a0f32a0) is for:
 Ip \Driver\Tcpip DriverObject 8a01ada0
Current Irp 00000000 RefCount 48 Type 00000012 Flags 00000050
Dacl e1c3a1ac DevExt 00000000 DevObjExt 8a0f3358
ExtensionFlags (0000000000)   Device queue is not busy.
```

So that networking API drivers don't need to employ various interfaces for each transport protocol they might want to use, Microsoft established the Transport Driver Interface (TDI) standard. As mentioned earlier in this chapter, a TDI interface is essentially a convention for the way network requests format into IRPs and for the way network addresses and communications are allocated. Transport protocols that adhere to the TDI standard export the TDI interface to their clients, which include networking API drivers such as AFD and the redirector. A transport protocol implemented as a Windows device driver is known as a TDI transport. Because TDI transports are device drivers, they format requests they receive from clients as IRPs.

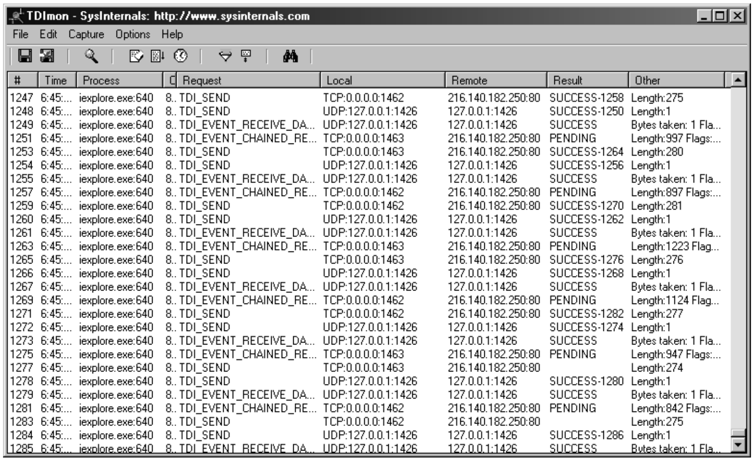
Support functions in the \Windows\System32\Drivers\Tdi.sys library, along with definitions developers include in their drivers, make up the TDI interface. The TDI programming model is very similar to that of Winsock. A TDI client executes the following steps to establish a connection with a remote server:

1. The client allocates and formats an *address open* TDI IRP to allocate an address. The TDI transport returns a file object, which is known as an *address object*, that represents the address. This step is the equivalent of using the *bind* Winsock function.
2. The client then allocates and formats a *connection open* TDI IRP, and the TDI transport returns a file object, which is known as a *connection object*, that represents the connection. This step is the equivalent of the use of the Winsock *socket* function.
3. The client associates the connection object to the address object with an *associate address* TDI IRP. (There's no equivalent to this step in Winsock.)
4. A TDI client that accepts remote connections issues a *listen* TDI IRP specifying the number of connections supported for a connection object and then issues an *accept* TDI IRP, which completes when a remote system establishes a connection (or an error occurs). These operations are equivalent to the use of the Winsock *listen* and *accept* functions.
5. A TDI client that wants to establish a connection with a remote server issues a *connect* TDI IRP, specifying the connection object, that the TDI transport completes when a connection is established (or an error occurs). Issuing a *connect* TDI IRP is the equivalent of using the *connect* Winsock function.

TDI also supports connectionless communications for connectionless protocols such as UDP. In addition, TDI provides a means whereby a TDI client can register *event callbacks* (that is, functions that are directly invoked) with TDI transports. When it receives data from across the network, a TDI transport can invoke a registered client receive callback, for example. This event-based callback feature of TDI allows the TDI transport to notify its clients of network events, and clients that rely on event callbacks don't need to preallocate resources such as buffers when receiving network data because they can view the contents of the buffers supplied by a TDI protocol driver.

EXPERIMENT: Watching TDI Activity

TDImon, a utility from *www.sysinternals.com*, is a form of filter driver that attaches to the \Device\Tcp and \Device\Udp device objects that the TCP/IP driver creates. After attaching, TDImon sees every IRP that TDI clients issue to these protocols. By intercepting TDI client event callback registration, it also monitors event callbacks. The TDImon driver sends information about the TDI activity for display in its GUI, where you can see the time of an operation, the type of TDI activity that took place, the local and remote addresses of a TCP connection or the local address of a UDP endpoint, the resulting status code of the IRP or event callback, and additional information such as the number of bytes sent or received. Here's a screen shot of TDImon watching the TDI activity that is generated when Microsoft Internet Explorer browses a Web page:



#	Time	Process	C	Request	Local	Remote	Result	Other
1247	6:45:...	iexplore.exe:640	8..	TDI_SEND	TCP:0.0.0.0:1462	216.140.182.250:80	SUCCESS-1258	Length:275
1248	6:45:...	iexplore.exe:640	8..	TDI_SEND	UDP:127.0.0.1:1426	127.0.0.1:1426	SUCCESS-1250	Length:1
1249	6:45:...	iexplore.exe:640	8..	TDI_EVENT_RECEIVE_DA...	UDP:127.0.0.1:1426	127.0.0.1:1426	SUCCESS	Bytes taken: 1 Fla...
1251	6:45:...	iexplore.exe:640	8..	TDI_EVENT_CHAINED_RE...	TCP:0.0.0.0:1463	216.140.182.250:80	PENDING	Length:997 Flags:...
1253	6:45:...	iexplore.exe:640	8..	TDI_SEND	TCP:0.0.0.0:1463	216.140.182.250:80	SUCCESS-1264	Length:280
1254	6:45:...	iexplore.exe:640	8..	TDI_SEND	UDP:127.0.0.1:1426	127.0.0.1:1426	SUCCESS-1256	Length:1
1255	6:45:...	iexplore.exe:640	8..	TDI_EVENT_RECEIVE_DA...	UDP:127.0.0.1:1426	127.0.0.1:1426	SUCCESS	Bytes taken: 1 Fla...
1257	6:45:...	iexplore.exe:640	8..	TDI_EVENT_CHAINED_RE...	TCP:0.0.0.0:1462	216.140.182.250:80	PENDING	Length:897 Flags:...
1259	6:45:...	iexplore.exe:640	8..	TDI_SEND	TCP:0.0.0.0:1462	216.140.182.250:80	SUCCESS-1270	Length:281
1260	6:45:...	iexplore.exe:640	8..	TDI_SEND	UDP:127.0.0.1:1426	127.0.0.1:1426	SUCCESS-1262	Length:1
1261	6:45:...	iexplore.exe:640	8..	TDI_EVENT_RECEIVE_DA...	UDP:127.0.0.1:1426	127.0.0.1:1426	SUCCESS	Bytes taken: 1 Fla...
1263	6:45:...	iexplore.exe:640	8..	TDI_EVENT_CHAINED_RE...	TCP:0.0.0.0:1463	216.140.182.250:80	PENDING	Length:1223 Flag...
1265	6:45:...	iexplore.exe:640	8..	TDI_SEND	TCP:0.0.0.0:1463	216.140.182.250:80	SUCCESS-1276	Length:276
1266	6:45:...	iexplore.exe:640	8..	TDI_SEND	UDP:127.0.0.1:1426	127.0.0.1:1426	SUCCESS-1268	Length:1
1267	6:45:...	iexplore.exe:640	8..	TDI_EVENT_RECEIVE_DA...	UDP:127.0.0.1:1426	127.0.0.1:1426	SUCCESS	Bytes taken: 1 Fla...
1269	6:45:...	iexplore.exe:640	8..	TDI_EVENT_CHAINED_RE...	TCP:0.0.0.0:1462	216.140.182.250:80	PENDING	Length:1124 Flag...
1271	6:45:...	iexplore.exe:640	8..	TDI_SEND	TCP:0.0.0.0:1462	216.140.182.250:80	SUCCESS-1282	Length:277
1272	6:45:...	iexplore.exe:640	8..	TDI_SEND	UDP:127.0.0.1:1426	127.0.0.1:1426	SUCCESS-1274	Length:1
1273	6:45:...	iexplore.exe:640	8..	TDI_EVENT_RECEIVE_DA...	UDP:127.0.0.1:1426	127.0.0.1:1426	SUCCESS	Bytes taken: 1 Fla...
1275	6:45:...	iexplore.exe:640	8..	TDI_EVENT_CHAINED_RE...	TCP:0.0.0.0:1463	216.140.182.250:80	PENDING	Length:947 Flags:...
1277	6:45:...	iexplore.exe:640	8..	TDI_SEND	TCP:0.0.0.0:1463	216.140.182.250:80		Length:274
1278	6:45:...	iexplore.exe:640	8..	TDI_SEND	UDP:127.0.0.1:1426	127.0.0.1:1426	SUCCESS-1280	Length:1
1279	6:45:...	iexplore.exe:640	8..	TDI_EVENT_RECEIVE_DA...	UDP:127.0.0.1:1426	127.0.0.1:1426	SUCCESS	Bytes taken: 1 Fla...
1281	6:45:...	iexplore.exe:640	8..	TDI_EVENT_CHAINED_RE...	TCP:0.0.0.0:1462	216.140.182.250:80	PENDING	Length:842 Flags:...
1283	6:45:...	iexplore.exe:640	8..	TDI_SEND	TCP:0.0.0.0:1462	216.140.182.250:80		Length:275
1284	6:45:...	iexplore.exe:640	8..	TDI_SEND	UDP:127.0.0.1:1426	127.0.0.1:1426	SUCCESS-1286	Length:1
1285	6:45:...	iexplore.exe:640	8..	TDI_EVENT_RECEIVE_DA...	UDP:127.0.0.1:1426	127.0.0.1:1426	SUCCESS	Bytes taken: 1 Fla...

As evidence that TDI operations are inherently asynchronous, the PENDING codes in the Result column indicate that an operation initiated but that the IRP defining the operation hasn't yet completed. To accurately show the order of completions with respect to the start of other operations, the issuing of each IRP or event callback is tagged with a sequence number. If other IRPs are issued or completed before an IRP completes, the IRP's completion is also tagged with a sequence number that is shown in the Result column. For example, the IRP that was tagged with sequence number 1278 in the screen shot completed after the one tagged with 1279 was issued, so you see 1280 in the Result column for IRP 1278.

TCP/IP Extensions

Other Windows networking services extend basic networking features of the TCP/IP protocol driver by relying on add-on drivers that integrate with the TCP/IP protocol driver using private interfaces. These include network address translation (NAT), IP filtering, IP hooking, and Internet Protocol security (IPSec). Figure 13-17 shows how these extensions interface with the TCP/IP driver.

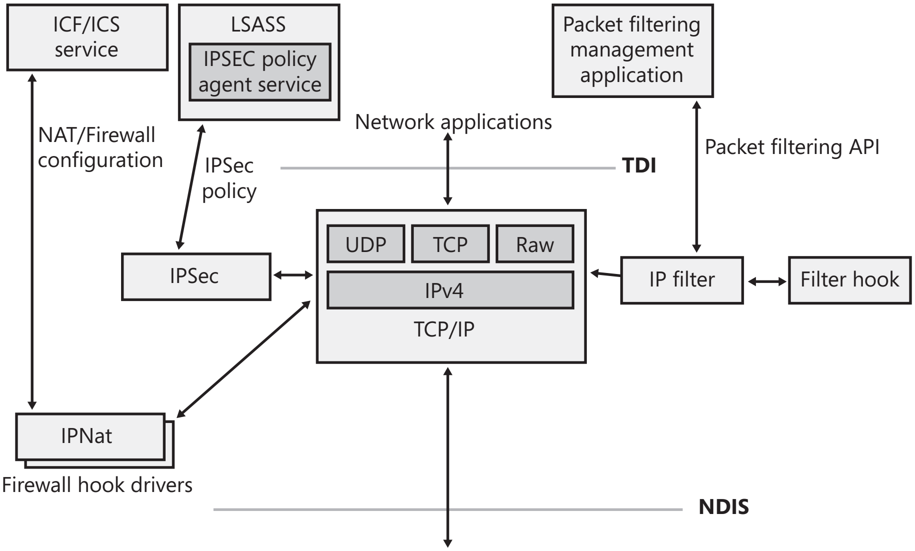


Figure 13-17 TCP/IP extension architecture

Network Address Translation

Network address translation (NAT) is a routing service that allows multiple private IP addresses to map to a single public IP address. Without NAT, each computer of a LAN must be assigned a public IP address to communicate across the Internet. NAT allows one computer of the LAN to be assigned an IP address and the other computers to use private IP address and be connected to the Internet through that computer. NAT translates between private IP addresses and the public IP address as necessary, routing packets between LAN computers and the Internet.

NAT components on Windows consist of a NAT device driver, \Windows\System32\Drivers\Ipnat.sys, that interfaces with the TCP/IP stack as well as editors that can perform additional packet processing beyond address and port translation. NAT can be installed as a routing protocol component with the Routing And Remote Access MMC snap-in or by configuring Internet Connection Sharing (ICS) using the Network Connections folder (although NAT is much more configurable when installed using the Routing And Remote Access MMC snap-in).

IP Filtering

Windows 2000, Windows XP, and Windows Server 2003 all include a very basic IP filtering capability where a user can choose to allow only certain ports or IP protocols. While this capability can serve to protect a computer from unauthorized network accesses, its drawback is that it is static and does not automatically create new filters for traffic initiated by applications running on the computer.

Windows XP introduces a personal firewall capability called Internet Connection Firewall (ICF) that goes beyond the basic filtering just described. ICF implements a *stateful firewall*, which is one that keeps track of traffic flow so that it distinguishes between TCP/IP traffic that originates on the local LAN and unsolicited traffic that originates on the Internet. When ICF is enabled on an interface, by default all unsolicited incoming traffic received by the computer is discarded. A user or application can define exceptions so that services running on the computer, such as file and printer sharing or a Web site, can be accessed from other computers.

The ICF/ICS service, which executes in a Svchost process, passes exception rules defined in the ICF user interface to the IPNat driver. In kernel mode, ICF is implemented in the same driver, \Windows\System32\Drivers\Ipnat.Sys, that implements Network Address Translation. The NAT driver registers with the TCP/IP driver as a *firewall hook* driver. The TCP/IP driver executes the callback functions of each registered firewall hook as it processes both inbound and outbound IP packets. A callback function can provide NAT functionality by modifying source and destination addresses in a packet, or as a firewall by returning a status code to TCP/IP that requests that TCP/IP drop the packet and cease processing for it.

Although Ipnat implements ICF using the TCP/IP firewall hook interface, Microsoft recommends that third parties implement packet filtering as an NDIS Intermediate Driver (described later in this chapter).

IP Filter and Filter Hook

Windows XP and Windows Server 2003 include a user-mode packet filtering API and an IP filter driver, \Windows\System32\Drivers\Ipfltrdrv.sys, that together allow applications to manage packet flow into and out of a system. The IP filter driver also allows at most one driver to register as a *filter hook* driver. In a manner similar to the way it interacts with firewall hook drivers, TCP/IP executes a function that the IP filter driver specifies, which allows the IP filter to drop or modify packets. The IP filter in turn invokes a callback specified by the filter hook driver, propagating modifications or a request to drop the packet to the TCP/IP driver.

The filter hook functionality provided by the system allows third-parties to add translation, firewall, logging, or other functionality to the system.

Internet Protocol Security

Internet Protocol security (IPSec), which is integrated with the Windows TCP/IP stack, helps protect unicast IP data against attacks such as eavesdropping, sniffer attacks, data modification, IP address spoofing, and man-in-the-middle attacks. You can use IPSec to provide defense-in-depth against network-based attacks from untrusted computers, attacks which can result in the denial-of-service of applications, services, or the network; data corruption, data theft, user-credential theft; and the administrative control over servers, other computers, and the network. IPSec helps defend against network-based attacks through cryptography-based security services, security protocols, and dynamic key management.

IPSec provides the following properties for unicast IP packets sent between trusted hosts:

- Data origin authentication, which verifies the origin of an IP packet and ensures that unauthenticated parties cannot access data.
- Data integrity, which protects an IP packet from being modified in transit without being detected.
- Data confidentiality, which encrypts the payload of IP packets before transmission. Data confidentiality ensures that only the IPSec peer with which a computer is communicating can read and interpret the contents of the packets.
- Anti-replay (or replay protection), which ensures that each IP packet is unique and can't be reused. This property prevents an attacker from intercepting IP packets and inserting modified packets into a data stream between a source computer and a destination computer. When anti-replay is used, attackers cannot reply to captured messages to establish a session or gain unauthorized access to data.

You can use IPSec to help defend against network-based attacks by configuring host-based IPSec packet filtering and enforcing trusted communications. When you use IPSec for host-based IPSec packet filtering, IPSec can permit or block specific types of unicast IP traffic based on source and destination address combinations and specific protocols and specific ports. Because IPSec is integrated at the IP layer (layer 3) of the TCP/IP stack and it is applied to all applications, you do not need to configure separate security for each application that uses TCP/IP.

In an Active Directory environment, Group Policy can be used to configure domains, sites, and organizational units (OUs), and IPSec policies can then be assigned as required to Group Policy Objects (GPOs). Alternatively, you can configure and assign local IPSec policies. Active Directory-based IPSec policies are stored in Active Directory, and a copy of the current policy is maintained in a cache in the local Registry. Local IPSec policies are stored in the local system registry.

To establish trusted communications, IPSec uses mutual authentication, and it supports the following authentication methods: Kerberos version 5, a computer X.509 version 3 public key certificate, or a preshared key.

The Windows implementation of IPSec is based on IPSec Requests for Comments (RFCs). The Windows IPSec architecture includes the IPSec Policy Agent, the Internet Key Exchange (IKE) protocol, and an IPSec driver:

- **IPSec Policy Agent** The IPSec Policy Agent runs as a service in the Local Security Authority Subsystem process. (See Chapter 8 for information on LSASS.) In the Services snap-in in the Microsoft Management Console (MMC), the IPSec Policy Agent appears in the list of computer services under the name IPSEC services. The IPSec Policy Agent obtains IPSec policy from an Active Directory domain or the local registry, and then passes IP address filters to the IPSec driver and authentication and security settings to IKE.

- **IKE** IKE waits for requests to negotiate security associations (SAs) from the IPSec driver, negotiates the SAs, and then sends the SA settings back to the IPSec driver. SAs are a combination of mutually agreeable IPSec policy settings and keys that defines the security services, mechanisms, and keys that are used to help secure communications between IPSec peers. Each SA is a one-way or simplex connection that secures the traffic it carries. IKE negotiates main mode SAs and quick mode SAs when requested by the IPSec driver. The IKE main mode (or ISAKMP) SA protects the IKE negotiation. The quick mode (or IPSec) SAs protect application traffic.
- **IPSec driver** The IPSec driver is a device driver (\Windows\System32\Drivers\Ipsec.sys) that is bound to the TCP/IP driver and that processes packets that pass through the TCP/IP driver. The IPSec driver monitors and secures outbound unicast IP traffic, and it monitors, decrypts, and validates inbound unicast IP packets. The driver receives filters from the IPSec Policy Agent, and then permits, blocks, or secures packets as required. To secure traffic, the IPSec driver uses active SA settings, or it requests that new SAs be created.

You can use the IP Security Policy Management snap-in that is available in MMC to create and manage IPSec policy. This snap-in can be used to create, modify, and store local IPSec policies or Active Directory–based IPSec policies, and to modify IPSec policy on remote computers. In Windows XP and Windows Server 2003, after IPSec-secured communication is established, you can monitor IPSec information for local computers and for remote computers by using the IP Security Monitor snap-in that is available in MMC.

NDIS Drivers

When a protocol driver wants to read or write messages formatted in its protocol's format from or to the network, the driver must do so using a network adapter. Because expecting protocol drivers to understand the nuances of every network adapter on the market (proprietary network adapters number in the thousands) isn't feasible, network adapter vendors provide device drivers that can take network messages and transmit them via the vendors' proprietary hardware. In 1989, Microsoft and 3Com jointly developed the Network Driver Interface Specification (NDIS), which lets protocol drivers communicate with network adapter drivers in a device-independent manner. Network adapter drivers that conform to NDIS are called NDIS drivers or NDIS miniport drivers. The version of NDIS that ships with Windows 2000 is NDIS 5, and the version that ships with Windows XP and Windows Server 2003 is NDIS 5.1.

The NDIS library (\Windows\System32\Drivers\Ndis.sys) implements the NDIS boundary that exists between TDI transports (typically) and NDIS drivers. As is Tdi.sys, the NDIS library is a helper library that NDIS driver clients use to format commands they send to NDIS drivers. NDIS drivers interface with the library to receive requests and send back responses. Figure 13-18 shows the relationship between various NDIS-related components.

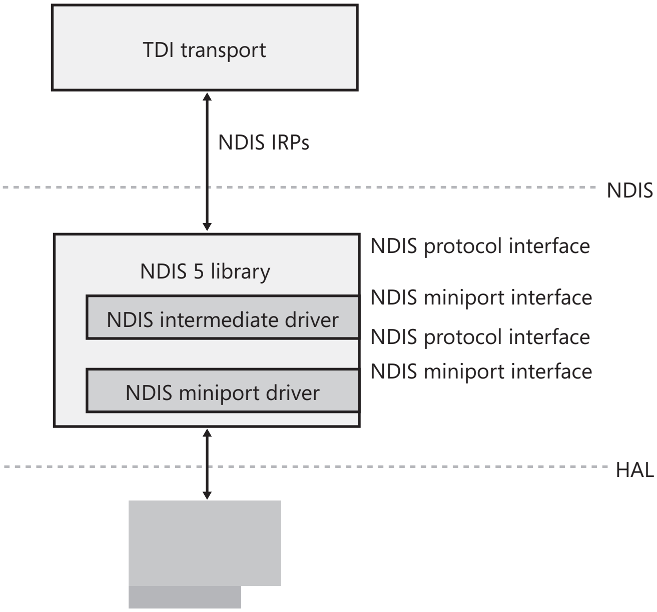


Figure 13-18 NDIS components

One of Microsoft's goals for its network architecture was to let network adapter vendors easily develop NDIS drivers and take driver code and move it between Consumer Windows and Windows 2000. Thus, instead of merely providing the NDIS boundary helper routines, the NDIS library provides NDIS drivers an entire execution environment. NDIS drivers aren't genuine Windows drivers because they can't function without the encapsulation the NDIS library gives them. This insulation layer wraps NDIS drivers so thoroughly that NDIS drivers don't accept and process IRPs. Rather, TDI protocol drivers call a function in the NDIS library, *NdisAllocatePacket*, and pass the packets to an NDIS miniport by calling an NDIS library function (*NdisSend*, for example). By default, NDIS drivers also don't have to worry about reentrancy, in which the NDIS library invokes an NDIS driver with a new request before the driver has finished servicing a previous request. Exemption from reentrancy means that NDIS driver writers don't need to worry about complex synchronization, which is made even more tricky because of the parallel execution possible on a multiprocessor.

Note The NDIS library uses NDIS packets to represent I/O requests instead of IRPs. TDI transports allocate an NDIS packet by calling *NdisAllocatePacket* and then pass the packet to an NDIS miniport by calling an NDIS library function (*NdisSend*, for example).

Although the NDIS library's serialization of NDIS drivers simplifies development, serialization can hamper multiprocessor scalability. Standard NDIS 4 drivers (the Windows NT 4 version of the NDIS library) don't scale well for certain operations on multiprocessors. Microsoft gave developers a deserialized operation option in NDIS 5. NDIS 5 drivers can indicate to the NDIS library that they don't want to be serialized; the NDIS library will then forward requests to the driver as fast as it receives the IRPs that describe the requests. Responsibility for queuing and managing multiple simultaneous requests falls on the NDIS driver, but deserialization confers the benefit of higher multiprocessor performance.

NDIS 5 also includes the following features:

- NDIS drivers can report whether or not their network medium is active, which allows Windows to display a network connected/disconnected icon on the taskbar. This feature also allows protocols and other applications to be aware of this state and react accordingly. The TCP/IP transport, for example, will use this information to determine when it should reevaluate addressing information it receives from DHCP.
- TCP/IP task offloading allows a miniport to use advanced features of a network adapter to perform operations such as packet checksums and Internet Protocol security (IPSec). This task offloading can improve system performance by relieving the CPU from these operations.
- Wake-on-LAN allows a wake-on-LAN-capable network adapter to bring Windows out of a suspend power state. Events that can trigger the network adapter to signal the system include media connections (such as plugging a network cable into the adapter), the receipt of protocol-specific patterns registered by a protocol (the TCP/IP transport asks to be woken for Address Resolution Protocol [ARP] requests), and, for Ethernet adapters, the receipt of a *magic* packet (a network packet that contains 16 contiguous copies of the adapter's Ethernet address).
- Connection-oriented NDIS allows NDIS drivers to manage connection-oriented media such as Asynchronous Transfer Mode (ATM) devices. (Connection-oriented NDIS is described in more detail shortly.)

The interfaces that the NDIS library provides for NDIS drivers to interface with network adapter hardware are available via functions that translate directly to corresponding functions in the HAL.

EXPERIMENT: Listing the Loaded NDIS Miniports

The Ndiskd kernel debugger extension library includes the *!miniports* and *!miniport* commands, which let you list the loaded miniports using a kernel debugger and, given the address of a miniport block (a data structure Windows uses to track miniports), see detailed information about the miniport driver. The following example shows the *!miniports* and *!miniport* commands being used to list all the miniports and then specifics about the miniport responsible for interfacing the system to a PCI Ethernet adapter. (Note that WAN miniport drivers work with dial-up connections.)

```
kd> .load ndiskd
Loaded ndiskd extension DLL

kd> !miniports
Driver verifier level: 0
Failed allocations: 0
Miniport Driver Block: 817aa610
```

```
  Miniport: 817b1130 RAS Async Adapter
Miniport Driver Block: 81a1ef30
  Miniport: 81a1ea70 Direct Parallel
Miniport Driver Block: 81a21cd0
  Miniport: 81a217f0 WAN Miniport (PPTP)
Miniport Driver Block: 81a23290
  Miniport: 81a22130 WAN Miniport (L2TP)
Miniport Driver Block: 81a275f0
  Miniport: 81a25130 Intel 8255x-based PCI Ethernet Adapter (10/100)

kd> !miniport 81a25130
 Miniport 81a25130 : Intel 8255x-based PCI Ethernet Adapter (10/100)
    Flags          : 20413208
                     BUS_MASTER, INDICATES_PACKETS, IGNORE_REQUEST_QUEUE
                     IGNORE_TOKEN_RING_ERRORS, NDIS_5_0,
                     RESOURCES_AVAILABLE, DESERIALIZED, MEDIA_CONNECTED,
                     NOT_SUPPORTS_MEDIA_SENSE,
    PnPFlags       : 00010021
                     PM_SUPPORTED, DEVICE_POWER_ENABLED, RECEIVED_START
    CheckforHang interval: 2 seconds
        CurrentTick   : 0001
        IntervalTicks : 0001
    InternalResetCount : 0000
    MiniportResetCount : 0000

    References: 3
    UserModeOpenReferences: 0

    PnPDeviceState : PNP_DEVICE_STARTED
    CurrentDevicePowerState : PowerDeviceD0
    Bus PM capabilities
DeviceD1:1
DeviceD2:1
WakeFromD0:0
WakeFromD1:1
WakeFromD2:0
WakeFromD3:0

SystemStateDeviceState
PowerSystemUnspecifiedPowerDeviceUnspecified
S0D0
S1D1
S2PowerDeviceUnspecified
S3PowerDeviceUnspecified
S4D3
S5D3
SystemWake: S1
DeviceWake: D1

WakeupMethodes Enabled 6:
WAKE_UP_PATTERN_MATCH  WAKE_UP_LINK_CHANGE
WakeUpCapabilities of the miniport
MinMagicPacketWakeUp: 4
MinPatternWakeUp: 4
MinLinkChangeWakeUp: 4
```

```
Current PnP and PM Settings:          : 00000030
                 DISABLE_WAKE_UP, DISABLE_WAKE_ON_RECONNECT,

Allocated Resources:
    Memory: f4100000, Length: 1000
    IO Port: 1440, Length: 40
    Memory: f4000000, Length: 100000
    Interrupt Level: 9, Vector: 9
Translated Allocated Resources:
    Memory: f4100000, Length: 1000
    IO Port: 1440, Length: 40
    Memory: f4000000, Length: 100000
    Interrupt Level: 12, Vector: 39
MediaType      : 802.3
DeviceObject   : 81a25030, PhysDO : 81a93cd0  Next DO: 81a63030
MapRegisters   : 819fc000
FirstPendingPkt: 0
SingleWorkItems:
  [0]: 81a254e8 [1]: 81a254f4 [2]: 81a25500 [3]: 81a2550c
  [4]: 81a25518 [5]: 81a25524
DriverVerifyFlags       : 00000000
Miniport Open Block Queue:
  8164b888: Protocol 816524a8 = NBF, ProtocolContext 81649030
  8191f628: Protocol 81928d88 = TCPIP, ProtocolContext 8191f728
 Miniport Interrupt 81a00970
```

The Flags field for the miniport that was examined indicates that the miniport supports deserialized operation (DESERIALIZED), that the media is currently active (MEDIA_CONNECTED), and that it is an NDIS 5 miniport driver (NDIS_5_0). Also listed are the adapter's system-to-device power-state mappings and the bus resources that the Plug and Play manager assigned to the adapter. (See the section "The Power Manager" in Chapter 9 for more information on power-state mappings.)

Variations on the NDIS Miniport

The NDIS model also supports hybrid TDI transport-NDIS drivers, called *NDIS intermediate drivers*. These drivers lie between TDI transports and NDIS drivers. To an NDIS driver, an NDIS intermediate driver looks like a TDI transport; to a TDI transport, an NDIS intermediate driver looks like an NDIS driver. NDIS intermediate drivers can see all network traffic taking place on a system because the drivers lie between protocol drivers and network drivers. Software that provides fault tolerant and load balancing options for network adapters, such as Microsoft's Network Load Balancing Provider, are based on NDIS intermediate drivers.

Connection-Oriented NDIS

Support for connection-oriented network hardware (for example, ATM) is native in Windows, which makes connection management and establishment standard in the Windows network architecture. Connection-oriented NDIS drivers use many of the same APIs that standard

NDIS drivers use; however, connection-oriented NDIS drivers send packets through established network connections rather than place them on the network medium.

In addition to miniport support for connection-oriented media, NDIS 5 includes definitions for drivers that work to support a connection-oriented miniport driver:

- Call managers are NDIS drivers that provide call setup and teardown services for connection-oriented clients (described shortly). A call manager uses a connection-oriented miniport to exchange signaling messages with other network entities such as network switches or other call managers. A call manager supports one or more signaling protocols, such as ATM User-Network Interface (UNI) 3.1.
- An integrated miniport call manager (MCM) is a connection-oriented miniport driver that also provides call manager services to connection-oriented clients. An MCM is essentially an NDIS miniport driver with a built-in call manager.
- A connection-oriented client uses the call setup and teardown services of a call manager or MCM and the send and receive services of a connection-oriented NDIS miniport driver. A connection-oriented client can provide its own protocol services to higher levels in the network stack, or it can implement an emulation layer that interfaces connectionless legacy protocols and connection-oriented media. An example of an emulation layer fulfilled by a connection-oriented client is a LAN emulation (LANE), which hides the connected-oriented characteristics of ATM and presents a connectionless media (such as Ethernet) to protocols above it.

Figure 13-19 shows the relationships between these components.

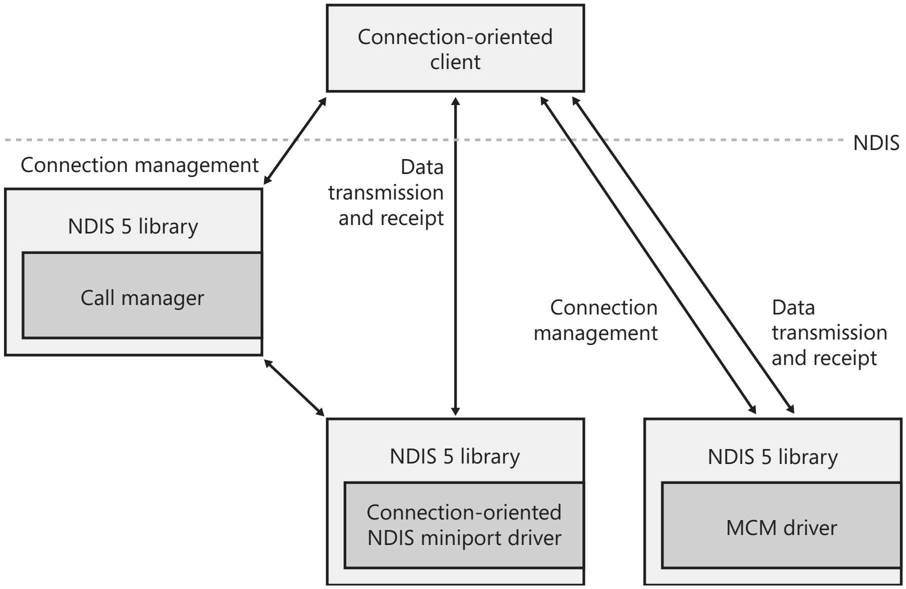
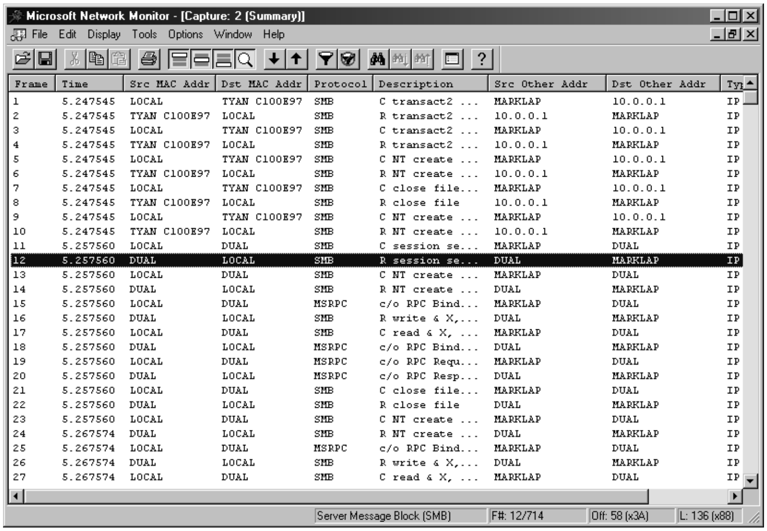

Connection-oriented client
NDIS
Connection management
Data transmission and receipt
NDIS 5 library
Call manager
Connection management
Data transmission and receipt
NDIS 5 library
Connection-oriented NDIS miniport driver
NDIS 5 library
MCM driver

Figure 13-19 Connection-oriented NDIS drivers

EXPERIMENT: Using Network Monitor to Capture Network Packets

Windows Server comes with a tool named Network Monitor that lets you capture packets that flow through one or more NDIS miniport drivers on your system by installing an NDIS intermediate driver. Before you can use Network Monitor, you need to have the Windows Network Monitor Tools installed on your system. To install these tools, open Add/Remove Programs in Control Panel, and select Add/Remove Windows Components. Select Management And Monitoring Tools, click Details, select Network Monitor Tools, and click OK. After Network Monitor has been installed, you can launch Network Monitor by selecting it from the Administrative Tools folder.

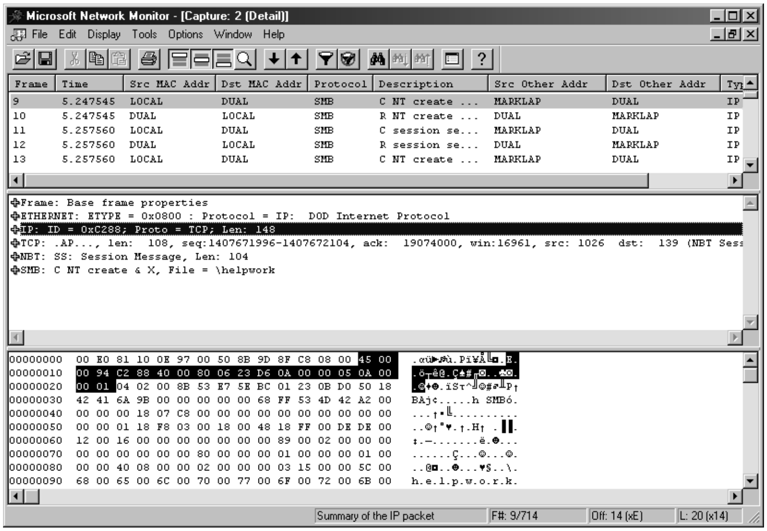
Network Monitor might ask you which network connection you want to monitor. After selecting one, begin monitoring by pressing the Start Capture button in the toolbar. Perform operations that generate network activity on the connection you're monitoring, and after you see that Network Monitor has captured packets, stop monitoring by clicking the Stop And View Capture button (the stop button that has glasses next to it). Network Monitor will present a view of the capture data like the following:



Frame	Time	Src MAC Addr	Dst MAC Addr	Protocol	Description	Src Other Addr	Dst Other Addr	Typ
1	5.247545	LOCAL	TYAN C100E97	SMB	C transact2 ...	MARKLAP	10.0.0.1	IP
2	5.247545	TYAN C100E97	LOCAL	SMB	R transact2 ...	10.0.0.1	MARKLAP	IP
3	5.247545	LOCAL	TYAN C100E97	SMB	C transact2 ...	MARKLAP	10.0.0.1	IP
4	5.247545	TYAN C100E97	LOCAL	SMB	R transact2 ...	10.0.0.1	MARKLAP	IP
5	5.247545	LOCAL	TYAN C100E97	SMB	C NT create ...	MARKLAP	10.0.0.1	IP
6	5.247545	TYAN C100E97	LOCAL	SMB	R NT create ...	10.0.0.1	MARKLAP	IP
7	5.247545	LOCAL	TYAN C100E97	SMB	C close file...	MARKLAP	10.0.0.1	IP
8	5.247545	TYAN C100E97	LOCAL	SMB	R close file	10.0.0.1	MARKLAP	IP
9	5.247545	LOCAL	TYAN C100E97	SMB	C NT create ...	MARKLAP	10.0.0.1	IP
10	5.247545	TYAN C100E97	LOCAL	SMB	R NT create ...	10.0.0.1	MARKLAP	IP
11	5.257560	LOCAL	DUAL	SMB	C session se...	MARKLAP	DUAL	IP
12	5.257560	DUAL	LOCAL	SMB	R session se...	DUAL	MARKLAP	IP
13	5.257560	LOCAL	DUAL	SMB	C NT create ...	MARKLAP	DUAL	IP
14	5.257560	DUAL	LOCAL	SMB	R NT create ...	DUAL	MARKLAP	IP
15	5.257560	LOCAL	DUAL	MSRPC	c/o RPC Bind...	MARKLAP	DUAL	IP
16	5.257560	DUAL	LOCAL	SMB	R write & X,...	DUAL	MARKLAP	IP
17	5.257560	LOCAL	DUAL	SMB	C read & X, ...	MARKLAP	DUAL	IP
18	5.257560	DUAL	LOCAL	MSRPC	c/o RPC Bind...	DUAL	MARKLAP	IP
19	5.257560	LOCAL	DUAL	MSRPC	c/o RPC Requ...	MARKLAP	DUAL	IP
20	5.257560	DUAL	LOCAL	MSRPC	c/o RPC Resp...	DUAL	MARKLAP	IP
21	5.257560	LOCAL	DUAL	SMB	C close file...	MARKLAP	DUAL	IP
22	5.257560	DUAL	LOCAL	SMB	R close file	DUAL	MARKLAP	IP
23	5.257560	LOCAL	DUAL	SMB	C NT create ...	MARKLAP	DUAL	IP
24	5.267574	DUAL	LOCAL	SMB	R NT create ...	DUAL	MARKLAP	IP
25	5.267574	LOCAL	DUAL	MSRPC	c/o RPC Bind...	MARKLAP	DUAL	IP
26	5.267574	DUAL	LOCAL	SMB	R write & X,...	DUAL	MARKLAP	IP
27	5.267574	LOCAL	DUAL	SMB	C read & X, ...	MARKLAP	DUAL	IP

Server Message Block (SMB) | F#: 12/714 | Off: 58 (x3A) | L: 136 (x88)

The screen shot shows the SMB (CIFS) packets that Network Monitor captured as remote files were accessed from the system. If you double-click on a line, Network Monitor will switch to a view of the packet that breaks it apart to show various layered application and protocol headers as shown in the following screen shot:



```
Frame  Time      Src MAC Addr  Dst MAC Addr  Protocol  Description      Src Other Addr  Dst Other Addr  Typ
9      5.247545  LOCAL         DUAL          SMB       C NT create ...  MARKLAP         DUAL            IP
10     5.247545  DUAL          LOCAL         SMB       R NT create ...  DUAL            MARKLAP         IP
11     5.257560  LOCAL         DUAL          SMB       C session se...  MARKLAP         DUAL            IP
12     5.257560  DUAL          LOCAL         SMB       R session se...  DUAL            MARKLAP         IP
13     5.257560  LOCAL         DUAL          SMB       C NT create ...  MARKLAP         DUAL            IP
```

```
+Frame: Base frame properties
+ETHERNET: ETYPE = 0x0800 : Protocol = IP:  DOD Internet Protocol
+IP: ID = 0xC288; Proto = TCP; Len: 148
+TCP: .AP..., len:  108, seq:1407671996-1407672104, ack:  19074000, win:16961, src: 1026  dst:  139 (NBT Sess
+NBT: SS: Session Message, Len: 104
+SMB: C NT create & X, File = \helpwork
```

```
00000000  00 E0 81 10 0E 97 00 50 8B 9D 8F C8 08 00 45 00   .αü►♬ù.Pï¥Å╙■.E.
00000010  00 94 C2 88 40 00 80 06 23 D6 0A 00 00 05 0A 00   .ö┬ê@.Ç♠#╓◙..♣◙.
00000020  00 01 04 02 00 8B 53 E7 5E BC 01 23 0B D0 50 18   .☺♦☻.ïSτ^╝☺#♂╨P↑
00000030  42 41 6A 9B 00 00 00 00 00 68 FF 53 4D 42 A2 00   BAj¢.....h SMBó.
00000040  00 00 00 18 07 C8 00 00 00 00 00 00 00 00 00 00   ...↑•╚..........
00000050  00 00 01 18 F8 03 00 18 00 48 18 FF 00 DE DE 00   ..☺↑°♥.↑.H↑ .▐▐.
00000060  12 00 16 00 00 00 00 00 00 00 89 00 02 00 00 00   ↕.▬.......ë.☻...
00000070  00 00 00 00 00 00 80 00 00 00 01 00 00 00 01 00   ......Ç...☺...☺.
00000080  00 00 40 08 00 00 02 00 00 00 03 15 00 00 5C 00   ..@◘..☻...♥§..\.
00000090  68 00 65 00 6C 00 70 00 77 00 6F 00 72 00 6B 00   h.e.l.p.w.o.r.k.
```

Summary of the IP packet | F#: 9/714 | Off: 14 (xE) | L: 20 (x14)

Network Monitor also includes a number of other features, such as capture triggers and filters, that make it a powerful tool for troubleshooting network problems.

Remote NDIS

Prior to the development of Remote NDIS, a vendor that developed a USB network device, for example, had to provide a driver that interfaced with NDIS as a miniport driver as well as interfacing with a USB WDM bus driver, as shown in Figure 13-20. If the vendor's hardware supported other busses, such as IEEE 1394, they were required to implement drivers that interfaced with each specific bus type.

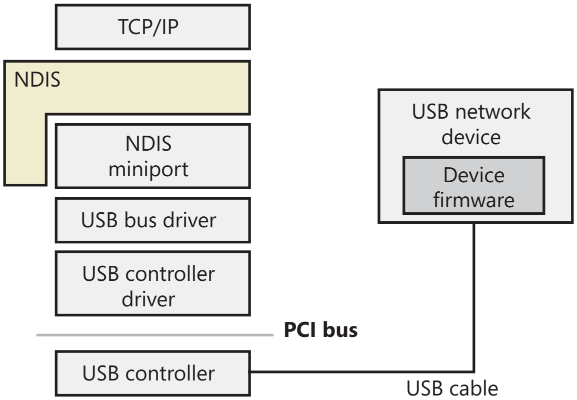


Figure 13-20 NDIS miniport driver for a USB network device

Remote NDIS is a specification for network devices on dynamic Plug and Play I/O buses such as USB, IEEE 1394, and Infiniband. The specification eliminates the need for a hardware vendor to write an NDIS miniport driver at all by defining bus-independent messages and the mechanism by which the messages are transmitted over various buses. Remote NDIS messages mirror the NDIS interface and include messages for initializing and resetting a device, transmitting and receiving packets, setting and querying device parameters, and indicating media link status.

The Remote NDIS architecture, in Figure 13-21, relies on a Microsoft-supplied NDIS miniport driver, \Windows\System32\Drivers\Rndismp.sys, that translates NDIS commands and forwards them to a bus transport driver for the bus on which a device is located. The architecture allows for a single NDIS miniport driver to be used for all Remote NDIS drivers and a single bus transport driver for each supported bus.

Currently, Remote NDIS for USB devices is included on Windows XP and Windows Server 2003 and available as a download from Microsoft for Windows 2000. While Remote NDIS on IEEE 1394 is fully specified, Windows does not yet support it.

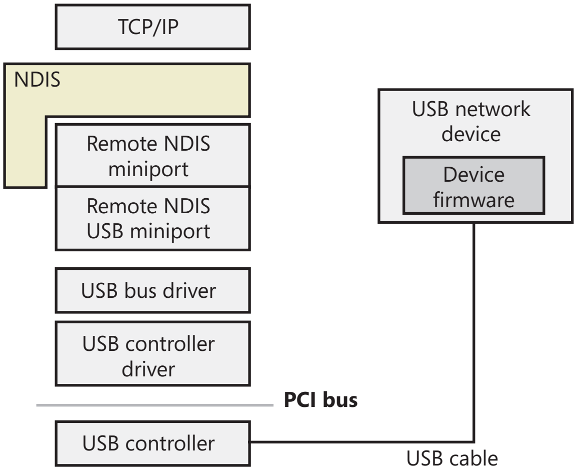


Figure 13-21 Remote NDIS architecture for USB network devices

QOS

If no special measures are taken, IP traffic is delivered over a network on a first-come, first-serve basis. Applications have no control over the priority of their messages, and they can experience *bursty* network behavior, where they occasionally obtain high throughput and low latencies, but otherwise receive poor network performance. While this level of service is acceptable in most situations, an increasing number of network applications demand more consistent service levels, or *quality of service* (QOS) guarantees. Video conferencing, media streaming, and enterprise resource planning (ERP) are examples of applications that require good network performance. QOS allows an application to specify minimum bandwidth and maximum latencies, which can be satisfied only if every networking software and hardware component between a sender and receiver supports QOS standards such as IEEE 802.1P, an industry standard that specifies the format of QOS packets and how OSI layer 2 devices (switches and network adapters) respond to them.

Windows QOS support is based on a handful of Microsoft-defined Winsock APIs that allow an application to request QOS for traffic over their Winsock sockets. For example, an application uses *WSCInstallQOSTemplate* to install a QOS template that specifies desired bandwidth and latency. A second API, the Traffic Control (TC) API, lets an administrative application more precisely control the traffic flow over a networks attached to the computer.

The heart of the Windows QOS implementation is the Resource Reservation Protocol (RSVP) Win32 service (\Windows\System32\Rsvp.exe), as shown in Figure 13-22 The RSVP Winsock service provider (\Windows\System32\Rsvpsp.dll) communicates application QOS requests via RPC to the RSVP service. The RSVP service in turn uses the TC API to control traffic flow. The TC API, which is implemented in \Windows\System32\Traffic.dll, sends I/O control commands to the Generic Packet Classifier (GPC) driver (\Windows\System32\Msgpc.sys). The GPC driver communicates closely with the QOS packet scheduler NDIS intermediate driver (\Windows\System32\Psched.sys) to control the flow of packets to the network from the computer so that the QOS levels promised to particular applications can be met, and so that appropriate QOS headers can be placed on packets for which QOS is desired.

Note RSVP signaling functionality is removed in Windows XP and Windows Server 2003. While the RSVP service still executes, it serves only as a conduit between applications and traffic control components.

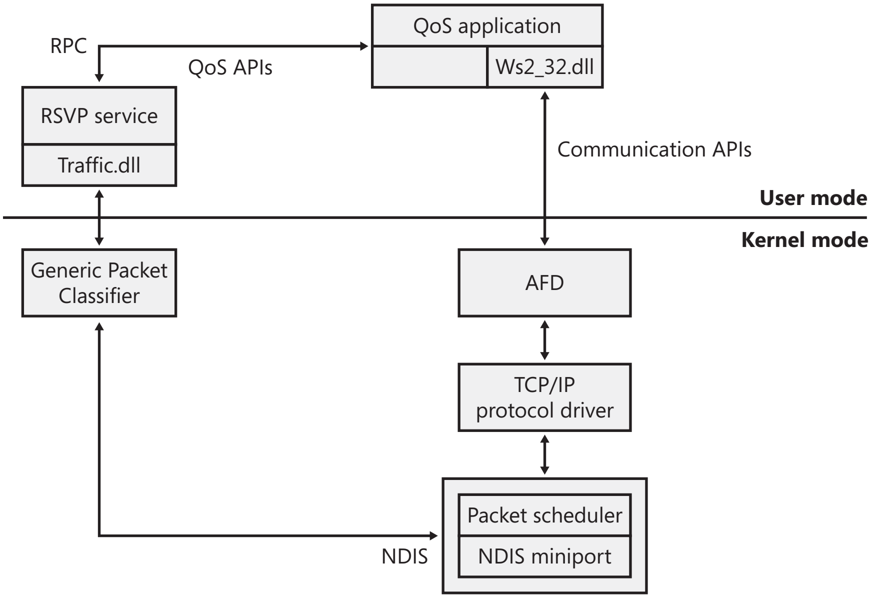


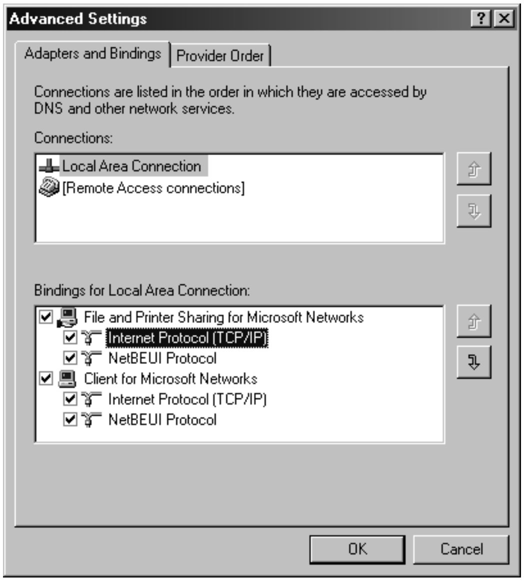
Figure 13-22 QOS architecture

Binding

The final piece in the Windows networking architecture puzzle is the way in which the components at the various layers—networking API layer, TDI transport driver layer, NDIS driver layer—locate one another. The name of the process that connects the layers is *binding*. You've witnessed binding taking place if you've changed your network configuration by adding or removing a component using the Network Connections folder.

When you install a networking component, you must supply an INF file for the component. (INF files are described in Chapter 9.) This file includes directions that setup API routines must follow to install and configure the component, including binding dependencies or binding relationships. A developer can specify binding dependencies for a proprietary component so that the Service Control Manager (the Service Control Manager described in Chapter 4) will not only load the component in the correct order but will also load the component only if other components the proprietary component depends on are present on the system. Binding relationships, which the bind engine determines with the aid of additional information in a component's INF file, establish connections between components at the various layers. The connections specify which components a network component on one layer can use on the layer beneath it.

For example, the Workstation service (redirector) will automatically bind to the TCP/IP and NWLink protocols. The order of the binding, which you can examine on the Adapters And Bindings tab in the Advanced Settings dialog box, shown in Figure 13-23, determines the priority of the binding. (See the section "Multiple Redirector Support" in this chapter for instructions on how to launch the Advanced Settings dialog box.) When the redirector receives a request to access a remote file, it submits the request to both protocol drivers simultaneously. When the response comes, the redirector waits until it has also received responses from any higher-priority protocol drivers. Only then will the redirector return the result to the caller. Thus, it can be advantageous to reorder bindings so that bindings of high priority are also the most performance efficient or applicable to most of the computers in your network. You can also manually remove bindings with the Advanced Settings dialog box.



Connections are listed in the order in which they are accessed by DNS and other network services.

Connections:

- Local Area Connection
- [Remote Access connections]

Bindings for Local Area Connection:

- ☑ File and Printer Sharing for Microsoft Networks
  - ☑ Internet Protocol (TCP/IP)
  - ☑ NetBEUI Protocol
- ☑ Client for Microsoft Networks
  - ☑ Internet Protocol (TCP/IP)
  - ☑ NetBEUI Protocol

Figure 13-23 Editing bindings with the Advanced Settings dialog box

The Bind value, in the Linkage subkey of a network component's registry configuration key, stores binding information for that component. For example, if you examine HKLM\SYSTEM\CurrentControlSet\Services\LanmanWorkstation\Linkage\Bind, you'll see the binding information for the Workstation service.

Layered Network Services

Windows includes network services that build on the APIs and components we've presented in this chapter. Describing the capabilities and detailed internal implementation of these services is outside the scope of this book, but this section provides a brief overview of remote access, Active Directory, Network Load Balancing, File Replication service (FRS), and Distributed File System (DFS).

Remote Access

Remote access, which is available with Windows Server with the Routing and Remote Access service, allows remote access clients to connect to remote access servers and access network resources such as files, printers, and network services as if the client were physically connected to the remote access server's network. Windows provides two types of remote access:

- Dial-up remote access is used by clients that connect to a remote access server via a telephone or other telecommunications infrastructure. The telecommunications medium is used to create a temporary physical or virtual connection between the client and the server.

- Virtual private network (VPN) remote access lets a VPN client establish a virtual point-to-point connection to the server over an IP network such as the Internet.

Remote access differs from remote control solutions because remote access acts as a proxy connection to a Windows network, whereas remote control software executes applications on a server, presenting a user interface to the client.

Active Directory

Active Directory is the Windows implementation of Lightweight Directory Access Protocol (LDAP) directory services. Active Directory is based on a database that stores objects representing resources defined by applications in a Windows network. For example, the structure and membership of a Windows domain, including the user account and password information, are stored in Active Directory.

Object classes and the attributes that define properties of objects are specified by a *schema*. The objects in the Active Directory are hierarchically arranged, much like the registry's logical organization, where container objects can store other objects, including other container objects. (See Chapter 8 for more information on container objects.)

Active Directory supports a number of APIs that clients can use to access objects within an Active Directory database:

- The LDAP C API is a C language API that uses the LDAP networking protocol. Applications written in C or C++ can use this API directly, and applications written in other languages can access the APIs through translation layers.
- Active Directory Service Interfaces (ADSI) is a COM interface to Active Directory implemented on top of LDAP that abstracts the details of LDAP programming. ADSI supports multiple languages, including Microsoft Visual Basic, C, and Microsoft Visual C++. ADSI can also be used by Microsoft Windows Script Host (WSH) applications.
- Messaging API (MAPI) is supported for compatibility with Microsoft Exchange client and Outlook Address Book client applications.
- Security Account Manager (SAM) APIs are built on top of Active Directory to provide an interface to logon authentication packages such as MSV1_0 (\Windows\System32\Msv1_0.dll, which is used for legacy NT LAN Manager authentication) and Kerberos (\Windows\System32\Kdcsvc.dll).
- Windows NT 4 networking APIs (Net APIs) are used by Windows NT 4 clients to gain access to Active Directory through SAM.
- NTDS API is used to lookup SIDs and GUIDs in an active directory (via DsCrackNames() mostly) as well as its main purpose of active directory management, and replication. Several third parties have written applications that monitor AD from these APIs.

Active Directory is implemented as a database file that by default is named \Windows\Ntds\Ntds.dit, and that is replicated across the domain controllers in a domain. The Active Directory directory service, which is a Windows service that executes in the Local Security Authority Subsystem (LSASS) process, manages the database, using DLLs that implement the on-disk structure of the database as well as provide transaction-based updates to protect the integrity of the database. On Windows 2000, the Active Directory database store is based on a version of the Extensible Storage Engine (ESE) database that is used by Microsoft Exchange Server version 5.5 client/server messaging and groupware and on Windows 2003 Server the store is based on a version of the ESE database used by Microsoft Exchange Server 2000. Figure 13-24 shows the Active Directory architecture.

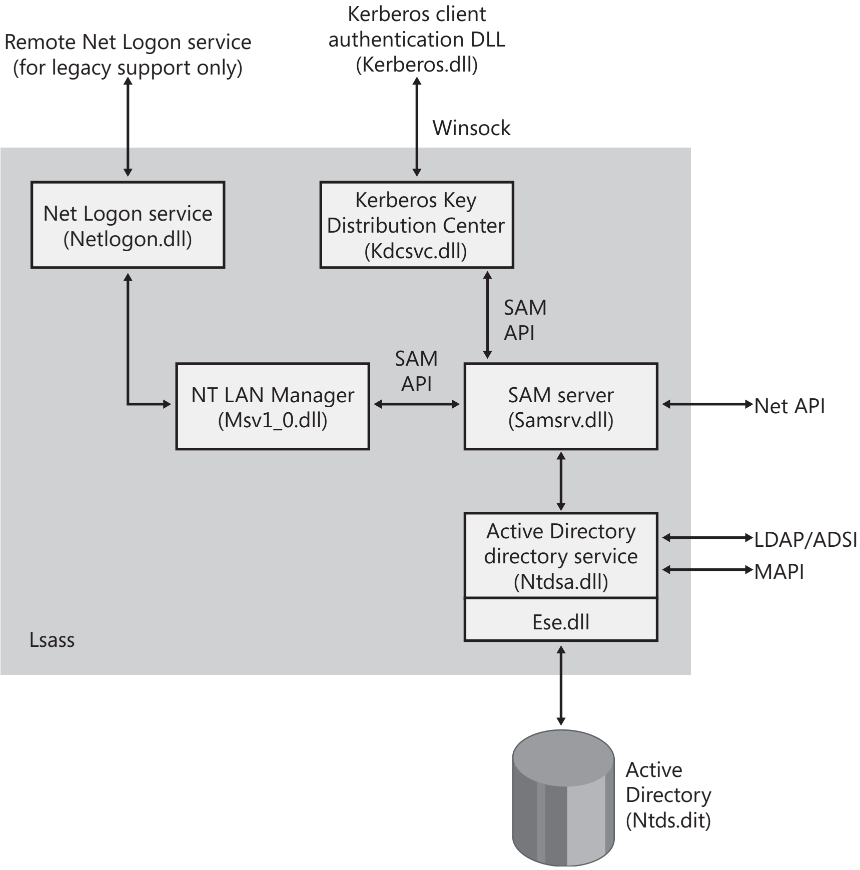


Figure 13-24 Active Directory architecture

Network Load Balancing

As we stated earlier in the chapter, Network Load Balancing, which is included with Windows Advanced Server, is based on NDIS intermediate-driver technology. Network Load Balancing allows for the creation of a cluster containing up to 32 computers, which are called *cluster hosts* in Network Load Balancing. The cluster maintains a single virtual IP address that is published for access by clients, and client requests go to all the computers in the cluster. However, only one cluster host responds to the request. The Network Load Balancing NDIS drivers effectively partition the client space among available cluster hosts in a distributed manner. This way, each host handles its portion of incoming client requests and every client request always gets handled by one and only one host. The cluster host that determines it should handle a client request allows the request to propagate up to the TCP/IP protocol driver and eventually a server application; the other cluster hosts don't. If a cluster host fails, the rest of the cluster realizes that the cluster host is no longer a candidate for processing requests and redistributes the incoming client requests to the remaining cluster hosts. No new client requests are sent to the downed cluster host. Another cluster host can be added to the cluster as a replacement, and it will then seamlessly start handling client requests.

Network Load Balancing isn't a general-purpose clustering solution because the server application that clients communicate with must have certain characteristics: the first is that it must be TCP/IP-based, and the second is that it must be able to handle client requests on any system in a Network Load Balancing cluster. This second requirement typically means that an application that must have access to shared state in order to service client requests must manage the shared state itself—Network Load Balancing doesn't include services for automatically distributing shared state across cluster hosts. Applications that are ideally suited for Network Load Balancing include a Web server that serves static content, Windows Media Server, and Terminal Services. Figure 13-25 shows an example of a Network Load Balancing operation.

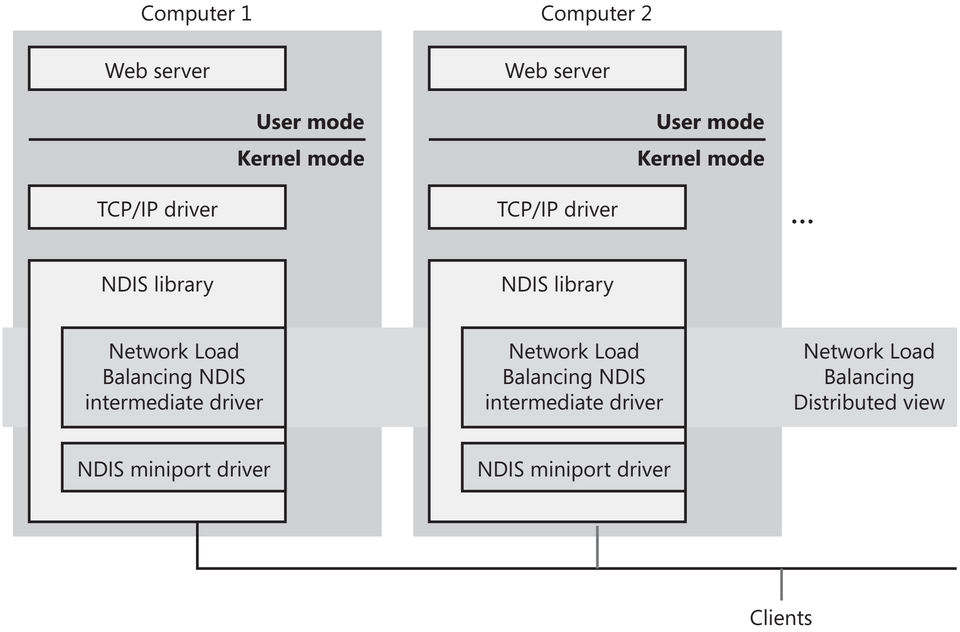


Figure 13-25 Network Load Balancing operation

File Replication Service

File Replication service (FRS) is included with Windows Server systems. Its primary purpose is to replicate the contents of a domain controller's \SYSVOL directory, which is where Windows domain controllers store logon scripts and group policies. (Group policies permit administrators to define usage and security policies for the computers that belong to a domain.) In addition, FRS can be used to replicate Distributed File System (DFS) shares between systems. FRS allows for distributed multimaster replication, which enables any server to perform replication activity. When a replicated directory or file is changed, the changes are propagated to the other domain controllers.

The fundamental concept in FRS is a *replica set*, which consists of two or more systems that replicate between themselves the contents of a directory tree according to an administratively defined schedule and topology. Only directories on NTFS volumes can be replicated because FRS relies on the NTFS change journal to detect changes to files in directories in a replica set. Because FRS is based on multimaster replication, it can theoretically support hundreds or even thousands of systems as part of a replica set, and the computers of a replica set can be connected with arbitrary network topologies (such as ring, star, or mesh). Computers can also be members of multiple replica sets.

FRS is implemented as a Windows service (\Windows\System32\Ntfrs.exe) that uses authenticated RPC with encryption to communicate between instances of itself running on different computers. In addition, because Active Directory contains its own replication capabilities, FRS uses Active Directory APIs to retrieve FRS configuration information from a domain's Active Directory.

Distributed File System

Distributed File System (DFS) is a service that layers on top of the Workstation service to connect together file shares into a single namespace. The file shares can reside on the same computer or on different computers, and DFS provides client access to the resources in a location-transparent manner. The root of a DFS namespace must be a file share defined on a Windows server.

In addition to delivering a unified network-resource namespace, DFS provides other benefits through DFS replica sets. An administrator can a create DFS replica set from two or more shares and use a replication mechanism such as FRS to copy data between the shares of a replica set to keep their contents synchronized. DFS provides several forms of load balancing by ordering and/or selecting members of a replica set to fulfill a client request for data on the replica set. In addition, DFS achieves high availability by routing requests to the working member or members of a replica set when a member becomes unavailable.

The components that make up the DFS architecture are shown in Figure 13-26. The server-side implementation of DFS consists of a Windows service (\Windows\System32\Dfssvc.exe) and a device driver (\Windows\System32\Drivers\Dfs.sys). The DFS service is responsible for exporting DFS topology management interfaces and maintaining the DFS topology in either the registry (on non–Active Directory systems) or Active Directory. The DFS driver performs topology lookups when it receives a client request so that it can direct the client to the system where the file it is requesting resides.

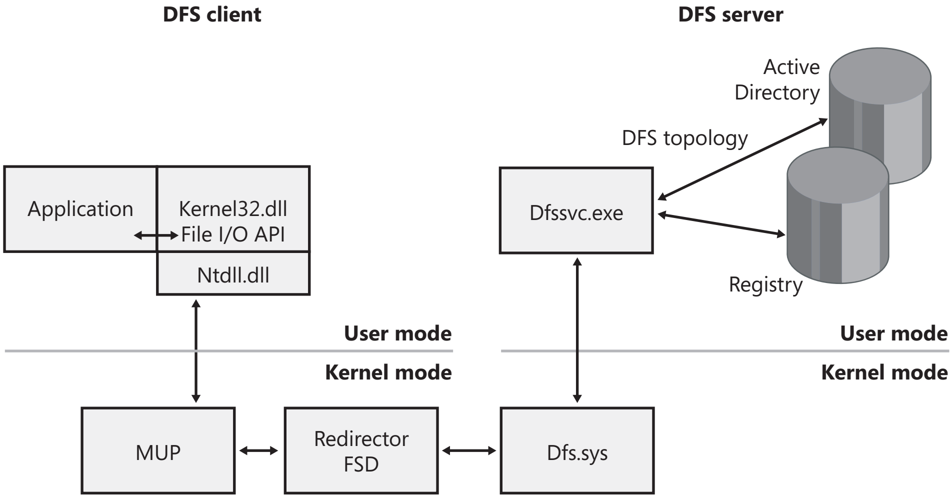


Figure 13-26 DFS components

On the client side, DFS support is implemented in the MUP driver, described earlier, and uses the CIFS redirector for its internal communication with DFS servers. The DFS client provider is implemented in \Windows\System32\Ntlanman.dll. When a client issues a file I/O request that specifies a file in the DFS namespace, the MUP driver on the client communicates with target file server by using the appropriate redirector.

Conclusion

The Windows network architecture provides a flexible infrastructure for networking APIs, network protocol drivers, and network adapter drivers. The Windows networking architecture takes advantage of I/O layering to give networking support the extensibility to evolve as computer networking evolves. When new protocols appear, developers can write a TDI transport to implement the protocol on Windows. Similarly, new APIs can interface to existing Windows protocol drivers. Finally, the range of networking APIs implemented on Windows affords network application developers a range of possible implementations, each with different programming models and protocol support.

Chapter 14

Crash Dump Analysis

Almost every Windows user has heard of, if not experienced, the infamous “blue screen of death.” This ominous term refers to the blue screen that is displayed when Microsoft Windows crashes, or stops executing, because of a catastrophic fault or an internal condition that prevents the system from continuing to run.

In this chapter, we’ll cover the basic problems that cause Windows to crash, describe the information presented on the blue screen, and explain the various configuration options available to create a *crash dump*, a record of system memory at the time of a crash that can help you figure out which component caused the crash. This section is *not* intended to provide detailed troubleshooting information on how to analyze a Windows system crash. This section will also show you how to analyze a crash dump to identify a faulty driver or component. The effort required to perform basic crash dump analysis is minimal and takes a few minutes. Even if an analysis ascertains the problematic driver for only one out of every five or ten crash dumps, it’s still worth doing: one successful analysis can avoid future data loss, system downtime, and frustration.

Why Does Windows Crash?

Windows crashes (stops execution and displays the blue screen) for the following reasons:

- A device driver or an operating system function running in kernel mode incurs an unhandled exception, such as a memory access violation (which is caused either by attempting to write to a read-only page or attempting to read an address that isn’t currently mapped and therefore is not a valid memory location).
- A call to a kernel support routine results in a reschedule, such as waiting for an unsignaled dispatcher object when the interrupt request level (IRQL) is DPC/dispatch level or higher. (See Chapter 3 for details on IRQLs.)
- A page fault on memory backed by data in a paging file or a memory mapped file occurs at an IRQL of DPC/dispatch level or above (which would require the memory manager to have to wait for an I/O operation to occur—and, as just stated, waits can’t occur at DPC/dispatch level or higher because that would require a reschedule).

- A device driver or operating system function explicitly crashes the system (by calling the system function *KeBugCheckEx*) because it detects an internal condition that indicates either a corruption or some other situation that indicates the system can't continue execution without risking data corruption.
- A hardware error, such as a machine check or a nonmaskable interrupt (NMI), occurs.

Microsoft's analysis of the crash dumps that Windows XP users have submitted to the Microsoft Online Crash Analysis (OCA, which is described later in this chapter) Web site reveals the breakdown of crash causes as shown in Figure 14-1. (The data shown was generated in April 2004.)

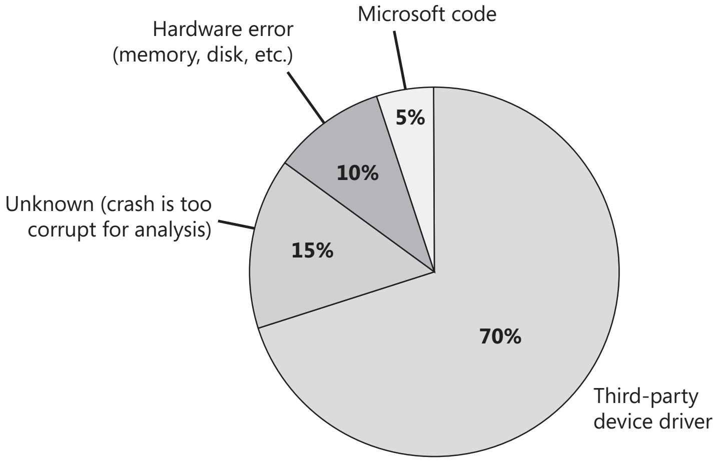


Figure 14-1 Causes of OCA-submitted crashes

When a kernel-mode device driver or subsystem causes an illegal exception, Windows faces a difficult dilemma. It has detected that a part of the operating system with the ability to access any hardware device and any valid memory has done something it wasn't supposed to do.

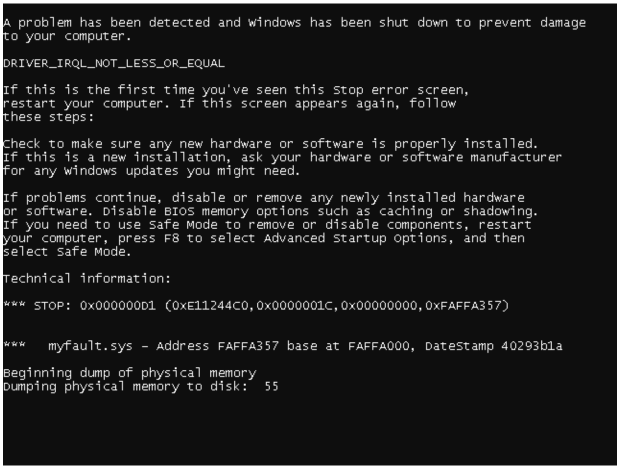
But why does that mean Windows has to crash? Couldn't it just ignore the exception and let the device driver or subsystem continue as if nothing had happened? The possibility exists that the error was isolated and that the component will somehow recover. But what's more likely is that the detected exception resulted from deeper problems—for example, from a general corruption of memory or from a hardware device that's not functioning properly. Permitting the system to continue operating would probably result in more exceptions, and data stored on disk or other peripherals could become corrupt—a risk that's too high to take.

The Blue Screen

Regardless of the reason for a system crash, the function that actually performs the crash is *KeBugCheckEx* (documented in the Windows DDK). This function takes a *stop code* (sometimes called a *bug check code*) and four parameters that are interpreted on a per–stop code basis. After

KeBugCheckEx masks out all interrupts on all processors of the system, it switches the display into a low-resolution VGA graphics mode (one implemented by all Windows-supported video cards), paints a blue background, and then displays the stop code, followed by some text suggesting what the user can do. Finally, *KeBugCheckEx* calls any registered device driver bug check callbacks (registered by calling the *KeRegisterBugCheckCallback* function), allowing drivers an opportunity to stop their devices. (It's possible that system data structures have been so seriously corrupted that the blue screen isn't displayed.) Figure 4-1 shows a sample Windows XP blue screen.

> **Note** Windows XP Service Pack 1 and higher, as well as Windows Server 2003, introduce the *KeRegisterBugCheckReasonCallback* function to enable device drivers to either append data to a crash dump or write crash dump information to alternate devices.



```
A problem has been detected and Windows has been shut down to prevent damage
to your computer.

DRIVER_IRQL_NOT_LESS_OR_EQUAL

If this is the first time you've seen this Stop error screen,
restart your computer. If this screen appears again, follow
these steps:

Check to make sure any new hardware or software is properly installed.
If this is a new installation, ask your hardware or software manufacturer
for any Windows updates you might need.

If problems continue, disable or remove any newly installed hardware
or software. Disable BIOS memory options such as caching or shadowing.
If you need to use Safe Mode to remove or disable components, restart
your computer, press F8 to select Advanced Startup Options, and then
select Safe Mode.

Technical information:

*** STOP: 0x000000D1 (0xE11244C0,0x0000001C,0x00000000,0xFAFFA357)


***   myfault.sys - Address FAFFA357 base at FAFFA000, DateStamp 40293b1a

Beginning dump of physical memory
Dumping physical memory to disk:  55
```

Figure 14-2 Example of a blue screen

KeBugCheckEx displays the textual representation of the stop code near the top of the blue screen and the numeric stop code and four parameters at the top of the Windows 2000 blue screen, but it displays these at the bottom of the Windows XP and Windows Server 2003 blue screens.

The first line lists the stop code and the four additional parameters passed to *KeBugCheckEx*. A text line near the top of the screen provides the text equivalent of the stop code's numeric identifier. According to the example in Figure 14-2, the stop code 0x000000D1 is a DRIVER_IRQL_NOT_LESS_OR_EQUAL crash. When a parameter contains an address of a piece of operating system or device driver code (as in Figure 14-2), Windows displays the base address of the module the address falls in, the date stamp, and the file name of the device driver. This information alone might help you pinpoint the faulty component.

Although there are more than a hundred unique stop codes, most are rarely, if ever, seen on production systems. Instead, just a few common stop codes represent the majority of Windows system crashes. Also, the meaning of the four additional parameters depends on the stop code (and not all stop codes have extended parameter information). Nevertheless, looking up the stop code and the meaning of the parameters (if applicable) might at least assist you in diagnosing the component that is failing (or the hardware device that is causing the crash).

You can find stop-code information in the section "Bug Checks (Blue Screens)" in the Windows Debugging Tools help file. (For information on the Windows Debugging Tools, see Chapter 1.) You can also search Microsoft's online Knowledge Base (*http://support.microsoft.com*) for the stop code and the name of the suspect hardware or application. You might find information about a workaround, an update, or a service pack that fixes the problem you're having. The Bugcodes.h file in the Windows DDK contains a complete list of the 150 or so stop codes, with some additional details on the reasons for some of them.

You often begin seeing blue screens after you install a new software product or piece of hardware. If you've just added a driver, rebooted, and gotten a blue screen early in system initialization, you can reset the machine, press the F8 key when instructed, and then select Last Known Good Configuration. Enabling last known good causes Windows to revert to a copy of the registry's device driver registration key (HKLM\SYSTEM\CurrentControlSet\Services) from the last successful boot (before you installed the driver). From the perspective of last known good, a successful boot is one in which all services and drivers have finished loading and at least one logon has succeeded. (Last known good is further described in Chapter 5.)

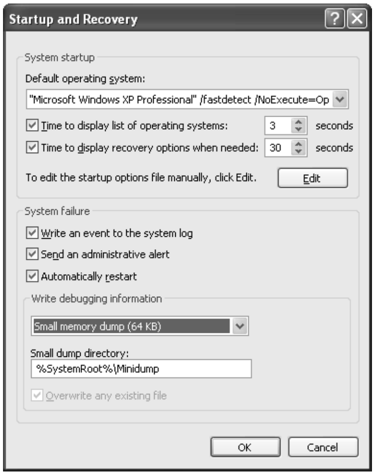
If you keep getting blue screens, an obvious approach is to uninstall the components you added just before the first blue screen appeared. If some time has passed since you added something new or you added several things at about the same time, you need to note the names of the device drivers referenced in any of the parameters. If you recognize any of the names as being related to something you just added (such as Scsiport.sys if you put on a new SCSI drive), you've possibly found your culprit.

Many device drivers have cryptic names, but one approach you can take to figure out which application or hardware device is associated with a name is to find out the name of the service in the registry associated with a device driver by searching for the name of the device driver under the HKLM\SYSTEM\CurrentControlSet\Services key. This branch of the registry is where Windows stores registration information for every device driver in the system. If you find a match, look for values named DisplayName and Description. Some drivers fill in these values to describe the device driver's purpose. For example, you might find the string "Virus Scanner" in the DisplayName value, which can implicate the antivirus software you have running. The list of drivers can be displayed in the Computer Management tool (from the Start menu, select Programs/Administrative Tools/Computer Management). In Computer Management, expand System Tools, System Information, and Software Environment, and then select Drivers.

More often than not, however, the stop code and the four associated parameters aren't enough information to troubleshoot a system crash. For example, you might need to examine the kernel-mode call stack to pinpoint the driver or system component that triggered the crash. Also, because the default behavior on Windows systems is to automatically reboot after a system crash, it's unlikely that you would have time to record the information displayed on the blue screen. That is why, by default, Windows attempts to record information about the system crash to the disk for later analysis, which takes us to our next topic, crash dump files.

Crash Dump Files

By default, all Windows systems are configured to attempt to record information about the state of the system when the system crashes. You can see these settings by opening the System tool in Control Panel, and then in the System Properties dialog box, click the Advanced tab and then click the Startup And Recovery button. The default settings for a Windows XP Professional system are shown in Figure 14-3.



System startup

Default operating system: "Microsoft Windows XP Professional" /fastdetect /NoExecute=Op

☑ Time to display list of operating systems: 3 seconds

☑ Time to display recovery options when needed: 30 seconds

To edit the startup options file manually, click Edit.

System failure

☑ Write an event to the system log

☑ Send an administrative alert

☑ Automatically restart

Write debugging information

Small memory dump (64 KB)

Small dump directory: %SystemRoot%\Minidump

☑ Overwrite any existing file

Figure 14-3 Crash dump settings

Three levels of information can be recorded on a system crash:

- **Complete memory dump** A complete memory dump contains all of physical memory at the time of the crash. This type of dump requires that a page file be at least the size of physical memory plus 1 MB (for a header). Because it can require an inordinately large page file on large memory systems, this type of dump file is the least common setting. Windows NT 4 supported only this type of crash dump file. This is the default setting for Windows Server systems.
- **Kernel memory dump** A kernel memory dump contains only the kernel-mode read/write pages present in physical memory at the time of the crash. This type of dump doesn't contain pages belonging to user processes. Because only kernel-mode code can directly cause Windows to crash, however, it's unlikely that user process pages are necessary to debug a crash. In addition, all data structures relevant for crash dump analy-

sis—including the list of running processes, stack of the current thread, and list of loaded drivers—are stored in nonpaged memory that saves in a kernel memory dump. There is no way to predict the size of a kernel memory dump because its size depends on the amount of kernel-mode memory allocated by the operating system and drivers present on the machine.

- **Small memory dump** A small memory dump (the default on Windows Professional), which is 64 KB in size (128 KB on 64-bit systems) and is also called a *minidump* or *triage dump*, contains the stop code and parameters, the list of loaded device drivers, the data structures that describe the current process and thread (called the EPROCESS and ETHREAD—described in Chapter 6), and the kernel stack for the thread that caused the crash.

While a complete memory dump is a superset of the other options, it has the drawback that its size tracks the amount of physical memory on a system and can therefore become unwieldy. It's not unusual for large server systems to have several gigabytes of memory, resulting in crash dump files that are too large to be uploaded to an FTP server or burned onto a CD. Because user mode code and data are not used during the analysis of most crashes (because crashes originate as a result of problems in kernel memory, and system data structures reside in kernel memory) much of the data stored in a complete memory dump is not relevant to analysis and therefore contributes wastefully to the size of a dump file. A final disadvantage is that the paging file on the boot volume (the volume with the \Windows directory) must be at least as large as the amount of physical memory on the system plus 1 MB. Because the size of paging files required, in general, inversely tracks the amount of physical memory present, this requirement can force the paging file to be unnecessarily large. You should therefore consider the advantages offered by the small and kernel memory dump options.

An advantage of a minidump is its small size, which makes it convenient for exchange via e-mail, for example. In addition, each crash generates a file in the directory \Windows\Minidump with a unique file name consisting of the string "Mini" plus the date plus a sequence number (for example, Mini082604-01.dmp). A disadvantage of minidumps is that to analyze them, you must have access to the exact images used on the system that generated the dump at the time you analyze the dump. (At a minimum, a copy of the matching Ntoskrnl.exe is needed to perform the most basic analysis.) This can be problematic if you want to analyze a dump on a system different from the system that generated the dump. However, the Microsoft symbol server contains images (and symbols) for Windows XP systems and later, so you can set the image path in the debugger to point to the symbol server and the debugger will automatically download the needed images. (Of course, the Microsoft image server won't have images for third-party drivers you have installed.)

A more significant disadvantage is that the limited amount of data stored in the dump can hamper effective analysis. You can also get the advantages of minidumps even when you configure a system to generate kernel or complete crash dumps by opening the larger crash with Windbg and using the **.dump /m** command to extract a minidump. Note that on Windows XP and Windows Server 2003, a minidump is automatically created even if the system is set for full or kernel dumps.

> **Note** You can use the .dump command from within Livekd to generate a memory image of a live system that you can analyze offline without stopping the system. This approach is useful when a system is exhibiting a problem, but is still delivering services, and you want to troubleshoot the problem without interrupting service. The resultant crash image isn't necessarily fully consistent because the contents of different regions of memory reflect different points in time, but it might contain information useful for an analysis.

The kernel memory dump option offers a practical middle ground. Because it contains all of kernel-mode–owned physical memory it has the same level of analysis-related data as a complete memory dump, but it omits the usually irrelevant user-mode data and code, and therefore can be significantly smaller. As examples, on a system running Windows XP with 256 MB of RAM, a kernel memory dump was 34 MB in size; and on another Windows XP system with 1.5 GB of RAM, a kernel memory dump took up 72 MB.

When you configure kernel memory dumps, the system checks whether the paging file is large enough (as outlined in Table 14-1), but these are only estimated sizes because there is no way to predict the size of a kernel memory dump. The reason you can't predict the size of a kernel memory dump is because its size depends on the amount of kernel-mode memory in use by the operating system and drivers present on the machine at the time of the crash.

Table 14-1 Default Minimum Paging File Sizes for Kernel Dumps

System Memory Size	Minimum Page File Size for Kernel Dumps
< 128 MB	50 MB
< 4 GB	200 MB
< 8 GB	400 MB
>= 8 GB	800 MB

Therefore, it is possible that at the time of the crash, the paging file is too small to hold a kernel dump. If you want to see the size of a kernel dump on your system, force a manual crash either by configuring the option to allow you to initiate a manual system crash from the console or by using the Notmyfault tool described later in this chapter. (Both these approaches are described later in the chapter.) When you reboot, you can check to make sure a kernel dump was generated and check its size to gauge how large to make your boot volume paging file. To be conservative, on 32-bit systems you can choose a page file size of 2 GB plus 1 MB, because 2 GB is the maximum kernel mode address space available.

Finally, even if the system successfully records the crash dump in the paging file at the time of the crash, there must be enough free disk space to extract the dump file. If there is not enough disk space, the crash dump is lost because the space used in the paging file to hold the dump is released and will be overwritten as the system begins to use the paging file. If you do not have enough space on the boot volume for saving the memory.dmp file, you can choose a location on any other local hard disk through the dialog box shown in Figure 14-3.

Crash Dump Generation

When the system boots, it checks the crash dump options configured by reading the registry value HKLM\System\CurrentControlSet\Control\CrashControl. If a dump is configured, it makes a copy of the disk miniport driver used to write to the boot volume in memory and gives it the same name as the miniport with the word "dump_" prefixed. It also checksums the components involved with writing a crash dump–including the copied disk miniport driver, the I/O Manager functions that write the dump, and the map of where the boot volume's paging file is on disk–and saves the checksum. When *KeBugCheckEx* executes, it checksums the components again and compares the new checksum with that obtained at the boot. If there's not a match, it does not write a crash dump, because doing so would likely fail or corrupt the disk. Upon a successful checksum match, *KeBugCheckEx* writes the dump information directly to the sectors on disk occupied by the paging file, bypassing the file system driver (which might be corrupted or even have caused the crash).

When SMSS enables paging during the boot process, the system looks in the boot volume's paging file to see whether a crash dump is present and protects the part of the paging file occupied by a dump. This makes part or all of the boot volume paging file unusable for the early part of the boot, which can cause notifications to display that the system is running low on virtual memory, a condition that is only temporary. Later in the boot, Winlogon determines whether or not a crash dump is in the paging file by calling the undocumented *NtQuerySystemInformation* API, and if a crash dump is there, it launches the Savedump process (\Windows\System32\Savedump.exe) to extract the crash dump from the paging file and copy it to its final location. These steps are shown in Figure 14-4.

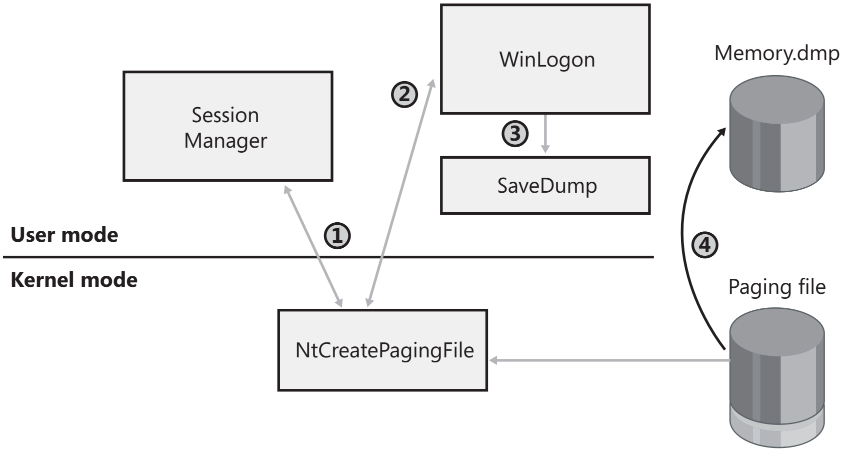
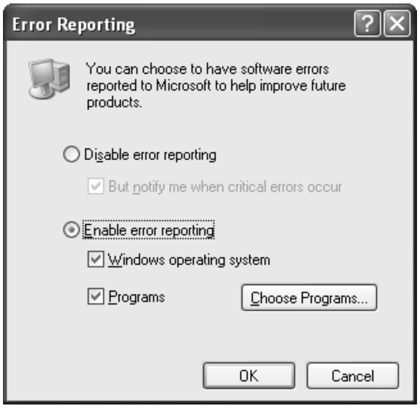


Figure 14-4 Crash dump file generation

Windows Error Reporting

As mentioned in Chapter 3, Windows XP and Windows Server 2003 include a facility called Windows Error Reporting, which facilitates the automatic submission of process and system crashes to Microsoft (or an internal error reporting server) for analysis. This feature is enabled by default, but it can be modified by changing Savedump's behavior, which takes the additional step of determining whether the system is configured to send a crash dump to Microsoft (or a private server, explained further in the "Online Crash Analysis" section later in the chapter) for analysis on a reboot following a crash. The Error Reporting dialog box, which you access from the Advanced tab of the Control Panel's System applet, is shown in Figure 14-5. This dialog box allows you to configure the system's error reporting settings, which are stored in the registry under HKLM\Software\Microsoft\PCHealth\ErrorReporting.



Error Reporting

You can choose to have software errors reported to Microsoft to help improve future products.

○ Disable error reporting

☑ But notify me when critical errors occur

● Enable error reporting

☑ Windows operating system

☑ Programs

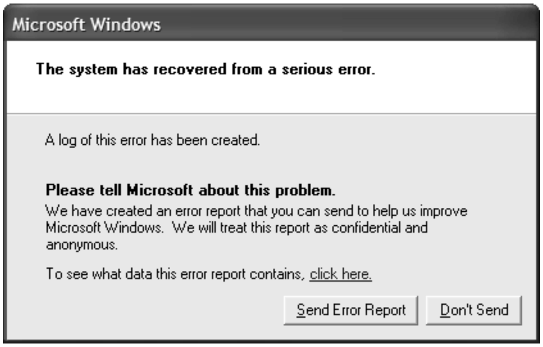
Figure 14-5 Error Reporting configuration dialog box

After rebooting from a crash, Savedump checks several values under the ErrorReporting key, including Showui, DoReport, and IncludeKernelFaults. If all three values are true, Savedump takes the following steps for preparing to send a crash dump report to the Microsoft Online Crash Analysis (OCA) site (or, if configured, an internal error reporting server):

1. If the type of dump it generated was not a minidump, it extracts a minidump from the dump file and stores it in the default location of \Windows\Minidumps.
2. It writes the name of the minidump file to HKLM\Software\Microsoft\PCHealth\ErrorReporting\KernelFaults.
3. It adds a command to execute Dumprep (\Windows\System32\Dumprep.exe) to HKLM\Software\Microsoft\Windows\CurrentVersion\Run so that Dumprep is executed during the first user's logon to the system.

Online Crash Analysis

When the Dumprep utility executes, as a result of Savedump having configured it to start, it checks the same three values referenced by Savedump to see whether the system is configured to send an error report to Microsoft after rebooting from a crash. If it is, Dumprep generates an XML-formatted file containing a basic description of the system, including the operating system version, a list of drivers installed on the machine, and the list of Plug and Play drivers loaded on the system at the time of the crash. It then presents the dialog box shown in Figure 14-6 that asks the user whether he or she wants to send an error report to Microsoft. If the user chooses to send the error report, and unless overridden by group policies, Dumprep sends the XML file and minidump to *http://Watson.Microsoft.Com*, which forwards the data to a server farm for automated analysis, described in the next section. Administrators can configure their systems using group policy to send the error data to an internal error-reporting network share to be later processed using the Microsoft Corporate Error Reporting (CER) Toolkit, which is available only to eligible Microsoft Software Assurance customers. (For more information, see *http://www.microsoft.com/resources/satech/cer*.)



The system has recovered from a serious error.

A log of this error has been created.

Please tell Microsoft about this problem.
We have created an error report that you can send to help us improve Microsoft Windows. We will treat this report as confidential and anonymous.

To see what data this error report contains, click here.

Figure 14-6 Crash dump error reporting dialog box

The server farm's automated analysis uses the same analysis engine that the Microsoft kernel debuggers use when you load a crash dump file into them (described shortly). The analysis generates a *bucket ID*, which is a signature that identifies a particular crash type. The server farm queries a database using the bucket ID to see whether a resolution has been found for the crash, and it sends a URL back to Dumprep that refers it to the OCA Web site (*http://oca.microsoft.com*). Dumprep launches the Internet browser to open the page on the OCA Web site that reports the preliminary crash analysis. If a resolution is available, the page instructs the user where to obtain a hotfix, service pack, or third-party driver update; otherwise, the user is given the option of following the progress of the crash analysis via e-mail.

Sites that are not connected to the Internet or do not want crash dumps to be sent automatically to Microsoft can configure, using group policy, the error data to be stored on an internal error-reporting network share to be later processed using the Microsoft CER Toolkit mentioned earlier.

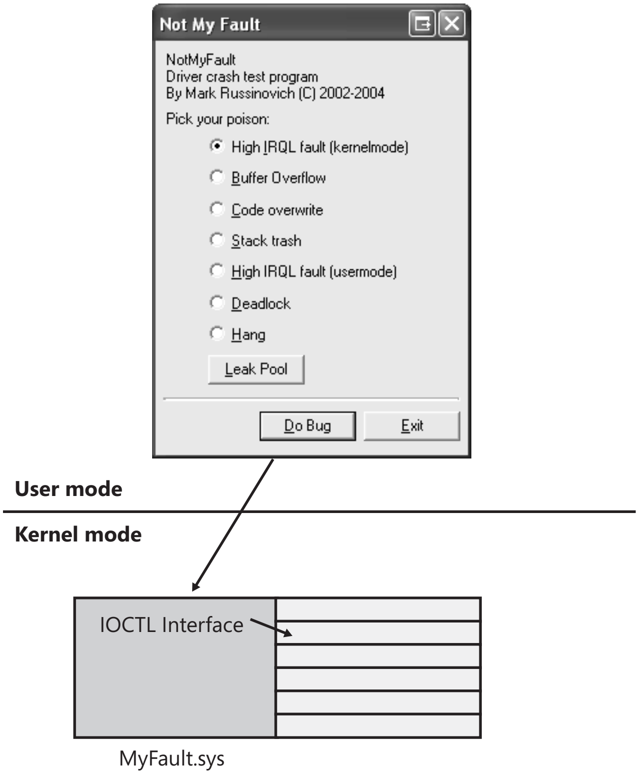
Basic Crash Dump Analysis

If OCA fails to identify a resolution or you are unable to submit the crash to OCA (if, for example, you have a crash dump file generated by Windows 2000, which doesn't support OCA), an alternative is analyzing crashes yourself. As mentioned earlier, Windbg and Kd both execute the same analysis engine used by OCA when you load a crash dump file and the basic analysis can sometimes pinpoint the problem. So you might be fortunate and have the crash dump solved by the automatic analysis. But if not, there are some straightforward techniques to try to solve the crash.

This section explains how to perform basic crash analysis steps, followed by tips on leveraging the Driver Verifier (which is introduced in Chapter 7) to catch buggy drivers when they corrupt the system so that a crash dump analysis pinpoints them.

Notmyfault

You can use the Notmyfault utility from *www.sysinternals.com/windowsinternals* to generate the crashes described here. Notmyfault consists of an executable named Notmyfault.exe and a driver named Myfault.sys. When you run the Notmyfault executable, it loads the driver and presents the dialog box shown in Figure 14-7, which allows you to crash the system in various ways or to cause the driver to leak paged pool. The crash types offered represent the ones most commonly seen by Microsoft Product Support Services. Selecting an option and clicking the Do Bug button causes the executable to tell the driver, by using the *DeviceIoControl* Windows API, which type of bug to trigger. Note that you should execute Notmyfault crashes on a test system or in a virtual machine because there is a small risk that memory it corrupts will be written to disk and result in file or disk corruption.



Not My Fault

NotMyFault
Driver crash test program
By Mark Russinovich (C) 2002-2004

Pick your poison:

- ● High IRQL fault (kernelmode)
- ○ Buffer Overflow
- ○ Code overwrite
- ○ Stack trash
- ○ High IRQL fault (usermode)
- ○ Deadlock
- ○ Hang

Figure 14-7 Notmyfault

Note The names of the Notmyfault executable and driver highlight the fact that user-mode cannot directly cause the system to crash. The Notmyfault executable can cause a crash only by loading a driver to perform an illegal operation for it in kernel-mode.

Basic Crash Dump Analysis

The most straightforward Notmyfault crash to debug is the one caused by selecting the High IRQL Fault (Kernelmode) option and clicking the Do Bug button. This causes the driver to allocate a page of paged pool, free the pool, raise the IRQL to above DPC/dispatch level, and then to touch the page it has freed. (See Chapter 3 for more information on IRQLs.) If that doesn't cause a crash, the process continues by reading memory past the end of the page until it causes a crash by accessing invalid pages. The driver performs several illegal operations as a result:

1. It references memory that doesn't belong to it.
2. It references paged pool at an IRQL that's DPC/dispatch level or higher, which is illegal because page faults are not permitted when the processor IRQL is DPC/dispatch level or higher.

3. When it goes past the end of the memory that it had allocated, it tries to reference memory that is potentially invalid.

The reason the first page reference might not cause a crash is that it won't generate a page fault if the page that the driver frees remains in the system working set. (See Chapter 7 for information on the system working set.)

When you load a crash generated with this bug into Kd, its analysis displays something like this:

```
Microsoft (R) Windows Debugger  Version 6.3.0011.2
Copyright (c) Microsoft Corporation. All rights reserved.


Loading Dump File [c:\windows\memory.dmp]
Kernel Summary Dump File: Only kernel address space is available

Symbol search path is: srv*c:\symbols*http://msdl.microsoft.com/download/symbols

Executable search path is:
Windows 2000 Kernel Version 2195 UP Free x86 compatible
Product: LanManNt, suite: Enterprise TerminalServer
Kernel base = 0x80400000 PsLoadedModuleList = 0x8046a4c0
Debug session time: Mon Apr 05 18:28:44 2004
System Uptime: 0 days 0:09:20.105
Loading Kernel Symbols
................................................................................
...............
Loading unloaded module list
No unloaded module list present
Loading User Symbols
..........
*******************************************************************************
*                                                                             *
*                        Bugcheck Analysis                                    *
*                                                                             *
*******************************************************************************

Use !analyze -v to get detailed debugging information.

BugCheck D1, {bead1800, 1c, 0, ec1fb357}

*** ERROR: Module load completed but symbols could not be loaded for myfault.sys

*** WARNING: Unable to verify checksum for NotMyfault.exe
*** ERROR: Module load completed but symbols could not be loaded for NotMyfault.exe
Probably caused by : myfault.sys ( myfault+357 )

Followup: MachineOwner
---------

kd>
```

The first thing to note is that Kd reports errors trying to load symbols for Myfault.sys and Notmyfault.exe. These are expected because the symbol files for them are not on the symbol-file path (which is configured to point at the Microsoft symbol server). You'll see similar errors for third-party drivers and executables that do not ship with the operating system.

The analysis text itself is terse, showing the numeric stop code and bug check parameters followed by a "probably caused by" line that shows the analysis engine's best guess at the offending driver. In this case it's on the mark and points directly at Myfault.sys, so there's no need for manual analysis.

The "Followup" line is not generally useful except within Microsoft, where the debugger looks for the module name in the Triage.ini file that's located within the Triage directory of the Debugging Tools for Windows installation directory. The Microsoft-internal version of that file lists the developer or group responsible for handling crashes in a specific driver, and the debugger displays their name in the Followup line when appropriate.

Verbose Analysis

Even though the basic analysis of the Notmyfault crash identifies the faulty driver, you should always have the debugger execute a verbose analysis by entering the command:

!analyze –v

The first obvious difference between the verbose and default analysis is the description of the stop code and its parameters. Following is the output of the command when executed on the same dump:

```
DRIVER_IRQL_NOT_LESS_OR_EQUAL (d1)
An attempt was made to access a pageable (or completely invalid) address at an
interrupt request level (IRQL) that is too high.  This is usually
caused by drivers using improper addresses.
If kernel debugger is available get stack backtrace.
Arguments:
Arg1: bead1800, memory referenced
Arg2: 0000001c, IRQL
Arg3: 00000000, value 0 = read operation, 1 = write operation
Arg4: ec1fb357, address which referenced memory
```

This saves you the trouble of opening the help file to find the same information, and the text sometimes suggests troubleshooting steps, an example of which you'll see in the next section on advanced crash dump analysis.

The other potentially useful information in a verbose analysis is the stack trace of the thread that was executing on the processor that crashed at the time of the crash. Here's what it looks like for the same dump:

```
STACK_TEXT:
WARNING: Stack unwind information not available. Following frames may be wrong.
```

```
b7e6dc34 804ac5de ff7c4040 beb4bf68 ff80a9c8 myfault+0x357
b7e6dd00 804a8f1e 00000070 00000000 00000000 nt!IopXxxControlFile+0x5e4
b7e6dd34 80461691 00000070 00000000 00000000 nt!NtDeviceIoControlFile+0x28
b7e6dd34 77f96be2 00000070 00000000 00000000 nt!KiSystemService+0xc4
0012f4a0 77e84c9b 00000070 00000000 00000000 ntdll!ZwDeviceIoControlFile+0xb
0012f504 004017c3 00000070 83360018 00000000 KERNEL32!DeviceIoControl+0x100
000200ac 00000000 00000000 00000000 00000000 NotMyfault+0x17c3
```

The preceding stack shows that the Notmyfault executable image, shown at the bottom, invoked the *DeviceIoControl* function in Kernel32.dll, which in turn invoked ZwDeviceIoControlFile in Ntdll.dll, and so on, until finally, the system crashed with the execution of an instruction in the Myfault image. A stack trace like this can be useful because crashes sometimes occur as the result of one driver passing another one improperly formatted, corrupt, or that has illegal parameters. The driver that's passed the invalid data might cause a crash and get the blame in an analysis, when the stack reveals that another driver was involved. In this sample trace, no driver other than myfault is listed. (The module "nt" is Ntoskrnl.)

If the driver singled out by an analysis is unfamiliar to you, use the lm (list modules) command to look at the driver's version information. Add the k (kernel modules) and v (verbose) options along with the m (match) option followed by the name of the driver and a wildcard:

```
kd> lm kv m myfault*
start    end        module name
f224d000 f224dbe0   myfault      (deferred)
    Image path: \??\C:\WINNT\system32\drivers\myfault.sys
    Timestamp: Thu Apr 29 14:53:12 2004 (40915D28)  Checksum: 00010090
    ImageSize : 00000BE0
    File version:     2.0.0.0
    Product version:  2.0.0.0
    File flags:       0 (Mask 3F)
    File OS:          40004 NT Win32
    File type:        3.7 Driver
    File date:        00000000.00000000
    Translations:     0409.04b0
    CompanyName:      Sysinternals
    ProductName:      Sysinternals Myfault
    InternalName:     myfault.sys
    OriginalFilename: myfault.sys
    ProductVersion:   2.0
    FileVersion:      2.0
    FileDescription:  Crash Test Driver
    LegalCopyright:   Copyright (C) M. Russinovich 2002-2004
```

In addition to using the description to identify the purpose of a driver, you can also use the file and product version numbers to see whether the version installed is the most up-to-date version available. (You can do this by checking the vendor Web site, for instance.) If version information isn't present (because it might have been paged out of physical memory at the time of the crash), look at the driver image file's properties in Explorer on the system that crashed.

Using Crash Troubleshooting Tools

The crash generated in the preceding section with Notmyfault's High IRQL Fault (Kernel-mode) option poses no challenge for the debugger's automated analysis. Unfortunately, most crashes are not so easy and often are impossible to debug. There are several levels of increasing severity in terms of system performance degradation that might help make a system that's producing crashes that cannot be analyzed into ones that can be. If the crashes generated after you configure a level and reboot aren't revealing the cause, try the next level.

1. If there are one or more drivers you consider likely sources of the crashes—because they were introduced into the system relatively recently, they were recently updated, or the circumstances of the crash implicate them—enable them for verification using the Driver Verifier and check all the verification options except for low resources simulation.
2. Enable the same level of verification as in level 1 on all unsigned drivers in the system. Or, if you are running Windows 2000 where the Driver Verifier doesn't distinguish between signed and unsigned drivers, enable it on all non-Microsoft drivers.
3. Enable the same verification as in level 1 on all drivers in the system. To maintain reasonable performance, you may want to divide the drivers into groups, enabling the Driver Verifier on one group at a time between reboots

Obviously, before you spend time and energy making system configuration changes and analyzing crashes, you should ensure that your system's kernel and drivers are the most recent available by using the services of Windows Update and third-party driver support sites.

Note If your system becomes unbootable because the Driver Verifier detects a driver error and crashes the system, then start in safe mode (where verification is disabled), run the Driver Verifier, and delete verification settings.

The following sections demonstrate how the Driver Verifier can make impossible-to-debug crashes into ones that you can solve. You should also refer to the Debugging Tools help file, which has tutorials on advanced debugging techniques.

Buffer Overrun and Special Pool

By far the most common source of crashes on Windows is pool corruption. Pool corruption usually occurs when a driver suffers from a buffer overrun or buffer underrun bug that causes it to overwrite data past either the end or start of a buffer it has allocated from paged or nonpaged pool. The Executive's pool-tracking structures reside on either side of a pool buffer and separate buffers from each other. These bugs, therefore, cause corruption to the pool tracking structures, to buffers owned by other drivers, or to both. The crashes caused by pool corruption are virtually impossible to debug because the system crashes when corrupted data is referenced, not when the corruption occurs.

> **Note** To assist in catching these difficult corruptions, Windows XP Service Pack 2 and later performs pool-block tail checking at all times. Thus, buffer overruns are likely to cause an immediate BAD_POOL_HEADER crash.

You can generate a pool corruption crash by running Notmyfault and selecting the Buffer Overflow bug. This causes Myfault to allocate a buffer and then overwrite the 40 bytes following the buffer. There can be a significant delay between the time you click the Do Bug button and a crash occurs, and you might even have to generate pool usage by exercising applications before a crash occurs, which highlights the distance between a corruption and its effect on system stability. An analysis of the resultant crash almost always reports Ntoskrnl or another driver as being the likely cause, which demonstrates the usefulness of a verbose analysis with its description of the stop code:

```
DRIVER_CORRUPTED_EXPOOL (c5)
An attempt was made to access a pageable (or completely invalid) address at an
interrupt request level (IRQL) that is too high.  This is
caused by drivers that have corrupted the system pool.  Run the driver
verifier against any new (or suspect) drivers, and if that doesn't turn up
the culprit, then use gflags to enable special pool.
Arguments:
Arg1: 4f4f4f53, memory referenced
Arg2: 00000002, IRQL
Arg3: 00000001, value 0 = read operation, 1 = write operation
Arg4: 80467139, address which referenced memory
```

The advice in the description is to run the Driver Verifier against any new or suspect drivers or to use Gflags to enable special pool. Both accomplish the same thing: to have the system detect a potential corruption when it occurs and crash the system in a way that makes the automated analysis point at the driver causing the corruption.

If the Driver Verifier's special pool option is enabled, verified drivers use special pool, rather than paged or nonpaged pool, for any allocations they make for buffers slightly less than a page in size. A buffer allocated from special pool is sandwiched between two invalid pages and by default is aligned against the top of the page. The special pool routines also fill the unused portions of the page in which the buffer resides with a random pattern. Figure 14-8 depicts a special pool allocation.

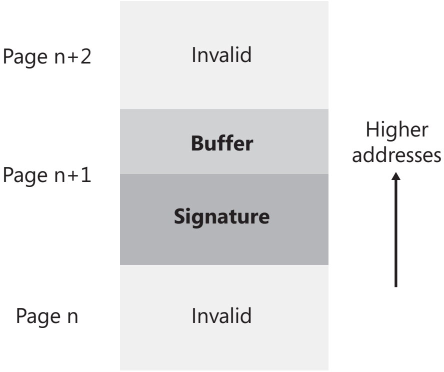


Figure 14-8 Special pool buffer allocation

The system detects any buffer overruns of under a page in size at the time of the overrun because they cause a page fault on the invalid page following the buffer. The signature serves to catch buffer underruns at the time the driver frees a buffer because the integrity of the pattern placed there at the time of allocation will have been compromised.

To see how the use of special pool causes a crash that the analysis engine easily diagnoses, run the Driver Verifier Manager. On Windows 2000 systems, go to the Settings tab, type **myfault.sys** in the edit box at the bottom of the page where it allows you to specify additional drivers, select the special pool check box, apply the changes, exit the Driver Verifier Manager, and reboot. On Windows XP and Windows Server 2003, choose the Create Custom Settings (For Code Developers) option on the first page of the wizard, choose Select Individual Settings From A Full List on the second, and select Special Pool. Choose the Select Drivers From A List option on the subsequent page, and on the page that lists drivers type **myfault.sys** into the File Find dialog box after pressing the button to add unloaded drivers. (You do not have to find myfault.sys in the File Find dialog box; just enter its name.) Then check the myfault.sys driver, exit the wizard, and reboot.

When you run Notmyfault and cause a buffer overflow, the system will immediately crash and the analysis of the dump reports this:

```
Probably caused by : myfault.sys ( myfault+3f1 )
```

A verbose analysis describes the stop code like this:

```
DRIVER_PAGE_FAULT_BEYOND_END_OF_ALLOCATION (d6)
N bytes of memory was allocated and more than N bytes are being referenced.
This cannot be protected by try-except.
When possible, the guilty driver’s name (Unicode string) is printed on
the bugcheck screen and saved in KiBugCheckDriver.
Arguments:
```

```
Arg1: beb50000, memory referenced
Arg2: 00000001, value 0 = read operation, 1 = write operation
Arg3: ec3473f1, if non-zero, the address which referenced memory.
Arg4: 00000000, (reserved)
```

Special pool made an elusive bug into one that instantly reveals itself and makes the analysis trivial.

Code Overwrite and System Code Write Protection

A driver with a bug that causes corruption or misinterpretation of its own data structures can reference memory the driver doesn't own when it interprets corrupted data as a memory pointer value. The target of the pointer can be anything in the virtual address space, including data belonging to other drivers, invalid memory, or the code of other drivers or the kernel. As with buffer overruns, by the time that corruption is detected and the system crashes it's usually impossible to identify the driver that caused the corruption. Enabling special pool increases the chance of catching wild-pointer bugs, but it does not catch code corruption.

When you run Notmyfault and select the Code Overwrite option, the Myfault driver corrupts the entry point to the *NtReadFile* kernel function. One of two things will happen at this point: if your system is running Windows XP and has 127 MB or less of physical memory or is running Windows Server 2003 and has 255 MB or less of physical memory, you'll get a crash for which an analysis points at Myfault.sys. The stop code description that a verbose analysis prints tells you that Myfault attempted to write to read-only memory:

```
ATTEMPTED_WRITE_TO_READONLY_MEMORY (be)
An attempt was made to write to readonly memory.  The guilty driver is on the
stack trace (and is typically the current instruction pointer).
When possible, the guilty driver's name (Unicode string) is printed on
the bugcheck screen and saved in KiBugCheckDriver.
Arguments:
Arg1: 804bb7fd, Virtual address for the attempted write.
Arg2: 004bb121, PTE contents.
Arg3: b804db60, (reserved)
Arg4: 0000000b, (reserved)
```

However, if you have Windows 2000 with more than 127 MB of memory or Windows XP Windows Server 2003 with more than 255 MB of memory, you'll get a different type of crash because the attempt to corrupt the memory isn't caught. Because *NtReadFile* is a commonly executed system service that is used by the Win32 subsystem to read keyboard and mouse input, the system will almost immediately crash as a thread attempts to execute the corrupted code and generates an illegal instruction fault. The analysis of crashes generated with this bug is always wrong, but it might vary, with Windows.sys and Ntoskrnl.exe commonly being the analyzer's best guess as to what's responsible. The bug check description for these crashes is:

```
KMODE_EXCEPTION_NOT_HANDLED (1e)
This is a very common bugcheck.  Usually the exception address pinpoints
the driver/function that caused the problem.  Always note this address
```

```
as well as the link date of the driver/image that contains this address.
Arguments:
Arg1: c0000005, The exception code that was not handled
Arg2: 80461885, The address that the exception occurred at
Arg3: 00000000, Parameter 0 of the exception
Arg4: 00000000, Parameter 1 of the exception
```

The reason for the different behaviors on different configurations relates to a mechanism introduced into Windows 2000 called *system code write protection*. Table 14-2 shows on which configurations system code write protection is enabled by default.

Table 14-2 System Code Write Protection Configurations

	Windows 2000	Windows XP and Windows Server 2003
System Code Write Protection Disabled	RAM > 127 MB	RAM > 255 MB

If system code write protection is enabled, the memory manager maps Ntoskrnl.exe, the HAL, and boot drivers using standard physical pages (4 KB on x86 and x64, and 8 KB on IA64). Because the granularity of protection in an image is the standard page size, the memory manager can write-protect code pages so that an attempt to modify them generates an access fault (as seen in the first crash). However, when system code write protection is disabled, the memory manager uses large pages (4 MB on x86, and 16 MB on IA64 and x86-64) to map Ntoskrnl.exe, which includes Windows 2000 with more than 127 MB of RAM and Windows XP or Windows Server 2003 with more than 255 MB of RAM. It cannot protect code because code and data might reside on the same page.

If system code-write protection is off and crash analysis reports unlikely causes for a crash or you suspect code corruption, you should enable it. Verifying at least one driver with the Driver Verifier is the easiest way to enable it. You can also enable it manually by adding two registry values under HKLM\System\CurrentControlSet\Control\Session Manager\Memory Management. First, specify the amount of RAM at which the memory manager uses large pages instead of standard pages to map Ntoskrnl.exe to an effectively infinite value. You do this by creating a DWORD value called LargePageMinimum and setting it to 0xFFFFFFFF. Then add another DWORD value named EnforceWriteProtection and set it to 1. You must reboot for the changes to take effect.

Note When the debugger has access to the image files included in a crash dump, the analysis internally executes the *!chkimg* debugger command to verify that a copy of an image in a crash dump matches the on-disk image and reports any differences. Note that *chkimg* will always report discrepancies in Ntoskrnl.exe if you've enabled the Driver Verifier.

Advanced Crash Dump Analysis

The preceding section leverages the Driver Verifier to create crashes that the debugger's automated analysis engine can resolve. You might still encounter cases where you cannot get a system to produce easily analyzable crashes and, if so, you will need to execute manual analysis to try and determine what the problem is.

- Use the *!process 0 0* debugger command to look at the processes running and make sure that you understand the purpose of each one. Try disabling or uninstalling unnecessary applications and services.
- Use the *lm* command with the *kv* option to list the loaded kernel-mode drivers. Make sure that you understand the purpose of any third-party drivers and that you have the most recent versions.
- Use the *!vm* command to see whether the system has exhausted virtual memory, paged pool, or nonpaged pool. If virtual memory is exhausted, the committed pages will be close to the commit limit, so try to identify a potential memory leak by examining the list of processes to see which one reports high commit usage. If nonpaged pool or paged pool is exhausted (that is, the usage is close to the maximum) see the "Troubleshooting Pool a Leak" experiment in Chapter 7.

There are other debugging commands that can prove useful, but more advanced knowledge is required to apply them. The *!irp* command is one of them. The next section shows the use of this command to identify a suspect driver.

Stack Trashes

Stack overrun or stack trashing results from buffer overrun or underrun bugs. However, instead of residing in pool, as you saw with Notmyfault's buffer overrun bug, the target buffer is on the stack of the thread that executes the bug. This type of bug is another one that's difficult to debug because the stack is the foundation for any crash dump analysis.

When you run Notmyfault and select Stack Trash, the Myfault driver overruns a buffer it allocates on the kernel stack of the thread that executes it. When Myfault tries to return control to the Ntoskrnl function that invoked, it reads the return address, which is the address at which it should continue executing, from the stack. The address was corrupted by the stack-buffer overrun, so the thread continues execution at some different address in memory—an address that might not even contain code. An illegal exception and crash occurs when the thread executes an illegal CPU instruction or it references invalid memory.

The driver that the crash dump analysis of a stack overrun points the blame at will vary from crash to crash, but the stop code will almost always be KMODE_EXCEPTION_NOT_HANDLED. If you execute a verbose analysis, the stack trace looks like this:

```
STACK_TEXT:
b7b0ebd4 00000000 00000000 00000000 00000000 0x0
```

This is consistent with the stack having been overwritten with zeros. Unfortunately, mechanisms like special pool and system code write protection can't catch this type of bug. Instead, you must take some manual analysis steps to determine indirectly which driver was operating at the time of the corruption. One way is to examine the IRPs that are in progress for the thread that was executing at the time of the stack trash. When a thread issues an I/O request, the I/O manager stores a pointer to the outstanding IRP on the Irp list of the ETHREAD structure for the thread. The *!thread* debugger command dumps the thread list of the target IRP. (If you don't specify a thread object address,*!thread* dumps the processor's current thread.) Then you can look at the IRP with the *!irp* command:

```
kd> !thread
THREAD ff740020  Cid 8f8.420  Teb: 7ffde000  Win32Thread: a20cdbe8 RUNNING
IRP List:
    bc5a7f68: (0006,0094) Flags: 00000000  Mdl: 00000000
Not impersonating
Owning Process ff75f120
...

kd> !irp bc5a7f68
Irp is active with 1 stacks 1 is current (= 0xbc5a7fd8)
 No Mdl Thread ff740020:  Irp stack trace.
     cmd  flg cl Device   File     Completion-Context
>[  e, 0]   0  0 ff79e4c0 ff7ac028 00000000-00000000
       \Driver\MYFAULT Args: 00000000 00000000 83360010 00000000
```

The output shows that the IRP's current and only stack location (designated with the ">" prefix) is owned by the Myfault driver. If this were a real crash, the next steps would be to ensure that the driver version installed is the most recent available, install the new version if it isn't, and if it is, to enable the Driver Verifier on the driver (with all settings except low memory simulation).

Hung or Unresponsive Systems

If a system becomes unresponsive (that is, you are receiving no response to keyboard or mouse input), the mouse freezes, or you can move the mouse but the system doesn't respond to clicks, the system is said to have *hung*. There can be a number of things that can cause the system to hang:

- A device driver does not return from its interrupt service (ISR) routine or deferred procedure call (DPC) routine
- A high priority realtime thread preempts the windowing system driver's input threads
- A deadlock (when two threads or processors hold resources each other wants and neither will yield what they have) occurs in kernel mode

If you have Windows XP or Windows Server 2003, you can check for deadlocks by using the Driver Verifier option called *deadlock detection*. Deadlock detection monitors the use of spin locks, fast mutexes, and mutexes, looking for patterns that could result in a deadlock. (For more information on these and other synchronization primitives, see Chapter 3.) If one is found, Driver Verifier crashes the system with an indication of which driver causes the deadlock. The simplest form of deadlock occurs when two threads hold resources each other thread wants and neither will yield what they have or give up waiting for the one they want. If you are running Windows XP or Windows Server 2003, the first step to troubleshooting hung systems is therefore to enable deadlock detection on suspect drivers, then unsigned drivers, and then all drivers, until you get a crash that pinpoints the driver causing the deadlock.

If you are running Windows 2000 or have verified all drivers and still get hangs, you must either manually crash the hung system and analyze the resulting dump or break in with the kernel debugger to take an alternate approach to investigating the hang.

There are two ways to approach a hanging system so that you can apply the manual crash troubleshooting techniques described in this chapter to determine what driver or component is causing the hang: The first is to crash the hung system and hope that you get a dump that you can analyze; the second is to break into the system with a kernel debugger and analyze the system's activity. Both approaches require prior setup and a reboot. You use the same exploration of system state with both approaches to try and determine the cause of the hang.

To manually crash a hung system, you must first add the DWORD registry value HKLM\System\CurrentControlSet\Services\i8042prt\Parameters\CrashOnCtrlScroll and set it to 1. After rebooting, the i8042 port driver, which is the port driver for PS2 keyboard input, monitors keystrokes in its interrupt service routine (ISR, which is discussed further in Chapter 3) looking for two presses of the scroll lock key while the right control key is depressed. When the driver sees that sequence, it calls *KeBugCheckEx* with the MANUALLY_INITIATED_CRASH (0xE2) stop code that indicates a user-manually initiated crash. When the system reboots, open the crash dump file and apply the techniques mentioned earlier to try and determine why the system was hung (for example, determining what thread was running when the system hung, what the kernel stack indicates was happening, and so on). Note that this works for most hung system scenarios, but it won't work if the i8042 port driver's ISR doesn't execute. (The i8042 port driver's ISR won't execute if all processors are hung as a result of their IRQL being higher than the IRQL of the ISR's IRQL, or if corruption of system data structures extends to interrupt-related code or data.)

Note Manually crashing a hung system by using the support provided in the i8042 port driver does not work with USB keyboards. It works with PS2 keyboards only.

Another way to trigger a crash is if your hardware has a built in "crash" button. (Some high-end servers have this.) In this case, the crash is initiated by signaling the nonmaskable interrupt (NMI) pin of the system's motherboard. To enable this, set the registry DWORD value HKLM\System\CurrentControlSet\Control\CrashControl\NMICrashDump to 1. Then when you press the dump switch, an NMI is delivered to the system and the kernel's NMI interrupt handler calls *KeBugCheckEx*. This works in more cases than the i8042 port driver mechanism because the NMI IRQL is always higher than that of the i8042 port driver interrupt. See *http://www.microsoft.com/whdc/system/CEC/dmpsw.mpsx* for more information.

If you are unable to manually generate a crash dump, you can attempt to break into the hung system by first making the system boot into debugging mode. You do this in one of two ways. You can press the F8 key during the boot and select Debugging Mode, or you can create a debugging-mode boot option in Boot.ini by copying an existing boot entry from the system's Boot.ini and adding the /DEBUG switch. When using the F8 approach, the system will use the default connection (Serial Port COM2 and 19200 Baud). With the /DEBUG option, you must also configure the connection mechanism to be used between the host system running the kernel debugger and the target system booting in debugging mode and then configure the /Debugport and /Baudrate switches appropriately for the connection type. The two connection types are a null modem cable using a serial port or, for Windows XP and Windows Server 2003 systems, an IEEE 1394 (Firewire) cable using 1394 ports on each system. For details on configuring the host and target system for kernel debugging, see the Windows Debugging Tools help file.

When booting in debugging mode, the system loads the kernel debugger at boot time and makes it ready for a connection from a kernel debugger running on a different computer connected through a serial cable or IEEE 1394 cable. Note that the kernel debugger's presence does not affect performance. When the system hangs, run the Windbg or Kd debugger on the connected system, establish a kernel debugging connection, and break into the hung system. This approach will not work if interrupts are disabled or the kernel debugger has become corrupted.

Note Booting a system in debugging mode does not affect performance if it's not connected to another system; however, a system that's configured to automatically reboot after a crash will not do so if it's booted with kernel debugging enabled, because the kernel debugger waits for a connection from another system after a crash.

Instead of leaving the system in its halted state while you perform analysis, you can also use the debugger ".dump" command to create a crash dump file on the host debugger machine. Then you can reboot the hung system and analyze the crash dump offline (or submit it to Microsoft). Note that this can take a long time if you are connected using a serial null modem cable (vs. a higher speed 1394 connection), so you might want to just capture a minidump using the ".dump /m" command. Alternatively, if the target machine is capable of writing a crash dump, you can force it to do so by issuing the ".crash" command from the debugger. This will cause the target machine to create a dump onto its local hard drive that you can examine after the system reboots.

You can cause a hang by running Notmyfault and selecting the Hang option. This causes the Myfault driver to queue a DPC on each processor of the system that executes an infinite loop. Because the IRQL of the processor while executing DPC functions is DPC/dispatch level, the keyboard ISR will respond to the special keyboard crashing sequence.

Once you've broken into a hung system or loaded a manually generated dump from a hung system into a debugger, you should execute the *!analyze* command with the *-hang* option. This causes the debugger to examine the locks on the system and try to determine whether there's a deadlock, and if so, what driver or drivers are involved. However, for a hang like the one that Notmyfault's hang generates, the *!analyze* analysis command will report nothing useful.

If the *!analyze* command doesn't pinpoint the problem, execute *!thread* and *!process* in each of the dump's CPU contexts to see what each processor is doing. (Switch CPU contexts with the ~ command—for example, use *~1* to switch to processor 1's context.) If a thread has hung the system by executing in an infinite loop at an IRQL of DPC/dispatch level or higher, you'll see the driver module in which it has become stuck in the stack trace of the *!thread* command. The stack trace of the crash dump you get when you crash a system experiencing the Notmyfault hang bug looks like this:

```
STACK_TEXT:
f9e66ed8 f9b0d681 000000e2 00000000 00000000 nt!KeBugCheckEx+0x19
f9e66ef4 f9b0cefb 0069b0d8 010000c6 00000000 i8042prt!I8xProcessCrashDump+0x235
f9e66f3c 804ebb04 81797d98 8169b020 00010009 i8042prt!I8042KeyboardInterruptService+0x21c
f9e66f3c fa12e34a 81797d98 8169b020 00010009 nt!KiInterruptDispatch+0x3d
WARNING: Stack unwind information not available. Following frames may be wrong.
ffdff980 8169b288 f9e67000 0000210f 00000004 myfault+0x34a
8054ace4 ffdff980 804ebf58 00000000 0000319c 0x8169b288
8054ace4 ffdff980 804ebf58 00000000 0000319c 0xffdff980
8169ae9c 8054ace4 f9b12b0f 8169ac88 00000000 0xffdff980
…
```

The top few lines of the stack trace reference the routines that execute when you type the i8042 port driver's crash key sequence. The presence of the Myfault driver indicates that it might be responsible for the hang.

Another command that might be revealing is *!locks*, which dumps the status of all executive resource locks. By default, the command lists only resources that are under *contention*, which means that they are both owned and have at least one thread waiting to acquire them. Examine the thread stacks of the owners with the *!thread* command to see what driver they might be executing in.

When There Is No Crash Dump

In this section, we'll address how to troubleshoot systems that for some reason are not recording a crash dump. One reason why a crash dump might not be recorded is if the paging file on the boot volume is too small to hold the dump or if there is not enough free disk space to extract the dump after the reboot. These two cases can easily be remedied by either increasing the size of the paging file or configuring the dump to be saved to a volume with enough disk space to hold the extracted dump.

A third reason why there might not be a crash dump recorded is because the kernel code and data structures needed to write the crash dump have been corrupted at the time of the crash.

As described earlier, this data is checksummed when the system boots, and if the checksum made at the time of the crash does not match, the system does not even attempt to save the crash dump (so as not to risk corrupting data on the disk). So in this case, you need to catch the system as it crashes and then try to determine the reason for the crash.

A final reason occurs when the disk subsystem for the system disk is not able to process disk write requests (a condition that might have triggered the system failure itself). One such condition would be a hardware failure in the disk controller or maybe a cabling issue near the hard disk.

One simple option is to turn off the Automatically Restart option in the Startup And Recovery settings so that if the system crashes, you can examine the blue screen on the console. However, only the most straightforward crashes can be solved from just the blue-screen text.

To perform more in-depth analysis, you need to use the kernel debugger to look at the system at the time of the crash. This can be done by booting the system in debugging mode, which is described in the previous section. When a system booted in debugging mode crashes, instead of painting the blue screen and attempting to record the dump, it will wait forever until a host kernel debugger is connected. In this way, you can see the reason for the crash and perhaps perform some basic analysis using the kernel debugger commands described earlier. As mentioned in the previous section, you can use the .dump command in the debugger to save a copy of the crashed system's memory space for later debugging, thus allowing you to reboot the crashed system and debug the problem offline.

EXPERIMENT: The Blue Screen Screen Saver

A great way to remind yourself of what a blue screen looks like or to fool your office workers and friends is to run the Sysinternals Blue Screen screen saver from www.sysinternals.com. The screen saver simulates authentic looking blue screens that reflect the version of Windows on which you run it, generating all blue screen text using actual system information such as the list of loaded drivers. It also mimics an automatic-reboot, complete with the Windows startup splash screen. Note that unlike other screen savers where a mouse movement dismisses it, the Blue Screen screen saver requires a key press.

Using the following syntax for the Psexec tool from Sysinternals, you can even run the screen saver on another system:

```
psexec \\computername –i –d “c:\sysinternals bluescreen.scr” –s
```

The command requires that you have administrative privilege on the remote system. (You can use the –u and –p Psexec switches to specify alternate credentials.) Make sure that your coworker has a sense of humor!

Glossary

access-control list (ACL) The part of a security descriptor that enumerates who has what access to an object. The owner of an object can change the object's ACL to allow or disallow others access to the object. An ACL is made up of an ACL header and zero or more access-control entry (ACE) structures. An ACL with zero ACEs is called a null ACL and indicates that no user has access to the object.

access token A data structure that contains the security identification of a process or a thread, which includes its security ID (SID), the list of groups that the user is a member of, and the list of privileges that are enabled and disabled. Each process has a primary access token that it inherits by default from its creating process.

add-device routine A routine implemented by drivers that supports Plug and Play. The Plug and Play manager sends a driver notification via this routine whenever a device for which the driver is responsible is detected. In this routine, a driver typically allocates a device object to represent the device.

Address Windowing Extensions (AWE) A mechanism in Windows that allows a 32-bit application to allocate up to 128 GB of physical memory and then map views, or windows, into its 2-GB virtual address space. Using AWE puts the burden of managing mappings of virtual-to-physical memory on the programmer but solves the immediate need of being able to directly access more physical memory than can be mapped at any one time in a 32-bit process address space.

affinity mask A bitmask that specifies the processors on which the thread is allowed to run. The initial thread affinity mask is inherited from the process affinity mask.

aging A process performed on a page that increments a count indicating that the page hasn't been referenced since the last working set trim scan. On a single-processor system, the working set manager tries to remove pages that haven't been accessed recently. It does this by first clearing the accessed bit in the hardware page table entry (PTE) and then later checking the bit to see whether the page has been accessed. If the bit remains clear, the page wasn't accessed between scans and is aged. Later, the age of pages is used to locate candidate pages to remove from the working set.

alertable wait state A thread state that the thread enters either by waiting on an object handle and specifying that its wait is alertable (with the Windows WaitForMultipleObjectsEx function) or by testing directly whether it has a pending APC (using SleepEx). In both cases, if a user-mode APC is pending, the kernel interrupts (alerts) the thread, transfers control to the APC routine, and resumes the thread's execution when the APC routine completes. User-mode APCs are delivered to a thread only when it's in an alertable wait state.

allocation granularity The granularity with which virtual memory is allocated. Windows aligns each region of reserved process address space to begin on an integral boundary defined by the system allocation granularity value, which can be retrieved from the Windows *GetSystemInfo* function. Currently, this value is 64 KB. This size was chosen so that if support were added for future processors with large page sizes (for example, up to 64 KB), the risk of requiring changes to applications that made assumptions about allocation alignment would be reduced. (Windows kernel-mode code isn't subject to the same restrictions; it can reserve memory on a single-page granularity.)

alternative hive A hive that acts as a backup to the crucial SYSTEM hive. The alternate hive is stored in \Winnt\System32\Config as System.alt. Whenever a hive sync flushes dirty sectors to the SYSTEM hive, the hive sync also updates the System.alt hive. If the configuration manager detects that the

SYSTEM hive is corrupt when the system boots, the configuration manager attempts to load the alternate hive. If that hive is usable, it then uses that alternate to update the original SYSTEM hive.

APC queue A queue in which asynchronous procedure calls (APCs) waiting to execute reside. The APC queues (one for user mode and one for kernel mode) are thread-specific—each thread has its own APC queues (unlike the DPC queue, which is processor-wide).

asymmetric multiprocessing (ASMP) A system of processing on a multiprocessor system that typically selects one processor to execute operating system code while other processors run only user code.

asynchronous I/O An I/O model that allows an application to issue an I/O request and then continue executing while the request is completed. This type of I/O can improve an application's throughput because it allows the application to continue with other work while an I/O operation is in progress.

asynchronous procedure call (APC) A function that provides a way for user programs and system code to execute code in the context of a particular user thread (and hence a particular process address space). An APC can be either kernel mode or user mode. (Kernel-mode APCs don't require "permission" from a target thread to run in that thread's context, as user-mode APCs do.)

asynchronous read-ahead with history A method in which the cache manager maintains a history of the last two read requests in the private cache map for the file handle being accessed.

atomic transaction A technique for handling modifications to a database so that system failures don't affect the correctness or integrity of the database. The basic tenet of atomic transactions is that some database operations, called transactions, are all-or-nothing propositions. The separate disk updates that make up the transaction must be executed atomically; that is, once the transaction begins to execute, all its disk updates must be completed. If a system failure interrupts the transaction, the part that has been completed must be undone, or rolled back. The rollback operation returns the database to a previously known and consistent state, as if the transaction had never occurred. *See also* transaction.

attribute list A special kind of file attribute in an NTFS file header that contains additional attributes. The attribute list is created if a particular file has too many attributes to fit in the master file table (MFT) record. The attribute list attribute contains the name and type code of each of the file's attributes and the file reference of the MFT record where the attribute is located.

authentication packages Dynamic-link libraries (DLLs) that run in the context of the LSASS process and that implement Windows authentication policy. An authentication DLL is responsible for checking whether a given username and password match, and if so, returning to LSASS information detailing the user's security identity. Windows authentication packages include Kerberos and MSV1_0.

automatic working set trimming A technique the memory manager uses when physical memory runs low to increase the amount of free memory available in the system.

bad-cluster file A system file (filename $BadClus) that records any bad spots on the disk volume.

balance set manager A system thread that wakes up once per second to check and possibly initiate various scheduling and memory management–related events.

basic disk A disk that relies on the MS-DOS-style partitioning scheme. *See also* dynamic disk.

bitmap file A system file (filename $Bitmap) in which NTFS records the allocation state of the volume. The data attribute for the bitmap file contains a bitmap, each of whose bits represents a cluster on the volume, identifying whether the cluster is free or has been allocated to a file.

boot code Instructions executed when a system is booted.

boot device drivers Device drivers necessary to boot the system.

boot file A system file (filename $Boot) that stores the Windows bootstrap code.

boot partition The partition that contains core operating system files. The boot partition is identified by the system at startup. The code in a master boot record (MBR) scans the primary partition table until it locates a partition containing a flag that signals the partition is bootable. When the MBR finds at least one such flag, it reads the first sector from the flagged partition into memory and transfers control to code within the partition.

boot sector The first sector of the partition marked as active and from which the MBR boots. The boot sector contains information identifying the partition's file system format and structure.

boot volume The volume that contains the Windows operating system and its support files. The boot volume can be, but does not have to be, the same as the system volume.

bus driver Driver that services a bus controller, adapter, bridge, or any device that has child devices. Bus drivers are required drivers, and Microsoft generally provides them; each type of bus (such as PCI, PCMCIA, and USB) on a system has one bus driver.

cache manager The component of the Windows executive that provides systemwide caching services for NTFS and other file system drivers, including network file system drivers (servers and redirectors).

careful write A technique for constructing a file system's I/O and caching support. *See also* write-through.

change journal An internal file where an NTFS file system can record information that allows applications to efficiently monitor file and directory changes. A change journal is usually large enough to virtually guarantee that applications get a chance to process changes without missing any.

checked build A special debug version of Windows that is available only as part of the MSDN Professional (or Universal) subscription. The checked build is created by compiling the Windows sources with the compile-time flag DEBUG defined.

checkpoint record A record that helps NTFS determine what processing would be needed to recover a volume if a crash were to occur immediately. This record also includes redo and undo information.

class driver A type of kernel-mode device driver that implements the I/O processing for a particular class of devices, such as disk, tape, or CD-ROM.

clock algorithm A working set page replacement policy implemented on uniprocessor systems, similar to a least recently used policy (as implemented in most versions of UNIX).

clock interrupt handler A system routine that updates the system time and then decrements a counter that tracks how long the current thread has run.

cluster factor The cluster size on a volume, which is established when a user formats the volume with either the format command or the Disk Management Microsoft Management Console (MMC) snap-in.

cluster remapping A process in which NTFS dynamically retrieves good data from a cluster with a bad sector, allocates a new cluster, and copies the data to the new cluster.

clustering A method by which the memory manager resolves a page fault by reading into memory several pages near the page explicitly accessed.

clusters Same-size allocation units into which a volume is divided. Each cluster must be uniquely numbered using 16 bits.

collided page fault A fault that occurs when another thread or process faults a page that is currently being in-paged.

commitment The process by which the memory manager keeps track of private committed memory usage on a global basis.

common model A set of classes in the Common Information Model (CIM) that represent objects that are specific to management areas of a system but independent of a particular implementation. These classes are considered an extension of the CIM core model. *See also* core model.

complete memory dump A memory dump that contains all of physical memory at the time of the crash. This type of dump requires that a page file be at least the size of physical memory. Because it can require an inordinately large page file on large memory systems, this type of dump file is the least common. Windows NT 4 supported only this type of crash dump file.

completion port A mechanism to deliver I/O completion notification to threads. Once a file is associated with a completion port, any asynchronous I/O operations that complete on the file result in a completion packet being queued to the completion port. A thread can wait for any outstanding I/Os to complete on multiple files simply by waiting for a completion packet to be queued to the completion port. With completion ports, concurrency, or the number of threads that an application has actively servicing client requests, is controlled with the aid of the system.

configuration manager A major component of the executive that's responsible for implementing and managing the system registry.

container object A namespace object that can hold other objects, including other container objects. Examples of containers are directories in the file system namespace and keys in the registry namespace.

context switch The procedure of saving the volatile machine state associated with a running thread, loading another thread's volatile state, and starting the new thread's execution.

control objects A set of kernel objects that establishes semantics for controlling various operating system functions. This set includes the kernel process object, the asynchronous procedure call (APC) object, the deferred procedure call (DPC) object, and several objects the I/O system uses, such as the interrupt object.

core model A set of classes in the Common Information Model (CIM) provided as part of the WBEM standard. These classes are CIM's basic language and represent objects that apply to all areas of management. *See also* common model.

crash dump A record of system memory at the time of a crash that can help you figure out which component caused the crash.

critical sections An intra-process mutual exclusion synchronization primitive.

deadlock detection A Driver Verifier option in Windows XP and Windows Server 2003 that monitors the use of spin locks, fast mutexes, and mutexes, looking for patterns that could result in deadlock.

deferred procedure call (DPC) A routine that performs most of the work involved in handling a device interrupt after the interrupt service routine (ISR) executes. The DPC routine executes at an interrupt request level (IRQL) that is lower than that of the ISR to avoid blocking other interrupts unnecessarily. A DPC routine initiates I/O completion and starts the next queued I/O operation on a device.

deferred procedure call (DPC) object A kernel control object that describes a request to defer interrupt processing to DPC/dispatch level. (*See* interrupt request levels [irqls].) This object isn't visible to user-mode programs but is visible to device drivers and other system code. The most important piece of information the DPC object contains is the address of the system function that the kernel will call when it processes the DPC interrupt.

Deferred ready A state used for threads that have been selected to run on a specific processor but have not yet been scheduled. This new state in Windows Server 2003 exists so that the kernel can minimize the amount of time the systemwide lock on the scheduling database is held.

demand-paging policies A fetch policy that loads a page into physical memory only when a page fault occurs. In a demand-paging system, a process incurs many page faults when its threads first begin executing because the threads reference the initial set of pages they need to get going. Once this set of pages is loaded into memory, the paging activity of the process decreases.

desired access rights The accesses desired by a thread opening an object.

device drivers Loadable kernel-mode modules (typically ending in .sys) that interface between the I/O system and the relevant hardware. Device drivers on Windows don't manipulate hardware devices directly, but rather they call parts of the hardware application layer (HAL) to interface with the hardware.

device ID A device identifier reported to the Plug and Play manager. The identifiers are bus-specific; for a USB bus, an identifier consists of a vendor ID (VID) for the hardware vendor that made the device and a product ID (PID) that the vendor assigned to the device.

device instance ID (DIID) An identifier consisting of a device ID and an instance ID that the Plug and Play manager uses to locate the device's key in the enumeration branch of the registry (HKLM\SYSTEM\CurrentControlSet\Enum).

device object A data structure that represents a physical, logical, or virtual device on the system and describes its characteristics, such as the alignment it requires for buffers and the location of its device queue to hold incoming I/O request packets.

device-specific module (DSM) Third-party drivers used to manage details of the path management, such as load balancing policies that choose which path to route requests and error detection mechanisms to inform Windows when a path fails.

device tree An internal tree the Plug and Play manager creates that represents the relationships between devices. Nodes in the tree are called devnodes. *See also* devnode.

devnode A node in a device tree. A devnode contains information about the device objects that represent the device as well as other Plug and Play–related information the Plug and Play manager stores. *See also* device tree.

direct memory access (DMA) A third interface provided by the cache manager to cached data. The DMA functions are used to read from or write to cache pages without intervening buffers, such as when a network file system is doing a transfer over the network.

dirty page threshold The number of pages that the system cache keeps in memory before waking up the cache manager's lazy writer system thread to write out pages back to the disk. This value is computed at system initialization time and depends on physical memory size and the value of the registry value HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management\LargeSystemCache.

discretionary access control Allows the owner of a resource to determine who can access the resource and what they can do with it. The owner grants rights that permit various kinds of access to a user or to a group.

disk group Dynamic disks that share a common database. VERITAS's commercial volume-management software for Windows includes disk groups, but the Windows Logical Disk Manager (LDM) implementation includes only one disk group.

dispatch code Instructions of assembly language code stored in an interrupt object when it is initialized. When an interrupt occurs, this code is executed.

dispatch routines The main functions that a device driver provides. Some examples of dispatch routines are open, close, read, and write, and any other capabilities the device, file system, or network supports. When called on to perform an I/O operation, the I/O manager generates an IRP and calls a driver through one of the driver's dispatch routines.

dispatcher A set of routines in the kernel that implement Windows scheduling. Windows doesn't have a single "scheduler" module or routine—the code is spread throughout the kernel in which scheduling-related events occur.

dispatcher database A set of data structures the kernel maintains to make thread-scheduling decisions. The dispatcher database keeps track of which threads are waiting to execute and which processors are executing which threads. *See also* dispatcher ready queue.

dispatcher header A data structure that contains the object type, the signaled state, and a list of the threads waiting on that object.

dispatcher objects A set of kernel objects that incorporate synchronization capabilities and alter or affect thread scheduling. The dispatcher objects include the kernel thread, mutex (called mutant internally), event, kernel event pair, semaphore, timer, and waitable timer.

dispatcher ready queue The most important structure in the dispatcher database (located at KiDispatcherReadyListHead). The dispatcher ready queue is really a series of queues, one queue for each scheduling priority. The queues contain threads that are in the ready state, waiting to be scheduled for execution.

display driver Driver that translates device-independent graphics requests into device-specific requests. The device-specific requests are then paired with a kernel-mode video miniport driver to complete video display support. A display driver is responsible for implementing drawing operations, either by writing directly to the frame buffer or by communicating with the graphics accelerator chip on the controller.

driver object Data structure that represents an individual driver in the system and records for the I/O manager the address of each of the driver's dispatch routines (entry points).

driver support routines Routines that device drivers call to accomplish their I/O requests.

dynamic disk A disk that supports multipartition volumes, providing a more flexible partitioning scheme than that of a basic disk. *See also* basic disk.

dynamic-link library (DLL) A set of callable subroutines linked as a binary image that can be dynamically loaded by applications that use them.

environment subsystems User processes and DLLs that expose the native operating system services to user applications, thus providing an operating system environment, or personality. Windows 2000 ships with two environment subsystems: Windows and POSIX (Windows NT 4.0 had one for OS/2 1.2). Windows XP and later only ship with Windows, but an enhanced POSIX subsystem is included with the free Services for Unix product from Microsoft.

event An object with a persistent state (signaled or not signaled) that can be used for synchronization; also, a system occurrence that triggers an action.

exception A synchronous condition that results from the execution of a particular instruction. Running a single program with the same data under the same conditions can reproduce exceptions.

exception dispatcher A kernel module that services all exceptions, except those simple enough to be resolved by the trap handler. The exception dispatcher's job is to find an exception handler that can "dispose of" the exception.

executive The upper layer of Ntoskrnl.exe. (The kernel is the lower layer.) The executive contains the base operating system services, such as the process and thread manager, the virtual memory manager, the memory manager, the security reference monitor, the I/O system, and the cache manager. *See also* kernel.

executive objects Objects implemented by various components of the executive (such as the process manager, memory manager, I/O subsystem, and so on). The executive objects and object services are primitives that the environment subsystems use to construct their own versions of objects and other resources. Because executive objects are typically created either by an environment subsystem on behalf of a user application or by various components of the operating system as part of their normal operation, many of them contain (encapsulate) one or more kernel objects. *See also* kernel objects.

executive resources Resources that provide both exclusive access (such as a mutex) as well as shared read access (multiple readers sharing read-only access to a structure). Because executive resources are available only to kernel-mode code, they aren't accessible from the Windows API.

executive support routines Functions in Ntoskrnl.exe that provide services to device drivers.

extended partition A special partition type that contains a master boot record (MBR) with its own partition table. By using extended partitions, Microsoft's operating systems overcome the apparent limit of four partitions per disk. In general, the recursion that extended partitions permit can continue indefinitely, which means that no upper limit exists to the number of possible partitions on a disk. *See also* partition.

fast I/O A means of reading or writing a cached file without going through the work of generating an I/O request packet (IRP).

fast LPC A special interprocess communication facility used to send messages between threads.

file mapping objects Windows APO underlying primitives in the memory manager that are used to implement shared memory (called section objects internally). *See also* section object.

file reference A 64-bit value that identifies a file on an NTFS volume. The file reference consists of a file number and a sequence number. The file number corresponds to the position of the file's file record in the master file table minus 1 (or to the position of the base file record minus 1 if the file has more than one file record).

file system driver (FSD) A type of kernel-mode device driver that accepts I/O requests to files and satisfies the requests by issuing its own, more explicit, requests to physical device drivers. *See also* local file system driver (FSD), network file system driver (FSD).

file system filter driver A type of kernel-mode device driver that intercepts I/O requests, performs additional processing, and passes them on to lower-level drivers.

file system format Defines the way that file data is stored on storage media and impacts a file system's features. For example, a format that doesn't allow user permissions to be associated with files and directories can't support security. A file system format can also impose limits on the sizes of files and storage devices that the file system supports.

filter device object (FiDO) A device object that can be part of a devnode. One or more optional FiDOs can layer either between the physical device object (PDO) and the functional device object (FDO) or above the FDO. *See also* devnode, functional device object (FDO), physical device object (PDO).

filter driver *See* file system filter driver.

frame-based exception handlers The mechanism that permits each stack frame in a call stack to have its own exception handler declared.

foreground application The process that owns the thread that owns the window that's in focus.

free build The version of the Windows system that can be purchased as a retail product. It is built with full compiler optimizations turned on and has internal symbol table information stripped out from the images. *See also* checked build.

function driver The main device driver that provides the operational interface for its device. It is a required driver unless the device is used raw (an implementation in which I/O is done by the bus driver and any bus filter drivers, such as SCSI PassThru). A function driver is the driver that knows the most about a particular device and is usually the only driver that accesses device-specific registers.

functional device object (FDO) A device object that is a required part of a devnode. The function driver that the Plug and Play manager loads to manage a detected device creates the FDO. An FDO represents the logical interface to a device. *See also* devnode, filter device object (FiDO), physical device object (PDO).

granted access rights The accesses granted to a thread by the security reference monitor as the result of a successful object open.

Graphical Identification and Authentication (GINA) A user-mode DLL that runs in the Winlogon process and that Winlogon uses to obtain a user's name and password or smart card PIN. The standard GINA is \Winnt\System32\Msgina.dll.

handle An object identifier. A process receives a handle to an object when it creates or opens an object by name. Referring to an object by its handle is faster than using its name because the object manager can skip the name lookup and find the object directly.

handle table A table that contains pointers to all the objects that the process has opened a handle to. Handle tables are implemented as a three-level scheme, similar to the way that the x86 memory management unit implements virtual-to-physical address translation.

hardware abstraction layer (HAL) A loadable kernel-mode module (Hal.dll) that provides the low-level interface to the hardware platform on which Windows is running. The HAL hides hardware-dependent details such as I/O interfaces, interrupt controllers, and multiprocessor communication mechanisms—any functions that are architecture-specific and machine-dependent.

hardware device drivers Device drivers that manipulate hardware to write output to or retrieve input from a physical device or network. There are many types of hardware device drivers, such as bus drivers, human interface drivers, mass storage drivers, and so on.

hash A statistically unique value that is generated from a block of data (for example, a file) using cryptographic algorithms. Because different data results in different hashes, hashes can be used to detect changes to data from corruption or tampering. The Windows driver signing facility uses hashes.

heap A region of one or more pages that can be subdivided and allocated in smaller chunks by a set of functions provided by the heap manager.

heap manager A set of functions that allocate and deallocate variable amounts of memory (not on a page-size granularity). The heap manager functions exist in two places: Ntdll.dll and Ntoskrnl.exe. The subsystem APIs (such as the Windows heap APIs) use the copy in Ntdll, and various executive components and device drivers use the copy in Ntoskrnl.

hive One of a number of files stored on disk that contain registry information. Each hive contains a registry tree, which has a key that serves as the root or starting point of the tree.

hyperspace A special region used to map the process working set list and to temporarily map other physical pages for such operations as zeroing a page on the free list (when the zero list is empty and a zero page is needed), invalidating page table entries in other page tables (such as when a page is removed from the standby list), and on process creation setting up a new process's address space.

I/O completion routine A routine implemented by a layered driver that will notify the driver when a lower-level driver finishes processing an I/O request packet (IRP). For example, the I/O manager calls a file system driver's I/O completion routine after a device driver finishes transferring data to or from a file. The completion routine notifies the file system driver about the operation's success, failure, or cancellation, and it allows the file system driver to perform cleanup operations.

I/O request packet (IRP) A data structure that controls how the I/O operation is processed at each stage. Most I/O requests are represented by an IRP, which travels from one I/O system component to another.

I/O subsystem API The internal executive system services (such as NtReadFile and NtWriteFile) that subsystem DLLs call to implement a subsystem's documented I/O functions.

I/O system The Windows executive component that accepts I/O requests (from both user-mode and kernel-mode callers) and delivers them, in a different form, to I/O devices.

ideal processor The preferred processor that a particular thread should run on.

idle summary A bitmask (KiIdleSummary) in which each set bit represents an idle processor.

impersonation A capability that allows threads to have a different access token than that of the process.

initialization routine A driver routine that the I/O manager executes when it loads the driver into the operating system. The initialization routine creates system objects that the I/O manager uses to recognize and access the driver.

in-paging I/O A condition that occurs when a read operation must be issued to a file (paging or mapped) to satisfy a page fault. The in-page I/O operation is synchronous—the thread waits on an event until the I/O completes—and isn't interruptible by asynchronous procedure call (APC) delivery.

instancing The term for making separate copies of the same parts of a namespace. Instancing \DosDevices makes it possible for each user to have different drive letters and Windows objects such as serial ports.

intelligent file read-ahead A technique that predicts what data the calling thread is likely to read next based on the data it's currently reading.

inter-processor interrupt (IPI) An interrupt the kernel issues to request that another processor perform an action, such as dispatching a particular thread for execution or updating its translation look-aside buffer cache.

interrupt An asynchronous event (one that can occur at any time) that is unrelated to what the processor is executing. Interrupts are generated primarily by I/O devices, processor clocks, or timers, and they can be enabled or disabled.

interrupt dispatch table (IDT) A data structure that Windows uses to locate the routine that will handle a particular interrupt. The interrupt request level (IRQL) of the interrupting source serves as a table index, and table entries point to the interrupt-handling routines.

interrupt dispatcher A submodule of the kernel's trap handler that responds to interrupts.

interrupt object A kernel control object that allows device drivers to register interrupt service routines (ISRs) for their devices. An interrupt object contains all the information the kernel needs to associate a device ISR with a particular level of interrupt, including the address of the ISR, the interrupt request level (IRQL) at which the device interrupts, and the entry in the kernel's interrupt dispatch table with which the ISR should be associated.

interrupt request (IRQ) A value identifying an interrupt. On x86 systems, external I/O interrupts come into one of the lines on an interrupt controller. The controller in turn interrupts the processor on a single line. Once the processor is interrupted, it queries the controller to get the interrupt request (IRQ). The interrupt controller translates the IRQ to an interrupt number, uses this number as an index into the interrupt dispatch table (IDT), and transfers control to the appropriate interrupt dispatch routine. At system boot time, Windows fills in the IDT with pointers to the kernel routines that handle each interrupt and exception.

interrupt request levels (IRQLs) An interrupt priority scheme imposed by Windows. The kernel represents IRQLs internally as a number from 0 through 31 (0 to 15 on 64-bit Windows), with higher numbers representing higher-priority interrupts. Although the kernel defines the standard set of IRQLs for software interrupts, the HAL maps hardware-interrupt numbers to the IRQLs.

interrupt service routine (ISR) A device driver routine that the kernel's interrupt dispatcher transfers control to when a device issues an interrupt. In the Windows I/O model, ISRs run at a high device interrupt request level (IRQL), so they perform as little work as possible to avoid blocking lower-level interrupts unnecessarily. An ISR queues a deferred procedure call (DPC), which runs at a lower IRQL, to execute the remainder of interrupt processing. Only drivers for interrupt-driven devices have ISRs; a file system, for example, doesn't have one.

job object A nameable, securable, shareable object in Windows that controls certain attributes of processes associated with the job. A job object's basic function is to allow groups of processes to be managed and manipulated as a unit. The job object also records basic accounting information for all processes associated with the job and for all processes that were associated with the job but have since terminated.

journaling A logging technique originally developed for transaction processing that a recoverable file system such as NTFS uses to ensure volume consistency.

kernel The lowest layer in Ntoskrnl.exe. The kernel, a component of the executive, determines how the operating system uses the processor or processors and ensures that they are used prudently. The kernel provides thread scheduling and dispatching, trap handling and exception dispatching, interrupt handling and dispatching, and multiprocessor synchronization. *See also* executive.

kernel debugger A tool used to debug drivers, troubleshoot hung systems, and examine crash dumps. It is also a useful tool for investigating Windows internals because it can display internal Windows system information not visible through any standard utility. (The LiveKd tool on the companion CD allows the use of the standard kernel debuggers on a live system.)

kernel handle table A table (referenced internally with the name ObpKernelHandleTable) whose handles are accessible only from kernel mode and in any process context. A kernel-mode function can reference the handles in this table in any process context with no performance impact.

kernel memory dump A type of memory dump (the default on Windows Server systems) that contains only the kernel-mode read/write pages present in physical memory at the time of the crash. A kernel memory dump doesn't contain pages belonging to user processes. Because only kernel-mode code can directly cause Windows to crash, however, it's unlikely that user process pages are necessary to debug a crash. There is no way to predict the size of a kernel memory dump because its size depends on the amount of kernel-mode memory allocated by the operating system and drivers present on the machine.

kernel mode A privileged mode of code execution in a processor in which all memory is totally accessible and all CPU instructions can be issued. Operating system code (such as system services and device drivers) runs in kernel mode. *See also* user mode.

kernel objects A primitive set of objects implemented by the Windows kernel. These objects aren't visible to user-mode code but are created and used only within the executive. Kernel objects provide fundamental capabilities, such as synchronization, on which executive objects are built. *See also* executive objects.

kernel streaming filter drivers Kernel-mode drivers chained together to perform signal processing on data streams, such as recording or displaying audio and video.

kernel-mode device driver The only type of driver that can directly control and access hardware devices.

kernel-mode graphics driver A Windows subsystem display or print device driver that translates device-independent graphics (GDI) requests into device-specific requests.

key A mechanism to refer to data in the registry. Although keys appear in the object manager namespace, the registry manages them in a way similar to how it manages file objects. Zero or more key values are associated with a key object; key values contain data about the key.

key control block A structure that stores the full pathname of a registry key, includes the cell index of the key node that the control block refers to, and contains a flag that notes whether the configuration manager needs to delete the key cell that the key control block refers to when the last handle for the key closes. In Windows, all key control blocks are in an alphabetized binary tree to enable quick searches for existing key control blocks by name. A key object points to its corresponding key control block, so if two applications open the same registry key, each will receive a key object and both key objects will point to a common key control block.

key object An object type the configuration manager defines to integrate the registry's namespace with the kernel's general namespace.

keyed event A synchronization object new to Windows XP.

last known good control set A copy of the critical boot-time information under the registry key HKLM\SYSTEM\CurrentControlSet, made when a user successfully logs on. The last known good control set can be selected at boot time in case configuration changes made to the registry result in the system not being able to boot successfully.

lazy IRQL A performance optimization that avoids Programmable Interrupt Controller (PIC) accesses. When the interrupt request level (IRQL) is raised, the HAL notes the new IRQL internally instead of changing the interrupt mask. If a lower-priority interrupt subsequently occurs, the HAL sets the interrupt mask to the settings appropriate for the first interrupt and postpones the lower-priority interrupt until the IRQL is lowered. Thus, if no lower-priority interrupts occur while the IRQL is raised, the HAL doesn't need to modify the PIC.

lazy writer A set of system threads that call the memory manager to flush cache contents to disk as a background activity (asynchronous disk writing). The cache manager optimizes disk I/O by using its lazy writer.

legacy drivers Device drivers written for Microsoft Windows NT but that run unchanged on Windows 2000 and later. They are differentiated from other Windows drivers in that they don't support power management or work with the Windows Plug and Play manager.

local file system driver (FSD) A driver that manages volumes directly connected to the computer. *See also* file system driver.

local kernel debugging The ability to connect the kernel debugger to the local running system to view internal state (as opposed to connecting to a target system booted in debugging mode using a null modem or 1394 cable).

local procedure call (LPC) An interprocess communication facility for high-speed message passing (not available through the Windows API but rather through an internal mechanism available only to Windows operating system components). LPCs are typically used between a server process and one or more client processes of that server. An LPC connection can be established between two user-mode processes or between a kernel-mode component and a user-mode process.

local security authority (LSA) server A user-mode process running the image \Winnt\System32\LSASS.exe that is responsible for the local system security policy (such as which users are allowed to log on to the machine, password policies, privileges granted to users and groups, and the system security auditing settings), user authentication, and sending security audit messages to the Event Log. The LSA service (Lsasrv - \Winnt\System32\ Lsasrv.dll), a library that LSASS loads, implements most of this functionality.

local security authority (LSA) server policy database A database (stored in the registry under HKLM\SECURITY) that contains the system security policy settings. This database includes such information as what domains are trusted to authenticate logon attempts, who has permission to access the system and how (interactive, network, and service logons), who is assigned which privileges, and what kind of security auditing is to be performed.

Local Security Authority Subsystem (LSASS) The system user-mode process responsible for authentication of accounts accessing a computer.

Local System account A predefined local account that is used to start a service and provide the security context for that service. The name of the account is NT AUTHORITY\System. This account does not have a password, and any password information that you supply is ignored. The Local System account has full access to the system, including the directory service on domain controllers. Because the Local System account acts as a computer on the network, it has access to network resources.

log file A metadata file (filename $LogFile) NTFS uses to record all operations that affect the NTFS volume structure, including file creation or any commands, such as Copy, that alter the directory structure. The log file is used to recover an NTFS volume after a system failure.

log hive A registry hive the configuration manager uses to make sure that a nonvolatile registry hive (one with an on-disk file) is always in a recoverable state. Each nonvolatile hive has an associated log hive, which is a hidden file with the same base name as the hive and a .log extension. *See also* hive.

logging A transaction-processing technique NTFS uses to maintain file system integrity in case of system crashes or other failures. In NTFS logging, the suboperations of any transaction that alters important file system data structures are recorded in a log file before they are carried through on the disk so that if the system crashes, partially completed transactions can be redone or undone when the system comes back on line.

logical cluster numbers (LCNs) The numbers of all clusters from the beginning of the volume to the end with which NTFS refers to physical locations on a disk. To convert an LCN to a physical disk address, NTFS multiplies the LCN by the cluster factor to get the physical byte offset on the volume, as the disk driver interface requires.

logical prefetcher The kernel component that monitors boot and application startup file access and that preemptively reads that data to speed up subsequent booting and application startups.

logical sequence numbers (LSNs) The numbers that NTFS uses to identify records written to the log file.

logon process A user-mode process running Winlogon.exe that is responsible for capturing the username and password, sending them to the local security authority server for verification, and creating the initial process in the user's session.

look-aside list A fast memory allocation mechanism that contains only fixed-sized blocks. Look-aside lists can be either pageable or nonpageable, so they are allocated from paged or nonpaged pool.

LPC facility Local procedure call interprocess communication support in the kernel.

mapped file I/O The ability to view a file residing on disk as part of a process's virtual memory. A program can access the file as a large array without buffering data or performing disk I/O. The program accesses memory, and the memory manager uses its paging mechanism to load the correct page from the disk file. If the application writes to its virtual address space, the memory manager writes the changes back to the file as part of normal paging.

mask The process whereby interrupts wait for an executing thread to lower the IRQL before the interrupt is processed. Interrupts from a source with an IRQL above the current level interrupt the processor, whereas interrupts from sources with IRQLs equal to or below the current level are masked until an executing thread lowers the IRQL.

master file table (MFT) The heart of the NTFS volume structure. The MFT is implemented as an array of file records. The size of each file record is fixed at 1 KB, regardless of cluster size.

memory manager The Windows executive component that implements demand-paged virtual memory, giving each process the illusion that it has a large virtual address space (while mapping a subset of that address space to physical memory).

metadata Data that describes the files on a disk; also called volume structure data.

metadata files A set of files in each NTFS volume that contains the information used to implement the file system structure.

MFT mirror An NTFS metadata file (filename $MFTMirr) located in the middle of the disk called that contains a copy of the first few rows of the master file table.

miniport driver A type of kernel-mode device driver that maps a generic I/O request to a type of port into an adapter type, such as a specific SCSI adapter.

mirrored volume A volume on which the contents of a partition on one disk are duplicated in an equal-sized partition on another disk. Mirrored volumes are sometimes referred to as RAID level 1 (RAID-1).

mirror set A technique by which the contents of a partition on one disk are duplicated in an equal-size partition on another disk.

modified page writer A thread in the virtual memory manager that is responsible for limiting the size of the modified page list by writing pages to their backing store locations when the list becomes too big. The modified page writer consists of two system threads: one to write out modified pages (MiModifiedPageWriter) to the paging file and a second one to write modified pages to mapped files (MiMappedPageWriter).

mount A technique NTFS uses when it first accesses a volume; in this context, to mount means to prepare the volume for use. To mount the volume, NTFS looks in the boot file to find the physical disk address of the master file table.

mount points A mechanism that permits the linking of volumes through directories on NTFS volumes, which makes volumes accessible with no drive-letter assignment. Reparse points in NTFS make mount points possible.

MSDN Microsoft Developer Network, Microsoft's support program for developers. MSDN offers three CD-ROM subscription programs: MSDN Library, Professional, and Universal. For more information, see msdn.microsoft.com.

multipartition volumes Objects that represent sectors from multiple partitions and that file system drivers manage as a single unit. Multipartition volumes offer performance, reliability, and sizing features that simple volumes don't.

multipathing Where more than one set of hardware exists between the computer and a disk so that if a path fails, the system can still access the disk via an alternate path.

mutant Internal name for a mutex.

mutex A synchronization mechanism used to serialize access to a resource.

mutual exclusion A means whereby only one thread or processor is allowed access to a resource at any given time.

name retention The first phase of object retention, which the object manager implements. Name retention is controlled by the number of open handles to an object that exist. Every time a process opens a handle to an object, the object manager increments the open handle counter in the object's header. As processes finish using the object and close their handles to it, the object manager decrements the open handle counter. When the counter drops to 0, the object manager deletes the object's name from its global namespace. This deletion prevents new processes from opening a handle to the object.

native application An application that uses only system service APIs provided by Ntdll and that isn't a client of the Windows subsystem. Smss (Session Manager) is an example of a native application.

network file system driver (FSD) A driver that allows users to access data volumes connected to remote computers. *See also* file system driver.

network logon service A user-mode service inside the Services.exe process that responds to network logon requests. Authentication is handled as local logons are, by sending them to the LSASS process for verification.

network redirectors and servers File system drivers that transmit remote I/O requests to a machine on the network and receive such requests, respectively.

nonpaged pool Memory pool that consists of ranges of system virtual addresses that are guaranteed to be resident in physical memory at all times and thus can be accessed from any address space without incurring paging I/O. Nonpaged pool is created at system initialization and is used by kernel-mode components to allocate system memory.

Ntdll.dll A special system-support library primarily for the use of subsystem DLLs that contains system service dispatch stubs to Windows executive system services and internal support functions used by subsystems, subsystem DLLs, and other native images.

Ntkrnlmp.exe The executive and kernel for multiprocessor systems.

Ntoskrnl.exe The executive and kernel for uniprocessor systems.

object In the Windows executive, a single, run-time instance of a statically defined object type.

object attribute A field of data in an object that partially defines the object's state.

object directory A container object for other objects. The object directory is used to implement the hierarchical namespace within which other object types are stored.

object handle An index into a process-specific handle table, pointed to by the executive process (EPROCESS) block.

object manager The Windows executive component responsible for creating, deleting, protecting, and tracking objects. The object manager centralizes resource control operations that would otherwise be scattered throughout the operating system.

object methods The means for manipulating objects, usually to read or change the object attributes. For example, the open method for a process would accept a process identifier as input and return a pointer to the object as output.

object reuse protection A means of preventing users from seeing data that another user has deleted or from accessing memory that another user previously used and then released. Object reuse protection prevents potential security holes by initializing all objects, including files and memory, before they are allocated to a user.

object type A system-defined data type, including services that operate on instances of the data type and a set of object attributes; sometimes called an *object class*.

page directory A page the memory manager creates to map the location of all page tables for that process. Each process has a single page directory.

page directory entries (PDEs) The page directory is composed of PDEs, each of which is 4 bytes long and describes the state and location of all the possible page tables for that process.

page fault A reference to an invalid page. The kernel trap handler dispatches this kind of fault to the memory manager fault handler (MmAccessFault) to resolve.

page file backed section A section object that is mapped to committed memory.

page file quota A limit on the number of committed pages a process can consume—not necessarily page file space.

page frame database A database that describes the state of each page in physical memory. Pages are in one of eight states: active (also called valid), transition, standby, modified, modified-no-write, free, zeroed, or bad.

page frame number (PFN) database Describes the state of each page in physical memory.

page table A page of mapping information (made up of an array of page table entries) the operating system constructs that describes the location of the virtual pages in a process address space. Because Windows provides a private address space for each process, each process has its own set of process page tables to map that private address space because the mappings will be different for each process. The page tables that describe system space are shared among all processes.

page table entry (PTE) An entry in a process's page table that contains the address to which the virtual address is mapped. The page can be in physical memory or it can be on disk.

paged pool A region of virtual memory in system space that can be paged in and out of the system process's working set. Paged pool is created at system initialization and is used by kernel-mode components to allocate system memory. Uniprocessor systems have two paged pools; multiprocessor systems have four. Having more than one paged pool reduces the frequency of system code blocking on simultaneous calls to pool routines.

paging The process of moving memory contents to disk, freeing physical memory so that it can be used for other processes or for the operating system itself. Because most systems have much less physical memory than the total virtual memory in use by the running processes (2 GB or 3 GB for each process), the memory manager transfers, or pages, some of the memory contents to disk.

partition A discrete area of the hard disk in Microsoft operating systems. File systems (such as FAT and NTFS) format each partition into a volume. A hard disk can contain up to four primary partitions. *See also* extended partition.

partition table A part of the master boot record (MBR) that consists of four entries that define the locations of as many as four primary partitions on a disk. The partition table also records a partition's type. Numerous predefined partition types exist, and a partition's type specifies which file system the partition includes.

Physical Address Extension (PAE) A memory-mapping mode included in all Intel x86 processors since the Pentium Pro. With the proper chipset, the PAE mode allows access to up to 64 GB of physical memory. When the x86 executes in PAE mode, the memory management unit (MMU) divides virtual addresses into four fields.

physical device object (PDO) A device object that's a required part of a devnode. The PDO represents the physical interface to a device. *See also* devnode.

Plug and Play (PnP) manager A major component of the executive that determines which drivers are required to support a particular device and loads those drivers. The PnP manager retrieves the hardware resource requirements for each device during enumeration. Based on the resource requirements of each device, the PnP manager assigns the appropriate hardware resources such as I/O ports, IRQs, DMA channels, and memory locations. It is also responsible for sending proper event notification for device changes (addition or removal of a device) on the system.

port driver A type of kernel-mode device driver that implements the processing of an I/O request specific to a type of I/O port, such as SCSI.

port object A single executive object a local procedure call (LPC) exports to maintain the state needed for communication.

power manager A major component of the executive that coordinates power events and generates power management I/O notifications to device drivers. When the system is idle, the power manager can be configured to reduce power consumption by putting the CPU to sleep. Changes in power consumption by individual devices are handled by device drivers but are coordinated by the power manager.

printer driver A driver that translates device-independent graphics requests to printer-specific commands. These commands are then typically forwarded to a kernel-mode port driver such as the parallel port driver (Parport.sys) or the USB printer port driver (Usbprint.sys).

private cache map A structure that contains the location of the last two reads so that the cache manager can perform intelligent read-ahead.

process The virtual address space and control information necessary for the execution of a set of thread objects.

process and thread manager The services in the executive that export the process and thread object. These services use primitive functions in the kernel to create processes and threads.

process ID A unique identifier for a process (internally called a client ID).

process working set The subset of a process's virtual address space that is resident and owned by the running process. *See also* system working set.

processor affinity The set of processors a thread is permitted to run on.

processor control register (PCR) A data structure that along with its extension, the processor control block (PRCB), contains information about the state of each processor in the system, such as the current interrupt request level (IRQL), a pointer to the hardware interrupt dispatch table (IDT), the currently running thread, and the next thread selected to run. The kernel and the HAL use this information to perform architecture-specific and machine-specific actions. Portions of the PCR and PRCB structures are defined publicly in the Windows Device Driver Kit (DDK) header file Ntddk.h.

program A static sequence of instructions.

protected-mode A state during the boot process in which no virtual-to-physical translation occurs but a full 32 bits of memory becomes accessible. After the system is in protected-mode, Ntldr can access all of physical memory.

protocol driver A driver that implements a networking protocol such as TCP/IP, NetBEUI, or IPX/SPX.

prototype page table entries (prototype PTEs) A software structure the memory manager relies on to map potentially shared pages when a page can be shared between two processes. An array of prototype PTEs is created when a section object is first created.

quality of service (Qos) A networking technology that ensures that critical network applications receive highest priority.

quantum The length of time a thread is allowed to run before Windows interrupts the thread to find out whether another thread at the same priority level is waiting to run or whether the thread's priority needs to be reduced.

quantum unit A value that represents how long a thread can run until its quantum expires. This value is an integer value, not a length of time.

queue A method for threads to enqueue and dequeue notifications of the completion of I/O operations (called an I/O completion port in the Windows API).

queued spinlock A special type of spinlock that is used only by the kernel and not exported for executive components or device drivers. A queued spinlock scales better on multiprocessor systems than a standard spinlock does. *See also* spinlock.

quota charges In the Windows object manager, the record of how much the object manager subtracts from a process's allotted paged and/or nonpaged pool quota when a thread in the process opens a handle to the object.

RAID-5 volume A fault tolerant variant of a regular stripe volume. Fault tolerance is achieved by reserving the equivalent of one disk for storing parity for each stripe. Also called stripe volume with parity.

ready summary A bit mask (KiReadySummary) that Windows maintains to speed up the selection of which thread to run or preempt.

real-mode An operating mode in which no virtual-to-physical translation of memory addresses occurs, which means that programs that use the memory addresses interpret them as physical addresses and that only the first 1 MB of the computer's physical memory is accessible. Simple MS-DOS programs execute in a real-mode environment.

recoverability An advanced feature of NTFS that allows a system to recover from an unexpected system halt. If a system is halted unexpectedly, the metadata of a FAT volume can be left in an inconsistent state, leading to the corruption of large amounts of file and directory data. NTFS implements recoverability by logging changes to metadata in a transactional manner so that file system structures can be repaired to a consistent state with no loss of file or directory structure information. (File data can be lost, however.)

redo information Information included in the NTFS checkpoint record that explains how to reapply one suboperation of a fully logged (committed) transaction to the volume if a system failure occurs before the transaction is flushed from the cache.

reference count The object manager's record of how many object pointers it has dispensed to operating system processes. The object manager increments a reference count for an object each time it gives out a pointer to the object; when kernel-mode components finish using the pointer, they call the object manager to decrement the object's reference count.

reparse data User-defined data about the file or directory, such as its state or location, that can be read from the reparse point by the application that created the data, a file system filter driver, or the I/O manager.

reparse point An NTFS file or directory that has a block of data called reparse data associated with it.

Reparse tag The reparse tag allows the component responsible for interpreting the reparse point's reparse data to recognize the reparse point without having to check the reparse data.

resident attribute An attribute that is stored directly in the master file table. (If a file is small, all its attributes and their values [its data, for example] fit in the file record.)

resource arbitration A process by which the Plug and Play (PnP) manager optimally assigns hardware resources so that each device meets the requirements necessary for its operation. Because hardware devices can be added to the system after boot-time resource assignment, the PnP manager must also be able to reassign resources to accommodate the needs of dynamically added devices.

restricted token A token created from a primary or impersonation token using the CreateRestrictedToken function. The restricted token is a copy of the token it's derived from, with some possible modifications: Privileges can be removed from the token's privilege array. SIDs in the token can be marked as deny-only. SIDS in the token can be marked as restricted.

ring A privilege level defined in the x86 and x64 processor architectures to protect system code and data from being overwritten either inadvertently or maliciously by code of lesser privilege. Windows on the x86 and x64 platforms use privilege level 0 (or ring 0) for kernel mode and privilege level 3 (or ring 3) for user mode.

safe mode A boot configuration that consists of the minimal set of device drivers and services. By relying on only the drivers and services that are necessary for booting, Windows avoids loading third-party and other nonessential drivers that might crash.

SAM database A database (stored in the registry under HKLM\SAM) that contains the defined users and groups, along with their passwords and other attributes.

scatter/gather I/O A kind of high-performance I/O Windows supports, available via the Windows ReadFileScatter and WriteFileGather functions. These functions allow an application to issue a single read or write from more than one buffer in virtual memory to a contiguous area of a file on disk. To use scatter/gather I/O, the file must be opened for noncached I/O, the user buffers being used have to be page-aligned, and the I/Os must be asynchronous (overlapped).

section object An object that represents a block of memory that two or more processes can share. A section object can be mapped to the paging file or to another file on disk. The executive uses section objects to load executable images into memory, and the cache manager uses them to access data in a cached file. In the Windows subsystem, a section object is called a file-mapping object.

section object pointers Structure for each open file (represented by a file object) that is the key to maintaining data consistency for all types of file access as well as to providing caching for files. The section object pointers structure points to one or two control areas. One control area is used to map the file when accessed as a data file, and one is used to map the file when it is run as an executable image.

sector A hardware-addressable portion of a physical disk. A hard disk sector on an IBM-compatible PC is typically 512 bytes. Utilities that prepare hard disks for the definition of logical drives, including the MS-DOS Fdisk utility or the Windows Setup program, write a sector of data called a master boot record (MBR) to the first sector on a hard disk. The MBR includes a fixed amount of space that contains executable instructions (called boot code) and a partition table with four entries that define the locations of the primary partitions on the disk. *See also* boot code, partition table.

secure attention sequence (SAS) A keystroke combination that when entered notifies Winlogon of a user logon request.

Secure logon facility Requires that users can be uniquely identified and that they must be granted access to the computer only after they have been authenticated in some way.

Security Accounts Manager (SAM) service A set of subroutines responsible for managing the database that contains the usernames and groups defined on the local machine or for a domain (if the system is a domain controller). The SAM runs in the context of the LSASS process.

security auditing A way in which Windows detects and records important security-related events or any attempts to create, access, or delete system resources. Logon identifiers record the identities of all users, making it easier to trace anyone who performs an unauthorized action.

security descriptor The data structure that specifies who can perform what actions on an object. A security descriptor consists of attributes.

security identifier (SID) A means of uniquely identifying entities that perform actions in a system. A SID is a variable-length numeric value that consists of a SID structure revision number, a 48-bit identifier authority value, and a variable number of 32-bit subauthority or relative identifier (RID) values.

security quality of service (SQOS) The indicator specified by a client opening a server resource that specifies the maximum level of client impersonation the server is allowed.

security reference monitor (SRM) A component in the Windows executive (Ntoskrnl.exe) that enforces security policies on the local computer. It guards operating system resources, performing run-time object protection and auditing.

semaphore A counter that provides a resource gate by allowing some maximum number of threads to access the resources protected by the semaphore.

server processes User processes that are Windows services, such as the Event Log and Schedule services. Many add-on server applications, such as Microsoft SQL Server and Microsoft Exchange Server, also include components that run as Windows services.

session Consists of the processes and other system objects (such as the window station, desktops, and windows) that represent a single user's workstation logon session. Each session has a session-specific paged pool area used by the kernel-mode portion of the Windows subsystem (Win32k.sys) to allocate session-private GUI data structures. In addition, each session has its own copy of the Windows subsystem process (Csrss.exe) and logon process (Winlogon.exe).

session space A component of system space used to map information specific to a user session. The session working set list describes the parts of session space that are resident and in use.

shared cache map A structure that describes the state of a cached file, including its size and (for security reasons) its valid data length.

shared memory Memory visible to more than one process or that is present in more than one virtual address space.

signal state The state of a synchronization object.

simple volume A set of objects that represent sectors from a single partition that file system drivers manage as a single unit.

single sign-on The ability for logon information to be simultaneously authenticated on more than one system. For example, a user logging on to a Windows system might simultaneously be authenticated on a UNIX server. That user would then be able to access resources of the UNIX server from the machine running Windows without requiring additional authentication.

small memory dump A small memory dump (the default on Windows Professional, 64 KB in size) containing the stop code and parameters, the list of loaded device drivers, the data structures that describe the current process and thread (called the EPROCESS and ETHREAD), and the kernel stack for the thread that caused the crash.

spanned volume A single logical volume composed of a maximum of 32 free partitions on one or more disks. The Windows Disk Management Microsoft Management Console (MMC) snap-in combines the partitions into a spanned volume, which can then be formatted for any of the file systems that Windows supports.

sparse files Files, often large, that contain only a small amount of nonzero data relative to their size.

spinlock The locking mechanism the kernel uses to achieve multiprocessor mutual exclusion. The spinlock gets its name from the fact that the kernel (and thus, the processor) is held in limbo, "spinning," until it gets the lock. Spinlocks, like the data structures they protect, reside in global memory. *See also* queued spinlock.

stack frame Information, representing the activation of a procedure, that is pushed onto the stack when a procedure is invoked. A stack frame can have one or more exception handlers associated with it, each of which protects a particular block of code in the source program.

stream A sequence of bytes within a file.

striped volume Series of up to 32 partitions, one partition per disk, that combines into a single logical volume. Striped volumes are also known as RAID level 0 (RAID-0) volumes. A partition in a striped volume need not span an entire disk; the only restriction is that the partitions on each disk be the same size.

stop code An error code that classifies the type of error detected by a component that crashes the system.

structured exception handling A type of exception handling that allows applications to gain control when exceptions occur. The application can then fix the condition and return to the place the exception occurred, unwind the stack (thus terminating execution of the subroutine that raised the exception), or declare back to the system that the exception isn't recognized, and continue searching for an exception handler that might process the exception.

subsection A structure that describes the mapping information for each section of the file (read-only, read-write, copy-on-write, and so on).

subsystem dynamic-link libraries (DLLs) DLLs that translate a documented function into the appropriate undocumented Windows system service calls. This translation might or might not involve sending a message to the environment subsystem process that is serving the user application.

symbolic link A mechanism for referring to an object name indirectly.

symmetric encryption algorithm An algorithm that uses the same key to encrypt and decrypt data. Symmetric encryption algorithms are typically very fast, which makes them suitable for encrypting large amounts of data, such as file data. They do have a weakness, however; their security can be bypassed if their key is obtained.

symmetric multiprocessing (SMP) A multiprocessing operating system in which there is no master processor—the operating system as well as user threads can be scheduled to run on any processor. All the processors share just one memory space.

synchronization A thread's ability to synchronize its execution by waiting for an object to change from one state to another. A thread can synchronize with executive process, thread, file, event, semaphore, mutex, and timer objects. Section, port, access token, object directory, symbolic-link, profile, and key objects don't support synchronization.

synchronous I/O A model for I/O in which a device performs a data transfer and returns a status code when the I/O is complete. The program can then access the transferred data immediately. When used in their simplest form, the Windows ReadFile and WriteFile functions are executed synchronously. They complete an I/O operation before returning control to the caller.

system access-control list (SACL) Specifies which operations by which users should be logged in the security audit log.

system code write protection The setting of the read-only parts of the kernel as read-only pages.

system audit ACE An access-control entry (ACE) contained by a system access-control list (SACL) that, along with the system audit-object ACE, specifies which operations performed on the object by specific users or groups should be audited. System audit-object ACEs specify a GUID indicating the types of objects or subobjects that the ACE applies to and an optional GUID that controls propagation of the ACE to particular child object types. If an SACL is null, no auditing takes place on the object.

system cache Pages used to map files open in the system cache.

system page table entries (PTEs) Pool of system PTEs used to map system pages such as I/O space, kernel stacks, and memory descriptor lists.

system service dispatch table Table in which each entry contains a pointer to a system service rather than to an interrupt handling routine.

system services (or executive system services) Native functions in the Windows operating system that are callable from user mode. For example, NtCreateProcess is the internal system service the Windows CreateProcess function calls to create a new process.

system support processes User processes, such as the logon process and the Session Manager, that are not Windows services (that is, not started by the service controller).

system thread A kind of thread that runs only in kernel mode. System threads always reside in the system process (always process ID 2). These threads have all the attributes and contexts of regular user-mode threads (such as a hardware context, priority, and so on) but run only in kernel-mode executing code loaded in system-space code, whether that be in Ntoskrnl.exe or in any other loaded device driver. System threads don't have a user process address space and hence must allocate any dynamic storage from operating system memory heaps, such as paged or nonpaged pool.

system worker threads Threads created in the system process during system initialization that exist solely to perform work on behalf of other threads.

system working set The physical memory being used by the system cache, paged pool, pageable code in Ntoskrnl.exe, and pageable code in device drivers. *See also* process working set.

thread An entity within a process that Windows schedules for execution. A thread includes the contents of a set of volatile registers representing the state of the processor; two stacks, one for the thread to use while executing in kernel mode and one for executing in user mode; a private storage area for use by subsystems, run-time libraries, and DLLs; and a unique identifier called a thread ID (also internally called a client ID).

thread context A thread's volatile registers, the stacks, and the private storage area. Because this information is different for each machine architecture that Windows runs on, this structure is architecture-specific. In fact, the CONTEXT structure returned by the Windows GetThreadContext function is the only public data structure in the Windows API that is machine-dependent.

timer A mechanism that notifies a thread when a fixed period of time elapses.

transaction An I/O operation that alters file system data or changes the volume's directory structure. The separate disk updates that make up the transaction must be executed atomically; that is, once the transaction begins to execute, all its disk updates must be completed. If a system failure interrupts the transaction, the part that has been completed must be undone, or rolled back. The rollback operation returns the database to a previously known and consistent state, as if the transaction had never occurred.

transaction table A table that keeps track of transactions that have been started but that aren't yet committed. The suboperations of these active transactions must be removed from the disk during recovery.

transition A kind of invalid page table entry (PTE) in which the desired page is in memory on either the standby, modified, or modified-no-write list. The page is removed from the list and added to the working set.

translation look-aside buffer (TLB) A CPU cache of recently translated virtual page numbers.

trap A processor's mechanism for capturing an executing thread when an exception or an interrupt occurs, switching it from user mode into kernel mode, and transferring control to a fixed location in the operating system. In Windows, the processor transfers control to the kernel's trap handler.

trap frame A data structure in which the execution state of the interrupted thread is stored. This information allows the kernel to resume execution of the thread after handling the interrupt or the exception. The trap frame is usually a subset of a thread's complete context.

trap handler A module in the kernel that acts as a switchboard, fielding exceptions and interrupts detected by the processor and transferring control to code that handles the condition.

trusted facility management Requires support for separate account roles for administrative functions. For example, separate accounts are provided for administration (Administrators), user accounts charged with backing up the computer, and standard users.

trusted path functionality Prevents Trojan horse programs from being able to intercept users' names and passwords as they try to log in. The trusted path functionality in Windows comes in the form of its Ctrl+Alt+Delete logon-attention sequence, which cannot be intercepted by nonprivileged applications. This sequence of keystrokes, which is also known as the secure attention sequence (SAS), always pops up a logon dialog box, so would-be Trojan horses can easily be recognized. A Trojan horse presenting a fake logon dialog box will be bypassed when the SAS is entered.

type object An internal system object that contains information common to each instance of the object.

Unicode An international character set standard that defines unique 16-bit values for most of the world's known character sets.

update records The most common type of record NTFS writes to the log file. Each update record contains the information needed to redo an operation that updated the file system structure.

user mode The nonprivileged processor mode that applications run in. A limited set of interfaces is available in this mode, and the access to system data is limited. *See also* kernel mode.

VACB index array An array of pointers maintained by the cache manager to keep track of which views for a given file are mapped to the system cache.

view The portion of the section object required by a process. A section object can refer to files that are much larger than can fit in the address space of a process. (If the paging file backs a section object, sufficient space must exist in the paging file to contain it.) To access a very large section object, a process can map only a view of the section by calling the *MapViewOfFile* function and then specifying the range to map. Mapping views permits processes to conserve address space because only the views of the section object needed at the time must be mapped into memory. *See also* section object.

virtual address control blocks (VACBs) The data structures that the cache manager uses to manage the system address space into which it maps files.

virtual address descriptors (VADs) Data structures the memory manager maintains that keep track of which virtual addresses have been reserved in the process's address space. VADs are structured as a self-balancing binary tree to make lookups efficient.

virtual address space A set of virtual memory addresses that a process can use.

virtual block caching A method the Windows cache manager uses to keep track of which parts of which files are in the cache.

virtual cluster numbers (VCNs) VCNs number the clusters belonging to a particular file from 0 through m. VCNs aren't necessarily physically contiguous, but they can be mapped to any number of logical cluster numbers (LCNs) on a volume.

virtual device drivers (VDDs) Drivers used to emulate 16-bit MS-DOS applications. They trap what an MS-DOS application thinks are references to I/O ports and translate them into native Windows I/O functions. Because Windows is a fully protected operating system, user-mode MS-DOS applications can't access hardware directly and thus must go through a real kernel-mode device driver.

virtual memory manager Implements virtual memory, a memory management scheme that provides a large, private address space for each process that can exceed available physical memory.

volume One or more logical disk partitions that are treated as a single unit.

volume file A system file (filename $Volume) that contains the volume name, the version of NTFS for which the volume is formatted, and a bit that when set signifies that a disk corruption has occurred and must be repaired by the Chkdsk utility.

volume manager A term used to represent both FtDisk and DMIO because both FtDisk and DMIO support the same multipartition-volume types.

volume set A single logical volume composed of a maximum of 32 areas of free space on one or more disks.

wait block A data structure that represents a thread waiting on an object. Each thread that is in a wait state has a list of the wait blocks that represent the objects the thread is waiting on. Each dispatcher object has a list of the wait blocks that represent which threads are waiting on the object.

wait hint A value indicating how long a service should wait before informing the system a shutdown is complete.

WDM drivers Device drivers that adhere to the Windows Driver Model (WDM). WDM includes support for Windows power management, Plug and Play, and Windows Management Instrumentation (WMI). WDM is implemented on Windows 2000 and later, Windows 98, and Windows Millennium Edition, so WDM drivers are source compatible between these operating systems and in many cases are also binary compatible. There are three types of WDM drivers: bus drivers, function drivers, and filter drivers.

WDM Windows Management Instrumentation routines Enable device drivers to publish performance and configuration information and receive commands from the user-mode WMI service. Consumers of WMI information can be on the local machine or remote across the network.

Windows API The 32-bit interface to Windows 2000 and both the 32-bit and 64-bit programming interfaces to Windows XP and Windows Server 2003.

Windows services A mechanism to start processes at system startup time that provide services not tied to an interactive user. Services are similar to UNIX daemon processes and often implement the server side of client/server applications.

window station A window station contains desktops, and desktops contain windows. Only one window station can be visible on a console and receive user mouse and keyboard input. In a Terminal Services environment, one window station per session is visible, but services all run as part of the console session.

Windows drivers Device drivers that integrate with the Windows power manager and Plug and Play manager, when required. They include drivers for mass storage devices, protocol stacks, and network adapters.

Windows Management Instrumentation (WMI) manager A component of the executive that enables device drivers to publish performance and configuration information and receive commands from the user-mode WMI service. Consumers of WMI information can be on the local machine or remote across the network.

work item A unit of work placed on a queue dispatcher object when a device driver or an executive component requests a system worker thread's services by calling the executive functions ExQueueWorkItem or IoQueueWorkItem. Work items include a pointer to a routine and a parameter that the thread passes to the routine when it processes the work item. The routine is implemented by the device driver or executive component that requires passive-level execution.

working set A subset of virtual pages resident in physical memory. There are two kinds of working sets–process working sets and the system working set.

working set manager A routine that runs in the context of the balance set manager system thread to initiate automatic working set trimming to increase the amount of free memory available in the system. Although Windows attempts to keep memory available by writing modified pages to disk, when modified pages are being generated at a very high rate, more memory is required to meet memory demands. The working set manager is called when physical memory runs low (when MmAvailablePages is less than MmMinimumFreePages).

write-back A caching strategy the lazy write file system uses to improve performance. In write-back, the file system writes file modifications to the cache and flushes the contents of the cache to disk in an optimized way, usually as a background activity.

write-through An algorithm the FAT file system uses that causes disk modifications to be immediately written to the disk. Unlike the careful-write approach, the write-through technique doesn't require the file system to order its writes to prevent inconsistencies. *See also* careful write.

write throttling Prevents system performance from degrading because of a lack of memory when a file system or network server issues a large write operation.

zero page thread A kernel-mode system thread (thread 0 in the system process). A zero page thread zeroes out pages on the free list so that a cache of zero pages is available to satisfy future demand-zero page faults.

David Solomon

David Solomon, president of David Solomon Expert Seminars (www.solsem.com), has focused on explaining the internals of the Microsoft Windows NT operating system line since 1992. He has taught his world-renowned Windows Internals classes to thousands of developers and IT professionals worldwide. His clients include all the major software and hardware companies, including Microsoft. He was a recipient of the 1993 Microsoft Support Most Valuable Professional (MVP) award.

This is David's fourth book. His first book was *Windows NT for Open VMS Professionals* (Digital Press/Butterworth Heinemann, 1996). It explained Windows NT to VMS-knowledgeable programmers and system administrators. Prior to starting his own company, David worked for nine years as a project leader and developer in the VMS operating system development group at Digital Equipment Corporation. His second book, *Inside Windows NT, Second Edition* (Microsoft Press, 1998), covered the internals of Windows NT 4.0. His third book, *Inside Windows 2000, Third Edition* (Microsoft Press 2000) was co-authored by Mark Russinovich.

In addition to organizing and teaching seminars, David is a regular speaker at technical conferences such as Microsoft TechEd and Microsoft PDCs. He has also served as technical chair for several past Windows NT conferences. When he's not researching Windows, David enjoys sailing, reading, and watching Star Trek.

Mark Russinovich

Mark Russinovich is chief software architect and cofounder of Winternals Software (www.winternals.com), a company that specializes in advanced systems software for Microsoft Windows. He is a senior contributing editor for Windows IT Pro Magazine, where he contributes to the Windows Power Tools column. He and David Solomon deliver public and private seminars on Windows operating system internals and advanced troubleshooting to numerous companies and organizations, including regular deliveries to Microsoft employees. They also created a 12-hour self-paced Windows internals video tutorial, which Microsoft licensed for worldwide corporate use. Mark is also a frequent speaker at major industry conferences such as Microsoft Tech Ed, Microsoft IT Forum, Windows IT Pro Magazine's Connections, and MCP Magazine's TechMentor.

Mark has a B.S. from Carnegie Mellon University and an M.S. from Rensselaer Polytechnic Institute, both in computer engineering. In 1994, he earned a Ph.D. from Carnegie Mellon University, also in computer engineering. Before founding Winternals Software, he worked at Compuware NuMega Laboratories and IBM's Thomas J. Watson Research Center.