目录

前言

在1997年春季的那一学期，我们讲授了Linux 2.0操作系统这门课程。目的是鼓励学生阅读源代码。为了完成这项工作，我们按学期分配项目，一方面关注内核的变化，另一方面对版本的变化进行测试。我们也为学生写下课程笔记，主要是关于Linux任务切换和任务调度的一些主要特点。

在1998春季的那一学期我们沿着这一思路继续做下去，但是此时已转到Linux 2.1的开发版。我们的课程笔记变得越来越庞大。在1998年7月，我们与O'Reilly & Associates出版社联系，建议他们出版Linux内核全书。真正的工作在1998的秋天开始，持续了大约一年半的时间。我们读了数千行的代码，努力去理解其含义。在做了所有这些工作以后，可以说我们的努力是完全值得的。我们学到的很多东西也许你在本书中并不能全部找到，不过我们还是希望我们已经成功地在后面的内容中反映了这些信息。

有关这本书的读者

如果你想知道Linux如何工作，它的性能为什么会如此之高，你将会从本书中找到答案。阅读本书之后，你能找到阅读成千上万行代码的方法，区别出主要的数据结构和次要的语句片段的不同，简而言之，你将成为一名真正的Linux高手。

也许我们的工作有助于引导你深入Linux内核。我们讨论了在内核中使用的很多重要的数据结构、算法和编程技巧；在很多的例子中，我们逐行讨论有关代码片段。

当然，你应该手头有Linux源代码，为简洁起见，有些函数的描述可能不充分，你应该乐于花费一些功夫去解读它们。

另一方面，如果你想懂得现代操作系统中很多重要的设计思想，本书将给你有价值的洞察力。本书不仅仅是专门对系统管理员或编程人员的，它主要是针对那些想探究机器内部到底是怎样工作的人们的！与任何好向导一样，我们试图透过表面的一些特点看其内部。我们提供背景材料，例如在谈及主要特点时，同时说明其使用理由和历史。

原材料的组织

开始写这本书时，我们面临重大的决定：是应该涉及特定的硬件平台，还是跳过与硬件相关的细节，而集中于与硬件无关的纯粹的内核部分？

有关Linux内核的其他书选择后一种方式，因为下述理由，我们决定采用前一种方式：

- 高效率的内核充分利用硬件可利用的特点，诸如寻址技术、高速缓存(caches)、处理器异常（exception）、专用指令、处理器控制寄存器等等。如果我们想使你相信，内核在执行一个特殊的任务时确实工作得相当好，那我们必须首先告诉你内核工作在一个什么样的硬件平台上。
- 即使Unix内核大部分源代码是独立于处理器的，并且用C语言编写，但也有少数重要的部分是用汇编语言编写的。为了充分理解内核，就需要学习一些涉及相关硬件的汇编语言片段。

当涉及硬件特点时，我们的策略将非常简单：完全的硬件工作也需要软件的支持，当详细描述与此相关的内容时，只是简单地勾画出硬件的特点。实际上，我们感兴趣的是内核的设计而不是计算机的体系结构。

下一步就要选择所描述的计算机系统：尽管Linux目前已运行在很多个人计算机（PC）和工作站上，但我们决定把主要精力放在非常流行的，且便宜的IBM PC兼容机上，其中微处理器是Intel 80x86及PC中所支持的一些芯片。在以后的章节中，术语“Intel 80x86微处理器”将表示Intel 80386、80486、Pentium、Pentium Pro、

Pentium II及Pentium III微处理器及兼容模式。在少数情况下，对于特殊的模式会给出明确的说明。

在研究Linux的组成时，可以遵循不止一种顺序。我们努力遵循自底向上的顺序：从硬件相关的主题开始，以硬件完全无关的内容结束。实际上，在本书的第一个部分，我们给出Intel 80x86微处理器的很多参考书，而其他部分相对来说与硬件无关。在第十一章和第十三章中，给出两种很重要的例外情况。实际上，存储器管理，进程管理和文件系统这几部分相互渗透，因此，所谓自底向上的顺序并不像看起来那样简单；少数向前的引用（即涉及还需说明的话题）是不可避免的。

每一章的开始是本章内容的理论概述，然后按自底向上的顺序组织材料。我们以本章所需要的数据结构开始来支持所描述的功能。然后，我们通常从最低级函数移到高级函数，最后以说明系统调用如何在用户程序中使用而结束。

描述级别

支持各种体系结构的Linux源代码包含在大约4500个C语言和汇编语言文件中，这些文件存放在大约270个子目录中。源代码大约由200万行组成，占58MB以上的磁盘空间。当然，这本书只能覆盖源代码非常少的一部分。为了计算出Linux源代码有多么大，考虑一下你所读的书的全部源代码只占不到2MB的磁盘空间。因此，即使不对源码进行解释，只列出所有的代码，25本书也写不完（注1）！

因此，我们必须对要阐述那些内容做出选择，我们的决策大致如下：

- 我们相当全面地描述进程管理和存储器管理。
- 我们涵盖了虚拟文件系统和Ext2文件系统的主要内容，不过，并没有对所提及到的很多函数进行详尽的描述；我们没有讨论Linux所支持的其他文件系统。
- 我们描述了设备驱动程序，这在内核中占相当大的比例；也描述了所涉及的内

注1：不过，与其他商业化的大型操作系统相比，Linux属于很小的操作系统。例如，据报道，微软Windows 2000的源代码超过3000万行。与某些流行的应用程序相比，Linux也是比较小的；Netscape Communicator 5浏览器也大约有1700万行代码。

核接口，但是没有分析任何具体的驱动程序，即使是终端驱动程序也没有进行分析。

- 我们也没有涉及网络，因为这部分内容本身值得写一本书。

在很多例子中，为了易读，重写了原始代码，但以低效的方式重写。用优化的C语言和汇编代码混合在一起重写了与时间有关的程序片段。再次声明，我们的目的是为研究Linux源码的人提供一些帮助。

在讨论内核代码时，我们描述了Unix编程人员所熟悉的一些基础知识（共享和映射内存、信号、管道、符号链），也许他们听说过这些内容，但可能还想进一步了解。

本书的概述

为了对全书有一个大体了解，在第一章中对Unix内核内部结构给出一般性描述，说明Linux如何与其他著名的Unix系统进行竞争。

任何Unix内核的核心都是存储器管理。第二章说明Intel 80x86处理器所包含的特殊电路如何在内存中对数据进行寻址及Linux如何充分利用它们。

进程是Linux所提供的基本抽象，这在第三章中进行介绍。在这一章，我们也说明了每个进程是如何在无特权的用户模式下运行，又如何在有特权的内核模式下运行。用户模式与内核模式之间的转换只能通过已建立的中断和异常处理硬件机制实现，这些内容将在第四章中介绍。定时中断是Linux所关注的一种重要的中断类型，进一步的内容将在第五章中介绍。

下面我们再一次集中讨论内存。第六章描述的是用来处理系统中最宝贵的资源（当然除了处理器）所需要的复杂技术。必须保证把这种资源既提供给Linux内核也提供给用户应用程序。第七章讲述内核如何处理应用程序对内存发出的“无限(greedy)”请求。

第八章说明在用户模式下运行的进程如何利用内核所提供的服务；而第九章描述进程如何给其他进程发送同步信号。第十章说明在系统中Linux如何轮流执行每一个处于就绪状态的进程，以使所有的进程都能顺利执行完。内核也需要同步机制，这将在第十一章进行讨论。内核同步既适用于单处理机，也适用于多处理机系统。

现在我们准备进入另一个实质性话题，即Linux如何实现文件系统。很多章节涉及到这个话题。第十二章介绍了支持很多种不同文件系统的一般层次。某些Linux文件比较特殊，因为它们能提供直接到达硬件设备的陷阱门；第十三章进一步考察了这些特殊的文件和相应的硬件设备驱动程序。另一个值得考虑的问题是磁盘访问时间；第十四章说明灵活地利用RAM不仅可以减少磁盘的访问时间，还能极大地提高系统的性能。在前几章内容的基础上，我们将在第十五章讨论用户如何访问正规文件。在第十六章我们将结束对Linux存储器管理的讨论，说明Linux使用交换技术总是能确保有足够的内存可供使用。第十七章是讨论文件系统的最后一章，阐述了Linux最常用的文件系统，即所谓的Ext2。

最后两章我们结束对Linux内核的详细游览：第十八章介绍用户态进程可以使用的通信机制（不包括信号）；第十九章说明如何开始执行用户应用程序。

最后是必不可少的附录：附录一讲述Linux如何启动；而附录二描述怎样动态地重新配置正在运行的内核，可以根据需要增加或删除有关功能；附录三是Linux源代码的目录列表。源代码索引包括了在本书中引用的所有Linux符号，你将在这里找到定义每个符号的Linux文件名，以及对这个符号进行解释的页号。你会发现它非常方便。

背景知识

除了一些C语言编程技巧和汇编语言的知识外，理解这些内容没有其他的要求。

排版约定

下面是本书在英文字体上的两个约定：

等宽字体（`constant width`）

用来说明代码文件的内容或命令输出的内容，也表示源代码中出现的关键字。

斜体（*italic*）

用来说明文件名、目录名、程序名、指令名、命令行选项、URL以及要强调的新术语。

建议与评论

本书的内容都经过测试，尽管我们做了最大的努力，但错误和疏忽仍然是在所难免的。如果你发现有什么错误，或者是对将来的版本有什么建议，请通过下面的地址告诉我们：

美国：

O'Reilly & Associates,Inc.
101 Morris Street
Sebastopol, CA 95472

中国：

100080 北京市海淀区知春路 49 号希格玛公寓 B 座 809 室
奥莱理软件（北京）有限公司

你也可以发电子信息。如果要把电子邮件放到我们的邮件列表中，或索要书目，请发邮件到：

info@oreilly.com

如果要询问技术问题或对本书进行评论，请发邮件到：

bookquestions@oreilly.com

我们也有本书的网站，网站将列出读者的评论、勘误表以及进一步的出版计划。你可以访问的这个主页在：

http://www.oreilly.com/catalog/linuxkernel/

我们还有另外的网站，你能在这个网站找到作者所写的关于Linux 2.4的新特点。我们希望在本书的下一版中能用到这些资料。这个网站在：

http://www.oreilly.com/catalog/linuxkernel/updates/

有关这本书更多的资料请访问 O'Reilly 网站：

http://www.oreilly.com
http://www.oreilly.com.cn

致谢

如果没有罗马大学Tor Vergata分校工程学院很多学生的尽力帮助，这本书不可能完成，他们不但上了这门课，还努力解读了Linux内核的讲稿。他们不懈的努力紧紧抓住了源码真正的含义，也使我们对讲稿不断改进，并改正了很多错误。

Andy Oram是O'Reilly & Associates出版社我们这本书优秀的编辑，非常值得信任。他是O'Reilly出版社第一位对此项目给予信任的人，花费了很多时间和精力阅读我们的初稿。他提出的很多建议使本书的可读性更强，他还写出了不少出色的介绍性段落。

另外，对O'Reilly全体职员表示感谢，尤其是技术插图画家Rob Romano及提供工具支持的Lenny Muellner。

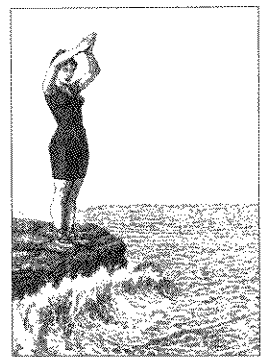
还有一些著名的技术审校，他们非常认真地阅读了这本书的内容（名字按字母顺序）：Alan Cox（译注1），Michael Kerrisk, Paul Kinzelman, Raph Levien, and Rik van Riel。他们的审校帮助我们去掉了很多错误和不准确的地方，使得本书更具有说服力。

— Daniel P. Bovet
Marco Cesati
2000年9月

译注1：Alan Cox是除Linus Torvalds之外对Linux贡献最大的开发者。也是目前Linux的日常维护者。

第一章

绪论



Linux是类Unix(Unix-like)操作系统大家族中的一名成员。从90年代末开始，Linux这位相对较新的成员突然变得非常流行，并且跻身于那些有名的商用Unix操作系统之列，如：AT&T公司开发出的SVR4（System V Release 4）（现在由Novell公司拥有），加利福尼亚大学伯克利分校（4.4BSD）发布的4.4 BSD，DEC公司（现在属于康柏）发布的Digital Unix，IBM公司发布的AIX，惠普公司发布的HP-UX，Sun公司发布的Solaris。

1991年，Linus Torvalds开发出最初的Linux，这个操作系统适用于基于Intel 80386微处理器的IBM PC兼容机。现在，Linus依然不遗余力地改进Linux，使它保持与各种硬件平台发展的同步更新，协调世界各地上百名开发者的开发工作。几年来，开发者已经使Linux可以在其他平台上运行，包括Alpha，SPARC，摩托罗拉的MC680x0，PowerPC及IBM System/390。

Linux一个最吸引人的优点在于，它不是商业化的操作系统，它遵循GNU公共许可证（GPL，注1），源代码全部开放，就像我们在本书中所介绍的那样，任何人都可

注1：GNU项目（*http://www.gnu.org*）由自由软件基金会所倡导，其目的是实现一个完整的操作系统，供大家自由使用。GNU C编译器的实用性就是这种Linux项目的成功范例。

以获得源码并研究它。只要下载源代码（官方的站点是*http://www.kernel.org/*），或者在Linux光盘上找到源代码，你就可以由表及里地探究这个最成功而又最现代的操作系统。实际上，这本书假定你手上有源代码，而且你能把我们介绍的方法应用到自己的探索中。

从技术角度来说，尽管Linux是一个真正的Unix内核，但它不是完整的Unix操作系统，因为以下几方面的应用程序没有全部包括在Linux中：如文件系统实用程序，窗口系统，图形化桌面，系统管理员命令，文本编辑程序，编译程序等等。然而，因为以上大部分应用程序都可在GNU许可证下免费获得，因此，可以把它们安装在任何一个Linux支持的文件系统中。

因为Linux是一个内核，因此很多Linux用户从CD-ROM获得一些商业发布版本，得到包含在标准Unix系统中的代码。另外，源代码也可以从几个不同的FTP网站获得。Linux源代码通常在*/usr/src/linux*目录下。在本书的其余部分，所有文件的目录都隐含指这一目录。

Linux与其他类Unix内核的比较

市场上的各种类Unix系统在很多重要的方面有所不同，其中有些已经有很长历史，并且显得有点过时。所有商业版本都是SVR4或4.4BSD的变体。所有版本都趋向于遵循通用的标准，如：IEEE的POSIX（Portable Operating Systems based on Unix）和X/Open的CAE（Common Applications Environment）。

现有标准仅仅指定了应用程序编程接口（API）——也就是说，指定了用户程序能够运行的一个已定义好的环境。因此，这些标准并没有对内核内部的设计进行任何限制（注2）。

为了定义通用的用户界面，类Unix内核通常共享基本的设计思想和特征。在这一点上，Linux和其他的类Unix操作系统是一样的。因此，不管你是在这本书中所读到的，还是你在Linux核中所看到的，都有助于你理解其他的Unix变体。

Linux内核2.2版试图与IEEE POSIX标准兼容。当然，这意味着在Linux系统下，

注2：实际上，一些非Unix操作系统（如Windows NT）也兼容POSIX。

花很少的力气，甚至不用对源码打补丁，就可以编译和运行目前的大多数Unix程序。此外，Linux包括了现代Unix操作系统的全部特点，诸如虚存，虚拟文件系统，轻量级进程，可靠的信号，SVR4进程间通信，支持对称多处理器（Symmetric Multiprocessor，SMP）系统等。

Linux内核本身并不是十分创新的。当Linus Torvalds写第一个内核的时候，他参考了Unix方面一些经典的书，比如Maurice Bach的《The Design of the Unix Operating System》（Prentice Hall, 1986），实际上，Linux仍然对Bach的书（即System V）中所描述的基准有些偏爱。但是，Linux没有拘泥于任何一个变体，相反，它尝试采用几种不同的Unix内核的好的特征和设计选择。

关于Linux如何与一些有名的商用Unix内核竞争，这里给出一个评价：

- Linux内核是单块结构。它是一个庞大、复杂的DIY（do-it-yourself）程序，由几个逻辑上不同的部分组成。在这一点上，它是相当传统的，大多数商用Unix变体也是单块结构。一个著名的例外是卡耐基－梅隆大学的Mach 3.0，它采用微内核（microkernel）的方法。
- 传统的Unix内核以静态的方式编译和连接。而大部分现代操作系统内核可以动态地装载和卸载部分内核代码（典型地，如设备驱动程序），通常把这种结构叫做模块（module）。Linux对模块的支持是很好的，因为它能动态地按需装载或卸载模块。在主流的商用Unix变体中，仅SVR4.2内核有类似的特点。
- 内核线程（kernel thread）。一些现代Unix内核，如Solaris 2.x和SVR4.2/MP，被当作一组内核线程来组织。内核线程是一个能被独立地调度的可执行上下文，也许它与用户程序相关，也许仅仅执行一些内核函数。线程之间的上下文切换比普通进程之间的上下文切换花费的代价要少得多，因为前者通常在一个共同的地址空间运行。Linux内核线程以一种十分受限制的方式来周期性地执行几个内核函数。因为Linux内核线程不能执行用户程序，因此，它们并不代表基本的可执行上下文的抽象（这就是下面要讨论的议题）。
- 支持多线程应用程序。大多数现代操作系统在某种程度上都支持多线程应用程序，即用共享应用程序的大部分数据结构的相对独立的执行流来设计用户程序。一个多线程用户程序由很多轻量级进程（lightweight process，LWP）或进程组成，这些进程有共同的地址空间、共同的物理内存页、共同的打开的文件等等。Linux定义了自己的轻量级进程版本，这与SVR4、Solaris等其他系

统上所使用的类型有所不同。当LWP的所有商用Unix变体都基于内核线程时，Linux却把轻量级线程当作基本的可执行上下文，通过非标准的系统调用clone()来处理它们。

- Linux是非抢占式（nonpreemptive）内核。这就意味着，不能随意地交错执行处于特权模式下的流。内核中的几部分代码假定它们能够运行和修改一些数据结构，无须担心被中断，也无须担心让另一个线程改变这些数据结构。通常，完全抢占式（preemptive）内核总是与专用实时操作系统有关。当前，在传统通用Unix系统中，仅仅Solaris 2.x和Mach3.0是完全的抢占式内核。SVR4.2/MP引入了一些固定的抢占点（fixed preemption points），作为得到有限的抢占能力的方法。

- 支持多处理器。几种Unix内核变体都利用了多处理器系统。Linux 2.2对对称多处理器（SMP）提供了一种更先进的支持，也就是说，系统不仅可以使用多处理器，而且任何一个处理器可以处理任何一个任务，它们之间没有任何区别。不过，Linux2.2没有充分地利用SMP。几个内核活动本该并发执行，如处理文件系统和连接网络，但现在必须顺序地执行。

- 文件系统。Linux的标准文件系统缺乏一些类似日志的先进特性。不过，更高级的文件系统已经可以在Linux上使用了，尽管它们没有被包含在Linux源代码中；其中，IBM AIX的日志文件系统（Journaling File System，JFS）和SGI公司Irix系统上的XFS文件系统就属于这类文件。有了强大的面向对象的虚拟文件系统技术（出自于Solaris和SVR4），把外部文件系统移植到Linux就变得相对容易了。

- 流（STREAMS）。尽管在现在大部分的Unix内核内包含了SRV4引入的I/O流子系统，并且已变成编写设备驱动程序、终端驱动程序及网络协议的首选接口，但是Linux并没有与此类似的子系统。

上述评价也许并未概括Linux的全部特点，但这几个特点就足以使Linux成为非常独特的操作系统。商用Unix内核常常引入新的特点来赢得更大的市场份额，但是，这些特点不一定有用，也不见得稳定或者效率很高。其实，现代的Unix内核倾向于不断膨胀。相反，Linux不受制于市场强加的限制和条件，所以，它能根据其设计者（主要是Linus Torvalds）的想法自由地发展。特别说明的是，与商用竞争者相比，Linux具有以下优势：

Linux 是免费的。

在硬件之外，你无须任何花费就能安装一套完整的 Linux 系统。

Linux 的所有部分可以充分地定制。

有了通用公共许可证（GPL），你就可以自由地阅读、修改内核和所有系统程序的的源代码（注 3）。

Linux 可以运行在低档、便宜的硬件平台。

你甚至可以用一个 4M 内存的旧 Intel 80386 系统构架网络服务器。

Linux 是强大的。

因为 Linux 充分挖掘了硬件部分的特点，因此，Linux 系统非常快。Linux 的主要目标是效率。事实上，商用系统的许多设计选择，如流 I/O 子系统，由于它们性能差而被 Linus 舍弃。

Linux 对源代码质量有一个高标准。

Linux 系统通常非常稳定，有非常低的故障率和系统维护时间。

Linux 内核非常小，而且紧凑。

我们甚至可以把一个内核映像和完整的根文件系统、包括所有基本的系统程序，放在 1.4M 的软盘上！就我们所知，没有一个商用 Unix 变体能从一张软盘上启动。

Linux 与很多常见的操作系统高度兼容。

Linux 可以让你直接安装以下文件系统的所有版本：MS-DOS 和 MS Windows，SVR4，OS/2，Mac OS，Solaris，SunOS，NeXTSTEP，还有很多 BSD 变体等等。另外，Linux 能和许多网络层一起工作，如以太网、光纤分布式数据接口（FDDI），高性能的并行接口（HIPPI），IBM 令牌环，AT&T 公司 WaveLAN，DEC 公司的 RoamAbout DS 等等。只要使用适当的库函数，Linux 系统甚至能直接运行为其他操作系统所编写的程序。例如，Linux 能执行为以下操作系统所编写的应用程序：MS-DOS，MS Windows，SVR3 及 R4，4.4BSD，SCO Unix，XENIX，以及其他 Intel 80x86 平台上运行的操作系统。

Linux 有很好的支持。

不管你信不信，比起任何有版权的操作系统，Linux 得到补丁与升级要容易得

注 3：一些商业公司已经开始在 Linux 下支持他们的产品，但其大部分产品并不是在 GNU 许可证下发布的。因此，不允许你读或修改他们的源代码。

多！只要你把问题发给一些新闻组或邮件列表，几个小时内就会得到回应。此外，当新的硬件产品投放市场以后，其Linux驱动程序在几周内就可得到。与此相反，硬件厂商仅仅给少数商业操作系统发布设备驱动程序，通常只有微软一家。因此，所有商用Unix变体只能运行在有限的硬件上。

随着安装Linux的机器（估计）已经超过了1200万台并且还在不断增长，那些已习惯于其他操作系统下标准特征的用户开始期望Linux也具有相同的特征。同样的情况是，Linux开发者的需求也在不断增加。值得庆幸的是，在Linus多年的密切指导下，Linux始终在不断发展以满足如此众多的需求。

硬件的依赖性

Linux试图在硬件无关的代码与硬件相关的代码之间维持一个清晰的划分。为了做到这点，在arch和include目录下都包含了9个子目录，也就是对应了9个所支持的硬件平台。这些平台的标准名字如下：

arm

Acorn个人计算机

alpha

康柏Alpha工作站

i386

IBM公司的个人计算机，基于Intel 80x86或与Intel 80x86兼容的微处理器

m68k

基于摩托罗拉MC680x0微处理器的个人计算机

mips

基于SGI公司MIPS微处理器的工作站。

ppc

基于Motorola-IBM的PowerPC微处理器的工作站

sparc

基于Sun公司SPARC微处理器的工作站。

sparc64

基于Sun公司的64位Ultra SPARC微处理器的工作站

s390

IBM的System/390大型机

Linux版本

Linux通过简单的编号来区别内核的稳定版和开发版。每个版本号由三位数字组成，由圆点分割。前两位数表示版本号、第三位数表示发布号。

如图1-1所示，如果第二位数字为偶数，表示稳定的内核；否则，表示开发中的内核。当写这本书时，Linux内核的当前稳定版为2.2.14，而当前开发版为2.3.51。1999年1月首次发布了内核2.2版，本书就是根据此版本而写，2.2版与2.0版内核有较大的不同，尤其是在内存管理方面。对2.3开发版的工作开始于1999年5月。

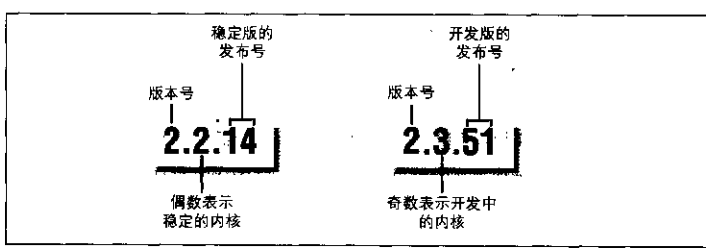


图1-1 Linux版本编号方式

一个稳定版本的新的发布主要用来纠用户报告的错误，但实现内核的主要算法和数据结构基本保持不变。

另一方面，开发版的不同版本之间可能有非常显著的差异。内核开发者可以自由地用可能导致内核有很大变化的不同的解决方案进行实验。用开发版运行应用程序的用户，当把内核升级到新版时，也许会经历一些不那么令人愉快的意外。这本书集中讨论我们可得到的最新的稳定版，因为实验中的内核所具有的所有特点，我们都既没办法知道哪个最终能被接受，也无法知道它们最终看上去会是什么样子。

当执笔写这本书时，Linux 2.4还没正式发布。通过看Linux 2.3.99-pre8预发布版，我们试图预测2.4版未来的特点和相对于2.2版内核的主要变化。Linux 2.4继承了Linux 2.2的好多东西、很多的概念、设计选择、算法和数据结构仍然是相同的。正因为如此，我们将就刚讨论的专题，通过粗略地勾画出Linux 2.4与Linux 2.2之间有什么不同，来结束每一章。你将注意到，新Linux之光在闪烁、对大公司，甚至整个业界，Linux之吸引力日趋突出。

操作系统的基本概念

任何计算机系统都包含一个基本的程序集合，称为操作系统。在这个集合里，最重要的程序被称为内核（kernel）。当操作系统启动时，内核被装入到RAM中，内核中包含了系统运行所必不可少的很多过程（procedure）。其他程序是一些不太重要的实用程序，尽管这些程序为用户提供了与计算机广泛的交互性（以及那些工作，用户之所以买计算机就是为了完成这些工作），但系统根本的样子和能力还是由内核决定。因此，在本书中，我们把注意力集中在内核。因此，我们将经常使用术语“操作系统”作为“内核”的同义词。

操作系统必须完成两个主要的目标：

- 与硬件部分相互作用，为所有包含在硬件平台上的低层可编程部件提供服务。
- 为运行在计算机系统上的应用程序（即所谓用户程序）提供一个执行环境。

一些操作系统允许所有的用户程序直接与硬件部分进行交互（典型的例子是MS-DOS）。与此相反，类Unix操作系统在用户应用程序前把与计算机物理组织相关的所有低层细节都隐藏起来。当程序想使用硬件资源时，必须向操作系统发出一个请求。内核对这个请求进行评估，如果允许使用这个资源，那么，内核代表应用程序与相关的硬件部分进行交互。

为了实施这种机制，现代操作系统依靠特殊的硬件特性来禁止用户程序直接与低层硬件部分打交道，或者直接访问任意的物理地址。硬件为CPU引入了至少两种不同的执行模式：用户程序的非特权模式和内核的特权模式。Unix把它们分别称为用户态（User Mode）和内核态（Kernel Mode）。

在本章的剩余部分，我们将介绍一些基本概念，在过去的20年里，这些概念推动了Unix、Linux和其他操作系统的设计。作为Linux的用户，你也许已熟悉了这些概念，但下面将对这些概念做进一步的研究，以解释它们寄予操作系统内核一些什么需求。这些概括的考虑涉及到类Unix系统，也涉及到Linux。希望这本书的其他章节能帮助你理解Linux内核。

多用户系统

一个多用户系统就是一台能并发和独立地执行分别属于两个或多个用户的若干应用程序的计算机。“并发”意味着几个应用程序能同时处于活动状态并竞争各种资源，如CPU、内存、硬盘等等。“独立”意味着每个应用程序能执行自己的任务，而无须考虑其他用户的应用程序在干些什么。当然，从一个应用程序切换到另一个会使每个应用程序的速度有所减慢，从而影响到达用户的响应时间。而现代操作系统内核引入许多复杂的特性（我们将在本书中考察这些特性）来减少每个程序执行时的延迟时间，给用户提供了尽可能快的响应时间。

多用户操作系统必须包含以下几个特点：

- 认证机制，核实用户身份。
- 一个保护机制，防止有错误的用户程序防碍其他应用程序在系统中运行。
- 一个保护机制，防止有恶意的用户程序干涉或窥视其他用户的活动。
- 计账机制，限制分配给每个用户的资源数。

为了确保能实现这些安全保护机制，操作系统必须利用与CPU特权模式相关的硬件保护机制，否则，用户程序将直接访问系统电路，克服这些强加于它的限制。Unix是实施系统资源硬件保护的多用户系统。

用户和组

在多用户系统中，每个用户在机器上有私用空间，典型地，他拥有一些磁盘空间的限额来存储文件，接受私人邮件信息等等。操作系统必须保证用户空间的私有部分仅仅对其拥有者是可见的。特别是必须能保证，没有用户能够开发一个用于侵犯其他用户私有空间的系统应用程序。

所有的用户由一个唯一的数字来标识，这个数字叫用户标识符（User ID，UID）。通常一个计算机系统只能由有限的人使用。当用户开始一个工作会话时，操作系统会要求输入一个名和口令，如果用户输入无效，则系统拒绝访问。因为口令是不公开的，因此用户的保密性得到保证。

为了和其他用户有选择地共享资料，每个用户是一个或多个组的一名成员，组由唯一的组标识符（Group ID，GID）标识。每个文件也恰好与一个组相对应。例如，可以设置这样的访问权限，拥有文件的用户具有对文件的读写权限，同组用户仅有只读权限，而系统中的其他用户没有对文件的任何访问权限。

任何类Unix操作系统都有一个特殊的用户，叫做root，超级用户（superuser），或管理员（supervisor）。系统管理员必须以root的身份登录，以便处理用户帐号，完成诸如系统备份、程序升级等维护任务。root用户几乎无所不能，因为操作系统对她不使用常用的保护机制。尤其是，root用户能访问系统中的每一个文件，能干涉每一个正在执行的用户程序的活动。

进程

所有的操作系统都使用一种基本的抽象：进程（process）。一个进程可以被定义为："执行程序的一个实例"，或者一个运行程序的"执行上下文"。在传统的操作系统中，一个进程在地址空间（address space）中执行一个单独的指令序列。地址空间是允许进程引用的内存地址集合。现代操作系统允许具有多个执行流的进程，也就是说，在相同的地址空间可执行多个指令序列。

多用户系统必须实施一种执行环境，在这种环境里，几个进程能并发地活动，并能竞争系统资源（主要是CPU）。允许进程并发活动的系统被称为多道程序系统或多处理技术系统（注4）。区分程序和进程是非常重要的：几个进程能并发地执行同一程序，同时，同一个进程能顺序地执行几个程序。

在单处理器系统上，只有一个进程能占用CPU，因此，在某一时刻，只能有一个执行流。一般来说，CPU的个数总是有限的，因而，只有几个进程能同时执行。选择哪个进程执行留给操作系统的一个被称为调度程序（scheduler）的部分决定。一些

注4：一些多处理技术操作系统不是多用户的，其中一个例子就是微软公司的Windows 98。

操作系统只允许有非抢占式进程，这就意味着，只有当进程自愿放弃CPU时，调度程序才被调用。但是，多用户系统中的进程调度必须是抢占式的。操作系统记录下每个进程占有的CPU时间，并周期性地激活调度程序。

Unix是抢占式、多重处理的操作系统。确实，在所有Unix系统中进程抽象确实是非常基本的概念。即使没有用户登录，没有程序运行，也有几个系统进程在监视外围设备。特别地，几个进程在监听系统终端等待用户登录。当用户输入一个登录名，监听进程就运行一个程序来验证用户的口令。如果用户身份得到证实，那么监听进程就创建另一个进程来执行shell，在那里可以输入命令。当一个图形化界面被激活时，有一个进程开始运行窗口管理器，界面上的每个窗口通常都由一个单独的进程来执行。当用户创建了一个图形化shell，一个进程运行图形化窗口，第二个进程运行用户可以输入命令的shell。对每一个用户命令，shell进程创建另一个进程来执行相应的程序。

类Unix操作系统采用进程/内核模式。每个进程都自以为它是系统中唯一的进程，可以独占操作系统所提供的服务。只要进程发出系统调用（即对内核提出请求），硬件就会把特权模式由用户态变成内核态，进程开始执行一个内核过程，该过程的执行目的被局限在一个非常小的范围内。这样，操作系统在进程执行的上下文中运行来满足进程的请求。一旦这个请求完全得到满足，内核过程将迫使硬件返回到用户态，进程将从系统调用的下一条指令开始继续执行。

内核体系结构

如前所述，大部分Unix内核的体系结构是单模块，每一个内核层都被集成到整个内核中，并代表当前进程在内核态下运行。相反，微内核操作系统只需要内核的一个很小的函数集，通常包括几个同步原语、一个简单的调度程序和进程间通信机制。运行在微内核上的几个系统进程运行其他的操作系统级函数，如内存分配程序、设备驱动程序、系统调用处理程序等等。

尽管关于操作系统的学术研究都是面向微内核的，但这样的操作系统一般比单模块的效率低，因为操作系统不同层次之间显式的消息传递要花费一定的代价。不过，从理论上来说，微内核操作系统比单模块有一定的理论优势。微内核操作系统迫使系统程序员采用模块化的方法，因为任何操作系统层都是一个相对独立的程序，这个程序必须通过定义明确的清晰的软件接口与其他层交互。此外，已有的微内核操

作系统可以很容易地移植到其他的体系结构上，因为所有与硬件相关的部分都被封装进微内核代码中。最后，微内核操作系统比单模块更加充分地利用了RAM，因为暂且不需要执行的系统进程可以被调出或撤消。

模块是内核的一个特点，在不影响性能的情况下，它能有效地实现许多微内核理论上的优点。模块是一个目标文件，其代码可以在运行时链接到内核或从内核中取下。这种目标代码通常由一组函数组成，用来实现一种文件系统、一个驱动程序或其他内核上层的功能。与微内核操作系统的外层不同，模块不是作为一个特殊的进程执行的。相反，与任何其他静态链接的内核函数一样，它在内核态代表当前进程执行。

使用模块的主要优点包括：

模块化方法

因为可以在运行时链接或卸下模块，因此，系统程序设计员必须引入定义明确的软件接口来访问由模块处理的数据结构。这就使得开发新的模块变得容易。

平台无关性

即使模块依赖于某些特殊的硬件特点，但它不依赖于某个固定的硬件平台。例如，符合SCSI标准的磁盘驱动程序模块，在IBM兼容PC与康柏的Alpha机上都能很好地工作。

节省内存的使用

当需要模块的功能时，把它链接到正在运行的内核中，否则，卸下该模块。这种机制对用户是透明的，因为链接和卸下是由内核自动完成的。

没有性能损失

模块的目标代码一旦被链接到内核，其作用与静态链接的内核目标代码完全等价。因此，当调用模块的函数时（注5），无须显式地消息传递。

Unix 文件系统概述

Unix操作系统的设计集中在它的文件系统上，后者有几个有趣的特点。因为在后面的章节中，将会反复提到这些特点，所以我们先来回顾几个最重要的。

注5：当模块被链接或被卸下时，性能稍有影响。但是，在微内核操作系统中，系统进程的创建和删除也是这样的。

文件

Unix 文件是以一列字节组成的信息载体（container），内核不解释文件的内容。很多编程的库函数实现了更高级的抽象，例如，由域构成的记录以及基于关键字编址的记录。然而，这些库中的程序必须依靠内核提供的系统调用。从用户观点来看，如图 1-2 所示，文件在一个树型的命名空间中进行组织。

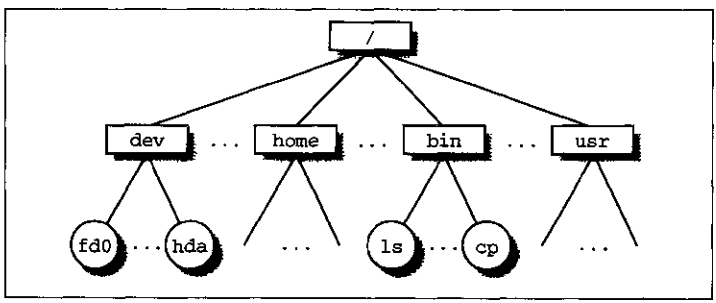


图 1-2　一个目录树的例子

除了叶子节点，树的所有节点都表示目录名。目录节点包含了它下面文件及目录的所有信息。文件或目录名是由一列字符组成，其中字符可以是任意一个 ASCII 字符（注 6），除“/”和空字符“\0”外。大多数文件系统对文件名的长度有一个限制，通常不能超过 255 个字符。与树的根相对应的目录被称为根目录。按照惯例，它的名字是“/”。在同一目录中的文件名必须不同，而在不同目录中的文件名可以相同。

Unix 的每个进程都有一个当前的工作目录（参见本章后面的“进程/内核模式”一节），它属于进程执行上下文（execution context），标识出进程所用的当前目录。为了标识一个特定的文件，进程使用路径名，路径名由斜杠及一列指向文件的目录名交替组成。如果路径名的第一个字符是斜杠，那么这个路径就是所谓的绝对路径，因为它的起点是根目录。否则，如果第一项是目录名或文件名，那么，这个路径就是所谓的相对路径，因为它的起点是进程的当前目录。

注 6：　一些操作系统允许以多种字符表来表示文件名，例如 Unicode，基于 16 位图形字符的扩展编码。

当标识文件名时，也用符号“.”和“..”。它们分别标识当前工作目录和父目录。如果当前工作目录是根目录，“.”和“..”是一致的。

硬链接和软链接

包含在一个目录中的文件名就是一个文件的硬链接（hard link），或简称链接。在同一目录或不同的目录中，同一文件可以有几个链接，因此对应几个文件名。

Unix 命令：

```
$ ln f1 f2
```

用来创建一个新的硬链接，即为路径f1标识的文件创建一个路径名为f2的硬链接。

硬链接有两方面的限制：

- 不允许给目录创建硬链接。因为这可能把目录树变为环形图，从而就不可能通过名字来定位一个文件。
- 只有在同一文件系统中的文件之间才能创建链接。这带来比较大的限制，因为现代 Unix 系统可能包含了几种文件系统，这些文件系统位于不同的磁盘/分区，用户也许没有注意到它们的物理划分。

为了克服这些限制，引入了软链接［即符号链接（symbolic link）］。符号链接是一个短文件，这个文件包含了另一个文件的任意一个路径名。这个路径名指向位于任意一个文件系统的任意文件，甚至可以指向一个不存在的文件。

Unix 命令：

```
$ ln -s f1 f2
```

创建一个路径名为 f2 的新的软链接指向路径名 f1。当这个命令执行时，文件系统创建一个新的软链接，并把路径名 f1 写入这个链接。然后插入（在合适的目录中）一个新的目录项，该目录包含路径 f2 中最后一个名字。以这种方式，任何对 f2 的引用都可以被自动转换成指向 f1 的一个引用。

文件类型

Unix 文件可以是下列类型之一：

- 正规文件（regular file）
- 目录（directory）
- 符号链（symbolic link）
- 块设备文件（block-oriented device file）
- 字符设备文件（character-oriented device file）
- 管道（pipe）和命名管道（named pipe）（即 FIFO）
- 套接字（socket）

前三种文件类型是任何 Unix 文件系统的基本类型。其实现将在第十七章中详细讨论。

设备文件与 I/O 设备以及集成到内核中的设备驱动程序相关。例如，当程序访问一个设备文件时，它直接作用于与那个文件相关的设备（参见第十三章）。

管道和套接字是用于进程间通信的特殊文件（参见本章后面的“同步和临界区”一节以及第十八章）。

文件描述符与索引节点

Unix 将文件和文件描述符（file descriptor）区分得非常清楚。除了设备文件和特殊文件，每个文件都由一列字符组成。这个文件不包含任何控制信息，如文件长度、文件结束符（EOF）。

文件系统用来管理文件的所有信息包含在一个叫做索引节点（inode）的数据结构中。每个文件都有自己的 inode，文件系统用它来识别一个文件。

虽然不同的 Unix 系统，其文件系统对 inode 的描述及其内核函数对 inode 的处理，差别可能非常大，但它们至少必须提供下面的属性，这些属性是在 POSIX 标准中指定的：

- 文件类型（参见前一节）。
- 与文件相关的硬链接个数。
- 以字节为单位的文件长度。
- 设备标识符（即包含这个文件的设备的标识符）。
- 用来在文件系统中标识文件的索引节点号。
- 文件拥有者的 UID。
- 文件的 GID。
- 几个时间标记，说明 inode 状态改变的时间、最后访问时间及最后修改时间。
- 访问权限和文件模式（mode）（参见下一节）。

访问权限和文件模式

文件潜在的用户分为三种类型：

- 作为这个文件所有者的用户。
- 同组用户，不包括所有者。
- 所有剩下的用户（其他）。

有三种类型的访问权限：读、写及执行，每组用户都有这三种权限。因此，文件访问权限的组合就用九种不同的二进制来标记。还有三种附加的标记，即 *suid*（Set User ID），*sgid*（Set Group ID）及 *sticky*（用来定义文件模式）。当这些标记应用到可执行文件时有如下含义：

`suid`

进程执行一个文件时通常保持进程拥有者的 UID。然而，如果设置了可执行文件 `suid` 的标志位，进程就获得了该文件拥有者的 UID。

`sgid`

进程执行一个文件时保持进程组的 GID。然而，如果设置了可执行文件 `sgid` 的标志位，进程就获得了该文件组的 ID。

sticky

设置了sticky标志位的可执行文件相应地对内核发出一个请求，当它执行结束以后，依然将该程序保留在内存中（注7）。

当进程创建一个文件时，文件拥有者的ID就是该进程的UID。而其组ID既可以是进程创建者的GID，也可以是父目录的GID，这取决于父目录sgid标志位的值。

文件操作的系统调用

当用户访问一个正规文件或目录文件的内容时，他实际上是访问存储在硬件块设备上的一些数据。从这个意义上说，文件系统是从用户级的观点来看硬盘分区的物理组织。因为处于用户态的进程不能直接与低层硬件打交道，所以，每个实际的文件操作必须在内核态下进行。

因此，Unix操作系统定义了几个与文件操作有关的系统调用。无论什么时候只要进程想对特定的文件施行一些操作，它就可以使用适当的系统调用，其参数是文件所在的路径名。

所有Unix内核都对硬件块设备的处理效率给予极大关注，其目的是为了获得非常好的系统整体性能。在本章下面的内容中，我们将描述Linux与文件操作相关的主题，尤其是讨论内核如何对文件相关的系统调用作出反应。为了理解这些内容，你得知道如何使用文件操作的主要系统调用。下面对此将给予描述。

打开一个文件

进程只能访问“打开的”文件。为了打开一个文件，进程调用系统调用：

```
fd = open(path, flag, mode)
```

其中的三个参数具有以下的含义：

path

表示被打开文件的路径（相对或绝对）。

注7：这个标志已经过时，现在使用基于代码页共享的其他方法（参见第七章）。

flag

指定文件打开的方式（例如，读、写、读/写，追加）。它也指定是否创建一个不存在的文件。

mode

指定新创建文件的访问权限。

这个系统调用创建一个“打开文件”对象，并返回所谓文件描述符的标识符。一个打开文件对象包括：

- 一些文件操作的数据结构，如指向内核内存区的缓冲区指针，这块内存区存放将要拷贝过来的文件的数据；offset域（即所谓文件指针）表示文件中当前的位置，从这个位置开始将进行下一个操作；等等。
- 指向进程被允许调用的一些内核函数的指针。这组允许的函数集合取决于参数flag的值。

我们将在第十二章中详细讨论打开文件对象。在这里，我们仅描述一些POSIX标准所指定的一般特性：

- 文件描述符表示了进程与打开文件之间的相互作用，而打开文件对象包含了与这种相互作用相关的数据。同一打开文件对象也许由几个文件描述符标识。
- 几个进程也许同时打开同一文件。在这种情况下，文件系统给每个文件分配一个单独的打开文件对象以及一个单独的文件描述符。当这种情况发生时，在这些进程对同一文件的I/O操作之间，Unix文件系统不提供任何形式的同步机制。然而，有几个系统调用，如flock()，允许进程对整个文件或部分文件实施同步（参见第十二章）。

为了创建一个新的文件，进程也可以调用create()系统调用，它与open()非常相似，都是由内核来处理。

访问一个打开的文件

对正规Unix文件，可以顺序地访问，也可以随机地访问，而对设备文件和命名管道文件，通常只能顺序地访问（参见第十三章）。在这两种访问方式中，内核把文件指针存放在打开的文件对象中，也就是说，当前位置就是进行下一次读或写操作的位置。

顺序访问隐含着一个假定，即：read()和write()系统调用总是指向当前文件指针的位置。为了修改这个位置，程序必须明确地调用lseek()系统调用。当打开一个文件时，内核把文件的指针设置在这个文件的第一个字节（偏移量为0）。

lseek()系统调用要求下列参数：

```
newoffset = lseek(fd, offset, whence);
```

其参数含义如下：

fd
: 表示打开文件的文件描述符。

offset
: 指定一个无符号整数，将用来计算文件指针新的位置。

whence
: 指定新的位置应该在什么地方，是否把offset值加到数字0上（即从文件开头的偏移），加到当前的文件指针上，或者加到文件最后一个字节上（从文件结尾的偏移）。

read()系统调用需要以下参数：

```
nread = read(fd, buf, count);
```

其参数含义如下：

fd
: 表示打开文件的文件描述符。

buf
: 指定在进程地址空间中缓冲区的地址，所读的数据就放在这个缓冲区。

count
: 表示所读的字节数。

当处理这样的系统调用时，内核会尝试从拥有文件描述符fd的文件中读count个字节，其起始位置为打开文件offset域的当前值。在某些情况下，可能遇到文件结束、空管道等等，内核无法成功地读出全部count个字节。返回的nread值就是

实际所读的字节数。文件指针也会更新为nread加上它原来的值。write()的参数与read()相似。

关闭文件

当进程无须再访问文件的内容时，它就调用系统调用：

```
res = close(fd);
```

释放与文件描述符fd相对应的打开文件对象。当一个进程终止时，内核关闭它所有仍然打开着的文件。

更名及删除文件

重新命名或删除一个文件时，进程不需要打开它。实际上，这样的操作并没有对这个文件的内容起作用，而是对一个或多个目录的内容起作用。例如，系统调用：

```
res = rename(oldpath, newpath);
```

改变了文件链接的名字，而系统调用：

```
res = unlink(pathname);
```

减少了文件链接数，删除了相应的目录项。只有当链接数为0时，文件才被真正删除。

Unix 内核概述

Unix内核提供了应用程序可以运行的执行环境。因此，内核必须实现一组服务及相应的接口。应用程序使用这些接口，而且通常不会与硬件资源直接打交道。

进程 / 内核模式

如前所述，CPU既可以运行在用户态下，也可以运行在内核态下。实际上，一些CPU可以有两种以上的执行状态。例如，Intel 80x86 微处理器有四种不同的执行状态。但是，所有标准的 Unix 内核都仅仅利用了内核态和用户态。

当一个程序在用户态下执行时，它不能直接访问内核数据结构或内核的程序。然而，当一个应用程序在内核态下运行时，这些限制不再有效。每种CPU模式都为从用户态到内核态的转换提供了特殊的指令，反之亦然。一个程序执行时，大部分时间都处在用户态下，只有需要内核所提供的服务时才切换到内核态。当内核满足了用户程序的需求后，它让程序又回到用户态下。

进程是动态的实体，在系统内通常只有有限的生存期。创建、撤消及同步已存在进程的任务都委托给内核中的一组例程来完成。

内核本身并不是一个进程而是进程的管理者。进程/内核模式假定，需要内核服务的进程使用被称为系统调用（system call）的特殊编程机制，每个系统调用都设置了一组参数来识别进程的要求，然后执行与硬件相关的CPU指令完成从用户态到内核态的转换。

除用户进程之外，Unix系统还包括几个所谓内核线程的特权进程，它们具有以下特点：

- 它们以内核态运行在内核地址空间。
- 它们不与用户直接交互，因此不需要终端设备。
- 它们通常在系统启动时创建，一直活跃着直到系统关闭。

注意进程/内核模式是怎样与CPU状态形成某种正交关系：在单处理器系统中，任何时候只有一个进程在运行，它要么处于用户态，要么处于内核态。如果进程运行在内核态，处理器就执行一些内核例程。图1-3举例说明了用户态与内核态之间的相互转换。处于用户态的进程1发出一个系统调用，之后，进程切换到内核态，系统调用被执行。然后，进程1恢复在用户态下执行，直到发生定时中断，且调度程序（scheduler）在内核态被激活。进程切换发生，进程2在用户态执行，直到硬件设备发出中断请求。中断的结果是，进程2切换到内核态并为中断提供服务。

Unix内核做的工作远不止系统调用。实际上，可以有几种方式激活内核例程：

- 进程调用一个系统调用。
- 正在执行进程的CPU发出一个异常信号（异常是一些反常情况，例如，一个无效的指令）。内核代表产生异常的进程处理异常。

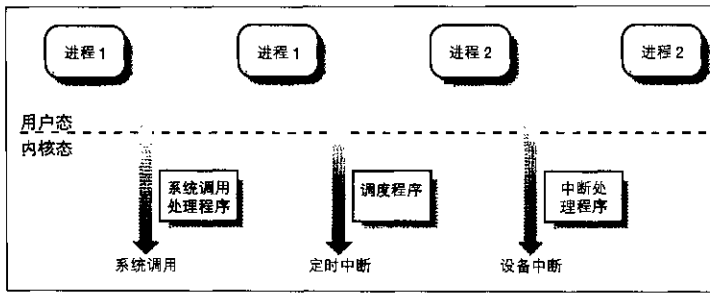


图 1-3 用户态与内核态之间的转换

- 外围设备向 CPU 发出一个中断信号以通知一个事件的发生，如一个要求注意的请求、一个状态的变化或一个 I/O 操作已经完成等。每个中断信号都是由内核中的中断处理程序来处理的。因为外围设备与 CPU 异步操作，因此，中断在不可预知的时间发生。
- 内核线程被执行。因为内核线程运行在内核态，所以，尽管其相应的程序被包装成一个进程，但必须认为它是内核的一部分。

进程的执行

为了让内核管理进程，每个进程由一个进程描述符（process descriptor）表示，这个描述符包含有关进程当前状态的信息。

当内核暂停一个进程的执行时，它在进程描述符中保存几个处理器寄存器的内容。这些寄存器包括：

- 程序计数器（PC）和栈指针（SP）寄存器
- 通用寄存器
- 浮点寄存器
- 包含 CPU 状态信息的处理器控制寄存器（Processor Status Word，PSW）
- 用来跟踪进程对 RAM 访问的内存管理寄存器

当内核决定恢复执行一个进程时，它用进程描述符中合适的域来装载CPU的寄存器。因为程序计数器中所存的值指向下一条将要执行的指令，进程从它停止的地方恢复执行。

当一个进程不在CPU上执行时，它正在等待一些事件。Unix内核可以区分很多等待状态，这些等待状态通常由进程描述符队列实现。每个队列（可能是空）对应一组等待特定事件的进程。

可重入内核

所有的Unix内核都是可重入的（reentrant），这意味着几个进程可以同时在内核态下执行。当然，在单处理器系统上，只有一个进程在真正运行，但是许多进程可以在内核态下被阻塞，或等待CPU，或等待一些I/O操作的完成。例如，当内核代表进程发出一个读请求后，就让磁盘控制器处理这个请求，并将恢复其他进程的执行。当这个设备满足了读请求时，一个中断就会通知内核，从而以前的进程可以恢复执行。

提供可重入的一种方式是编写函数，以便这些函数只能修改局部变量，而不能改变全局数据结构，这样的函数叫可重入函数。但是可重入内核不仅仅局限于这样的可重入函数（尽管一些实时内核正是如此实现的）。内核可以包含非重入函数，并且利用锁机制保证一次只有一个进程执行一个非重入函数。处于内核态的每个进程只能作用于自己的内存空间，不能干预其他的进程。

如果发生一个硬件中断，可重入内核能挂起当前正在执行的进程，即使这个进程处于内核态。这种能力是非常重要的，因为这能提高发出中断的设备控制器的吞吐量。一旦设备发出一个中断，它一直等到CPU应答它为止。如果内核能够快速应答，在CPU处理中断的时候设备控制器将能执行其他任务。

现在，让我们看一下内核的可重入性及它对内核组织的影响。内核控制路径（kernel control path）表示由内核执行的指令序列，用来处理系统调用、异常及中断。

在一种最简单的情况下，CPU从第一条指令到最后一条指令顺序地执行内核控制路径。然而，当下述事件之一发生时，CPU交错执行内核控制路径：

- 在用户态下执行的进程调用一个系统调用，其相应的内核控制路径证明这个请

求不能立即得到满足。然后，调用调度程序选择一个新的进程投入运行。结果，发生进程切换。第一个内核控制路径还没完成，CPU就重新开始执行其他的内核控制路径。在这种情况下，两条控制路径代表两个不同的进程在执行。

- 当运行一个内核控制路径时，CPU检测到一个异常（例如，访问的页不在RAM中）。第一个控制路径被挂起，CPU开始执行合适的过程。在我们的例子中，这个合适的过程能给进程分配一个新页，并从磁盘读它的内容。当这个过程结束时，第一个控制路径能恢复执行。在这种情况下，两个控制路径代表同一个进程在执行。
- 当CPU正在运行一个开中断（interrupt enabling）的内核控制路径时，发生了一个硬件中断。第一个内核控制路径还没执行完，CPU开始执行另一个内核控制路径来处理这个中断。当这个中断处理程序终止时，第一个内核控制路径恢复。在这种情况下，两个内核控制路径运行在同一进程的可执行上下文中，所花费的系统时间都算给这个进程。然而，中断处理程序无须代表这个进程运行。

图1-4显示了非交错和交错执行内核控制路径的几个例子。考虑以下三种不同的CPU状态：

- 在用户态下运行一个进程（User）
- 运行一个异常处理程序或系统调用处理程序（Excp）
- 运行一个中断处理程序（Intr）

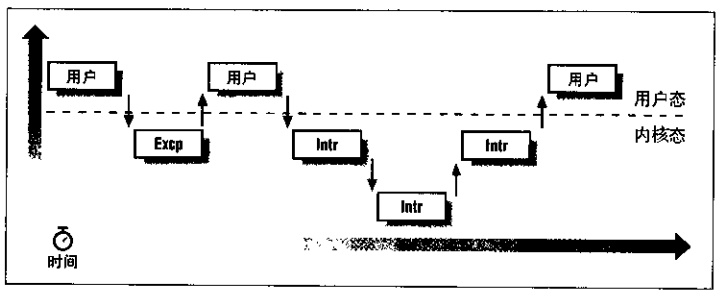


图1-4 内核控制路径的交错执行

进程地址空间

每个进程运行在它私有的地址空间。在用户态下运行的进程涉及到私有栈、数据和代码区。当在内核态运行时，进程访问内核的数据和代码区，但使用另外的栈。

因为内核是可重入的，几个内核控制路径（每个都与不同的进程相关）可以轮流执行。在这种情况下，每个内核控制路径都有它自己的私有内核态堆栈。

尽管每个进程看起来访问一个私有地址空间，但有时进程之间也共享部分地址空间。在一些情况下，进程明确地请求这种共享；在另外一些情况下，由内核自动完成共享，以节约内存的使用。

如果同一个程序（比如说编辑程序）由几个用户同时使用，则这个程序只被装入内存一次，其指令由所有需要它的用户共享。当然，其数据不被共享，因为每个用户将有独立的数据。这种共享的地址空间是由内核自动完成以节省内存。

进程间也能共享其部分地址空间来进行进程间通信，使用一种由 System V 引入并且已经被 Linux 支持的“共享内存”技术。

最后，Linux 支持 mmap() 系统调用，它允许部分文件或者驻留在设备中的存储能被映射到一个进程的部分地址空间。内存映射为正常的以读写方式传送数据提供了另一种选择。如果同一文件由几个进程共享，它的内存映射被包含在共享它的每一个进程的地址空间中。

同步和临界区

实现可重入内核需要利用同步机制：如果作用于内核数据结构的内核控制路径被挂起，那么，其他的内核控制路径就不能再作用于该数据结构，除非它已被重新设置成一个一致性（consistent）状态。否则，两个控制路径的交互作用将破坏所存储的信息。

例如，让我们假设全局变量 V 包含一些可使用的系统资源项的个数。第一个内核控制路径 A 读这个变量，并且确定仅有一个可用的资源项。这时，另一个内核控制路径 B 被激活，并读同一个变量 V，V 的值仍为 1。因此，B 对 V 减 1，并开始用这个资源项。然后，A 恢复执行。因为 A 已经读到 V 的值，它假定可以对 V 减 1，并使

用这个资源项，其实，B已经在使用。结果，V的值变为-1，两个内核控制路径在使用相同的资源项，有可能导致灾难性的后果。

当一些计算结果的得出取决于两个或多个进程如何被调度时，这段代码就是不正确的，我们说存在一种竞争条件（race condition）。

一般来说，对全局变量的安全访问通过原子操作（atomic operation）来保证。在前面的例子中，如果两个控制路径读V并减1是一个单独的、不可中断的操作，那么，就不可能出现数据错误。然而，内核包含很多不能用单一的操作访问的数据结构。例如，用单一的操作从链表中删除一个元素是不可能的，因为内核一次至少访问两个指针。临界区（critical region，注8）是这样的一段代码，进入这段代码的进程必须完成以后，另一个进程才能进入。

这些问题不仅出现在内核控制路径中，也出现在共享公共数据的进程之间。几种同步技术已经被采用。以下将集中讨论怎样同步内核控制路径。

非抢占式内核

在寻找彻底、简单地解决同步问题的方案中，大多数传统的Unix内核都是非抢占式的：当进程在内核态执行时，它不能被任意挂起，也不能被另一个进程代替。因此，在单处理器系统上，所有中断或异常处理程序不能更新的内核数据结构，内核对它们的访问都将是安全的。

当然，内核态的进程能自愿放弃CPU，但是，在这种情况下，它必须确保所有的数据结构都处于一致性状态。此外，当这个进程恢复执行时，必须重新检查以前访问过的数据结构的值，因为这些数据结构有可能被改变。

非抢占能力在多处理器系统上是低效的，因为运行在不同CPU上的两个内核控制路径本可以并发地访问相同的数据结构。

关中断

单处理器系统上的另一种同步机制是：在进入一个临界区之前禁止产生所有的硬件

注8：同步问题已在其他著作中进行了详细描述。有兴趣的读者可以参考有关Unix操作系统方面的书（参见本书末尾的参考书目）。

中断（interrupt disabling），离开临界区时再允许产生中断。这种机制尽管简单，但远不是最佳的。如果临界区比较大，关中断保持相对较长的时间就可能使所有的硬件活动处于冻结状态。

此外，在多处理器系统上，这种机制根本不起作用。无法保证其他的CPU不访问这个受保护的临界区中同一个数据结构。

信号量

一种广泛使用的机制是信号量（semaphore），它在单处理器系统和多处理器系统上都有效。信号量仅仅是与一个数据结构相关的一个计数器。当所有内核线程访问这个数据结构前，都要检查这个信号量。可以把每个信号量看成一个对象，其组成如下：

- 一个整数变量
- 一个等待进程的链表
- 两个原子方法：down()和up()

down()方法对信号量的值减1，如果这个值小于0，它就把正在运行的进程加入到这个信号量链表，然后阻塞该进程（即调用scheduler）。up()方法对信号量的值加1，如果这个值大于或等于0，激活这个信号量的链表中的一个或多个进程。

每一个要保护的数据结构都有它自己的信号量，其初始值为1。当内核控制路径希望访问这个数据结构时，它在适当的信号量上执行down()方法。如果新信号量的值不是负数，则允许访问这个数据结构。否则，把执行内核控制路径的进程加入到这个信号量的链表并阻塞它。当另一个进程在那个信号量上执行up()方法时，允许信号量链表上的进程之一继续执行。

自旋锁

在多处理器系统中，信号量并不总是解决同步问题的最佳方案。一些内核数据结构应该得到保护，以防止运行在不同CPU上的内核控制路径同时访问。在这种情况下，如果更新数据结构所需的时间比较短，那么，信号量可能是很低效的。为了检查信号量，内核必须把进程插入到信号量链表中，然后挂起它。因为这两种操作比较费时，完成这些操作时，其他的内核控制路径可能已经释放了这个信号量。

在这些情况下，多处理器操作系统使用了自旋锁（spin lock）机制。自旋锁与信号量非常相似，但没有进程链表：当一个进程发现锁被另一个进程锁着时，它就不停地“旋转”，不断执行一个指令的循环直到锁打开。

当然，自旋锁在单处理器环境下是无用的。当内核控制路径试图访问一个上锁的数据结构时，它开始无终止的循环。因此，正在更新受保护的数据结构的内核控制路径将没有机会继续执行和释放这个自旋锁。最后的结果是系统挂起。

避免死锁

与其他控制路径同步的进程或内核控制路径很容易进入死锁状态。举一个最简单的死锁的例子，进程*p1*获得访问数据结构*a*的权限，进程*p2*获得访问*b*的权限，但是*p1*在等待*b*，而*p2*在等待*a*。进程之间其他更复杂的循环等待的情况也可能发生。显然，死锁情形会引起受影响的进程或内核控制路径完全处于冻结状态。

只要涉及到内核设计，当所用内核信号量的类型增多时，死锁就成为一个突出问题。在这种情况下，很难保证内核控制路径在各种可能方式下的交错执行不出现死锁状态。有几种操作系统，包括Linux，通过以下两种方式避免死锁，即引入很有限的信号量类型和以递增的顺序请求信号量。

信号和进程间通信

Unix信号提供了把系统事件报告给进程的一种机制。每种事件都有自己的信号编号，通常用一个符号常量来表示，例如SIGTERM。有两种系统事件：

异步通告

例如，当用户在终端按下中断键（通常为CTRL-C）时，即向前台进程发出中断信号SIGINT。

同步错误或异常

例如，当进程访问内存非法地址时，内核向这个进程发送一个SIGSEGV信号。

POSIX标准定义了大约20种不同的信号，其中，有两个是用户自定义的，可以当作用户态下进程通信和同步的原语机制。一般来说，进程可以以两种方式对信号的接收做出反应：

- 忽略该信号。
- 异步地执行一个指定的过程（信号处理程序）。

如果进程不指定选择何种方式，内核就根据信号的编号执行一个缺省操作。五种可能的缺省操作是：

- 终止进程。
- 将执行上下文和进程地址空间的内容写入一个文件（核心转储core dump），并终止进程。
- 忽略信号。
- 挂起进程。
- 如果进程曾被暂停，恢复它的执行。

因为POSIX标准允许进程暂时阻塞信号，因此，内核信号的处理相当精细。此外，像SIGKILL这样的几个信号进程是不能直接处理的，也不能忽略它们。

AT&T的Unix System V引入了其他种类的在用户态下进程间通信机制，很多Unix内核也采用了它们：信号量，消息队列及共享内存。它们被统称为*System V IPC*。

内核把它们作为IPC资源来实现：进程要获得一个资源，可以调用shmget()，semget()或msgget()系统调用。与文件一样，IPC资源是持久不变（persistent）的，进程创建者、进程拥有者或超级用户进程必须明确地释放这些资源。

这里的信号量与本章“同步和临界区”一节中所描述的概念是相似的，只是用在用户态下的进程中。消息队列允许进程利用msgsnd()及msgget()系统调用交换消息，msgsnd()表示把消息插入到指定的队列中，msgget()表示从队列中提取消息。

共享内存为进程之间交换和共享数据提供了最快的方式。通过调用shmget()系统调用来创建一个新的共享内存，其大小按需设置。在获得IPC资源标识符后，进程调用shmat()系统调用，其返回值是进程的地址空间中新区域的起始地址。当进程希望把共享内存从地址空间分离出去时，它调用shmdt()系统调用。共享内存的实现依赖于内核如何实现进程的地址空间。

进程管理

在进程和它正在执行的程序之间，Unix做出一个清晰的划分。fork()和exit()系统调用分别被用来创建一个新进程和终止一个进程，而调用类exec()系统调用则是装入一个新程序。当执行了这样的系统调用以后，进程在包含所装入程序的新地址空间恢复运行。

调用fork()的进程是父进程，而新进程是它的子进程。父子进程能互相找到对方，因为描述每个进程的数据结构都包含有两个指针，一个直接指向它的父进程，另一个直接指向它的子进程。

fork()的一个很自然的实现是将父进程的数据与代码都复制，并把这个拷贝赋给子进程，这将相当费时。依赖硬件分页单元的内核采用写时复制（Copy-On-Write）技术，即把页的复制延迟到最后一刻（也就是，直到父或子进程需要时才写进页）。我们将在第七章“写时复制”一节中描述Linux是如何实现这一技术的。

exit()系统调用终止一个进程。内核对这个系统调用的处理是通过释放进程所拥有的资源，向父进程发送SIGCHILD信号，这个信号的缺省操作是忽略。

僵死进程

父进程如何查询子进程的终止呢？wait()系统调用允许进程等待，直到其子进程之一结束，它返回被终止的子进程的进程标识符（Process ID，PID）。

内核执行这个系统调用时，检查子进程是否已经终止。引入僵死进程（zombie process）状态是为了表示已终止的进程。进程停留在这种状态，一直到父进程执行了wait()系统调用。从进程描述符域中，系统调用处理程序获得有关资源使用的一些数据。一旦得到数据，就可以释放进程描述符。当进程执行wait()系统调用时，如果没有子进程结束，内核通常把该进程设置成等待状态，一直到子进程结束。

很多内核也执行waitpid()系统调用，它允许进程等待一个特殊的子进程。其他wait()系统调用的变体也是相当通用的。

对内核来说，保持子进程的有关信息一直到父进程产生wait()调用是一个不错的方法，但是，假设父进程终止而没有发布wait()调用呢？这些信息占用了一些内存中非常有用的位置，而这些位置本来可以用来为活动着的进程提供服务。例如，

很多shell允许用户在后台启动一个命令然后退出。正在运行命令shell的进程会终止，但它的子进程继续运行。

解决的办法是使用一个叫*init*的特殊系统进程，它在系统初始化的时候被创建。当一个进程终止时，内核改变它的描述符中所有子进程的指针，使这些子进程成为*init*的孩子。*init*监控所有子进程的执行，并且例行公事地发布wait()系统调用，其副作用是除去所有僵死的进程。

进程组和登录会话

现代Unix操作系统引入了进程组（process group）的概念以表示“作业（job）”的抽象。例如，为了执行命令行：

```
$ ls | sort | more
```

一个支持进程组的shell，例如bash，为三个相应的进程ls、sort及more创建了一个新的组。shell以这种方式作用于这三个进程，好象它们是单个的实体（更准确地说：作业）。每个进程描述符包含了一个进程组ID域。每一进程组可以有一个领头进程，其PID与这个进程组的ID相同。一个新创建的进程最初被插入到其父进程的进程组中。

现代Unix内核也引入了登录会话（login session）。非正式地说，一个会话包括一个进程的所有后代进程，这个进程就是在指定终端开始工作会话的进程（通常情况下，第一个shell命令进程是为用户创建的）。进程组中的所有进程必须在同一登录会话中。一个登录会话可以有几个进程组同时处于活动状态，其中，只有一个进程组一直处于前台，这意味着它可以访问终端，而其他活动着的进程在后台。当一个后台进程试图访问终端时，它将收到SIGTTIN或SIGTTOUT信号。在很多shell命令中，内部命令bg和fg用来把一个进程组放在后台或者前台。

内存管理

到目前为止，内存管理是Unix内核中最复杂的活动。在这本书里，我们将用超过三分之一的篇幅来描述Linux是如何实现它的。这部分只说明一些与内存管理相关的主要问题。

虚拟内存

所有现代的Unix系统都提供了一种有用的抽象，叫虚拟内存（virtual memory）。虚拟内存作为一种逻辑层，处于应用程序对内存的申请与硬件内存管理单元（Memory Management Unit，MMU）之间。虚拟内存有很多用途和优点：

- 几个进程可以并发地执行。
- 应用程序所需内存大于可用物理内存时，也可以运行。
- 进程可以执行只有部分代码装入到内存的程序。
- 允许每个进程访问可用物理内存的一个子集。
- 进程可以共享库函数或程序的单一的内存映像。
- 程序是可重定位的，也就是说，可以把程序放在物理内存的任何地方。
- 编程者可以编写与机器无关的代码，因为他们不必关心有关物理内存的组织结构。

虚拟内存子系统的主要因素是虚拟地址空间（virtual address space）的概念。进程所用的一组内存地址不同于物理内存地址。当进程使用一个虚拟地址时（注9），内核和MMU协作定位其在内存中的实际物理位置。

现在的CPU包含了能自动地把虚拟地址转换成物理地址的硬件电路。为了达到这个目标，把可用RAM划分成长度为4K或8K的页框（page frame），并且引入一组页表来指定虚地址与物理地址之间的对应关系。这些机制使得内存分配变得简单，因为一组非连续的物理地址页框可以满足一块连续的虚拟地址的请求。

随机访问内存（RAM）的使用

所有的Unix操作系统都将RAM明显区分为两部分，其中若干兆的字节专门用于存储内核映像（也就是内核代码和内核静态数据结构），RAM的其余部分通常由虚拟存储器系统来处理，并且以以下三种可能的方式来使用：

注9：这些地址的叫法在不同的计算机体系结构中是不一样的。正如我们在第二章中会看到的一样，Intel 80x86使用手册把它们叫做“逻辑地址”。

- 满足内核对缓存、描述符及其他动态内核数据结构的请求。
- 满足进程对一般内存区域的请求及对文件的内存映射的请求。
- 用高速缓存的方法从磁盘及其他缓冲设备获得较好的性能。

每种请求类型都是有价值的。但从另一方面来说，因为可用RAM是有限的，必须在请求类型之间做出平衡，尤其是当可用内存没有剩下多少时。此外，当可用内存达到临界极限时，可以调用页框回收算法（page-frame-reclaiming）释放额外的内存，哪种算法是最合适的页框回收算法呢？正如我们将在第十六章中看到的一样，对这个问题既没有简单的答案，也没有多少理论的支持，唯一可用的解决方法在于根据经验仔细地开发和谐的算法。

虚拟内存必须解决的一个主要问题是内存碎片。理想情况下，只有当空闲页框数太少时，内存请求才失败。然而，内核通常被迫使用物理上连续的内存区域，因此，即使有足够的可用内存，但它不能作为一个连续的大块使用时，内存的请求也会失败。

内核内存分配器

内核内存分配器（Kernel Memory Allocator，KMA）是一个子系统，它试图满足系统中所有部分对内存的请求。其中一些请求来自内核其他子系统，它们需要一些内存供内核使用，还有一些请求来自于用户程序的系统调用，用来增加用户进程的地址空间。一个好的KMA应该具有下列特点：

- 必须快。实际上，这是最重要的属性，因为它由所有的内核子系统调用（包括中断处理程序）。
- 必须把内存的浪费减到最少。
- 必须努力减轻内存的碎片（fragmentation）问题。
- 必须能与其他内存管理子系统合作，以便借用和释放页框。

已经提出了几种KMA，基于各种不同的算法技术，包括：

- 资源图分配算法（allocator）
- 2的幂次方空闲链表

- McKusick-Karels 分配算法
- 伙伴（Buddy）系统
- Mach 的区域（Zone）分配算法
- Dynix 分配算法
- Solaris 的 slab 分配算法

我们将在第六章中看到，Linux 的 KMA 在伙伴系统的上部采用了 slab 分配算法。

进程虚拟地址空间的处理

一个进程的虚拟地址空间包括所有进程被允许引用的虚拟内存地址。内核通常把进程虚拟地址空间当作内存区域描述符的链表来保存，例如，当进程通过 exec()类系统调用执行一些程序时，内核分配给进程的虚拟地址空间由以下内存区域组成：

- 程序的可执行代码
- 程序的初始化数据
- 程序的未初始化数据
- 初始程序栈（即用户态栈）
- 需要共享的库的可执行代码和数据
- 堆（由程序动态请求的内存）

所有现代Unix操作系统都采用了所谓请求调页（demand paging）的内存分配策略。有了请求分页，进程可以在它的页还没有在内存时就开始执行。当进程访问一个不存在的页时，MMU 产生一个异常；异常处理程序找到受影响的内存区域，分配一个空闲的页，并用适当的数据把它初始化。同理，当进程通过调用malloc()或brk()（由malloc()内部调用）系统调用动态地请求一些内存时，内核仅仅更新进程的堆内存区域的大小。只有当试图引用进程的虚拟内存地址而产生异常时，才给进程分配页框。

虚拟地址空间也采用其他更有效的策略，如前面提到的写时复制策略。例如，当一个新进程被创建时，内核仅仅把父进程的页框赋给子进程的地址空间，但是，把这

些页框标记为只读。一旦父或子进程试图修改页中的内容时，一个异常就会产生。异常处理程序把新页框赋给受影响的进程，并用原来页中的内容初始化新页。

交换与高速缓存

为了扩充进程所用的虚拟地址空间的大小，Unix 操作系统使用磁盘上的交换区域（swap area）。虚拟内存系统以一个页框的内容作为交换的基本单位。当一些进程引用已换出的页时，MMU产生一个异常，然后，异常处理程序分配一个新的页框，并用存储在磁盘上的旧内容初始化该页框。

另一方面，物理内存也被用做磁盘和其他块设备的高速缓存。这是因为硬盘非常慢，磁盘的访问需要几毫秒，与RAM的访问时间相比，这是相当长的时间。因此，磁盘通常是影响系统性能的瓶颈。作为一般规则，在最早的Unix系统就已经实现的一个策略是：通过把从磁盘读出的块装入到RAM的一组磁盘缓冲区（buffer）中来尽可能地推迟写磁盘的时间。sync()系统调用把所有“脏”的缓冲区（即缓冲区的内容与对应磁盘块的内容不一样）写入磁盘来使得磁盘同步。为了避免数据丢失，所有的操作系统都注意会周期性地把脏缓冲区写回磁盘。

设备驱动程序

内核通过设备驱动程序（device driver）与I/O设备打交道。设备驱动程序包含在内核中，由控制一个或多个设备的数据结构和函数组成，这些设备包括硬盘、键盘、鼠标、监视器、网络接口及连接到SCSI总线上的设备。通过特定的接口，每个驱动程序与内核中的其余部分（甚至与其他驱动程序）相互作用，这种方式具有以下优点：

- 可以把特定设备的代码封装在特定的模块中。
- 厂商可以不懂内核源代码，只知道接口规范，就能增加新的设备。
- 内核以统一的方式对待所有的设备，并且通过相同的接口访问这些设备。
- 可以把设备驱动程序写成模块，并动态地把它们装进内核，不需要重新启动系统。不再需要时，也可以卸下模块，以减少存储在RAM中内核映像的大小。

图1-5说明了设备驱动程序与内核其他部分及进程之间的接口。一些用户程序（P）

希望操作硬件设备，P就利用常用的与文件相关的系统调用及在*/dev*目录下能找到的文件向内核发出请求。实际上，设备文件是设备驱动程序接口中用户可见的部分。每个设备文件都有专门的设备驱动程序，它们由内核调用以执行对硬件设备的请求操作。

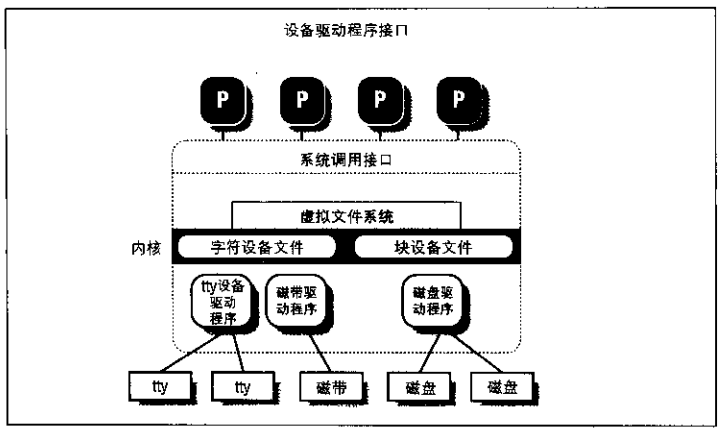
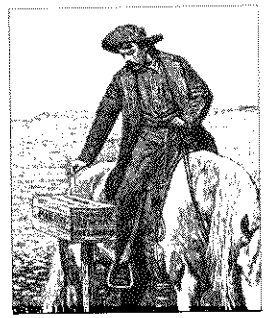


图1-5 设备驱动程序接口

这里值得一提的是，在Unix刚出现的时候，图形终端是罕见而且昂贵的，因此Unix内核只直接处理字符终端。当图形终端变得非常广泛时，一些如X Window系统那样的特别的应用就出现了，它们以标准进程的身份运行并且能直接访问图形界面的I/O端口和RAM的视频区域。一些新近的Unix内核，例如Linux 2.2，对一些帧缓冲设备只有有限的支持，因此，内核允许程序通过设备文件访问视频卡内的本地内存。

第二章
内存寻址



本章介绍寻址（addressing）技术。值得庆幸的是，如今的微处理器硬件线路可以使内存管理既高效又健壮，因此操作系统本身已经不必完全了解物理内存了。

作为本书的一部分，本章将详细描述Intel 80x86微处理器怎样进行芯片级的内存寻址，Linux又是如何利用寻址硬件的。我们希望当你学习内存寻址技术在Linux最流行的硬件平台上的详细实现方法时，能够更好地理解分页单元的一般原理，更好地研究内存寻址技术在其他平台上是如何实现的。

关于内存管理有三章，这是其中的第一章；还有第六章，讨论内核怎样给自己分配主存；以及第七章，考虑怎样给进程分配线性地址。

内存地址

程序员很自然地会使用内存地址（memory address）访问内存单元的内容，但是，当使用Intel 80x86微处理器时，我们必须区分以下三种不同的地址：

逻辑地址（logical address）

机器语言指令仍用这种地址指定一个操作数或一条指令的地址。这种寻址方式在Intel有名的分段结构中表现得尤为具体，它促使MS-DOS或Windows程序

员把程序分成若干段。每一个逻辑地址都由一个段(segment)和偏移量(offset)组成，偏移量指明了从段开始的地方到实际地址之间的距离。

线性地址 (linear address)

是一个32位无符号整数，可以用来表示高达4GB的地址，也就是，高达4 294 976 296个内存单元。线性地址通常用16进制数字表示，值的范围从0x00000000到0xffffffff。

物理地址 (physical address)

用于芯片级内存单元寻址。它们与从微处理器的地址引脚发送到内存总线上的电信号相对应。物理地址由32位无符号整数表示。

CPU控制单元通过一种称为分段单元（segmentation unit）的硬件电路把一个逻辑地址转换成线性地址；接着，第二个称为分页单元（paging unit）的硬件电路把一个线性地址转换成物理地址（见图2-1）。

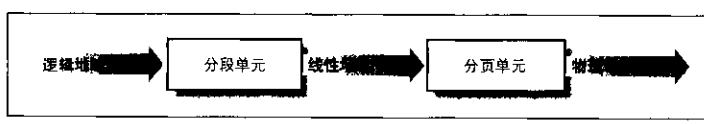


图2-1 逻辑地址转换

硬件的分段单元

从80386 CPU开始，Intel微处理器能执行两种不同的地址转换模式，分别称为实模式（real mode）和保护模式（protected mode）。保留实模式的主要原因是要与早期的CPU保持兼容，并让操作系统自举(参阅附录一中针对实模式的简短描述)。我们的重点集中在保护模式上。

段寄存器

一个逻辑地址由两部分组成：一个段标识符和一个指定段内相对地址的偏移量。段标识符是一个16位长的域，称为段选择符(segment selector)，偏移量是一个32位长的域。

为了快速方便地找到段选择符，处理器提供段寄存器（segmentation register），目的是存放段选择符。这些段寄存器称为cs，ss，ds，es，fs和gs。尽管只有6个段寄存器，但程序可以把同一个段寄存器用于不同的目的，方法是先将其值保存在内存中，用完后再恢复。

6个寄存器中3个有专门的用途：

cs

代码段寄存器，指向存放程序指令的段。

ss

栈段寄存器，指向存放当前程序栈的段。

ds

数据段寄存器，指向存放静态数据或者外部数据的段。

其他三个段寄存器作一般用途，可以用来访问任意的段。

cs寄存器还有一个很重要的功能：它含有一个两位的域，用以指明CPU的当前特权级（Current Privilege Level，CPL）。值为0代表最高优先级，而值为3代表最低优先级。Linux只用0级和3级，分别称之为内核态和用户态。

段描述符

每个段由一个8字节的段描述符（segment descriptor）来表示（参见图2-2），它描述了段的特征。段描述符被放在全局描述符表（Global Descriptor Table，GDT）或局部描述符表（Local Descriptor Table，LDT）中。

系统通常只定义一个GDT，而每个进程可以有自己的LDT。GDT在主存中的地址被放在处理器的gdtr寄存器中，当前正被使用的LDT的地址被放在处理器的ldtr寄存器中。

每一个段描述符由以下域组成：

- 32位的Base域，含有段的第一个字节的线性地址。
- 粒度位G，如果被清0，段大小以字节为单位，否则单位以4096字节计。

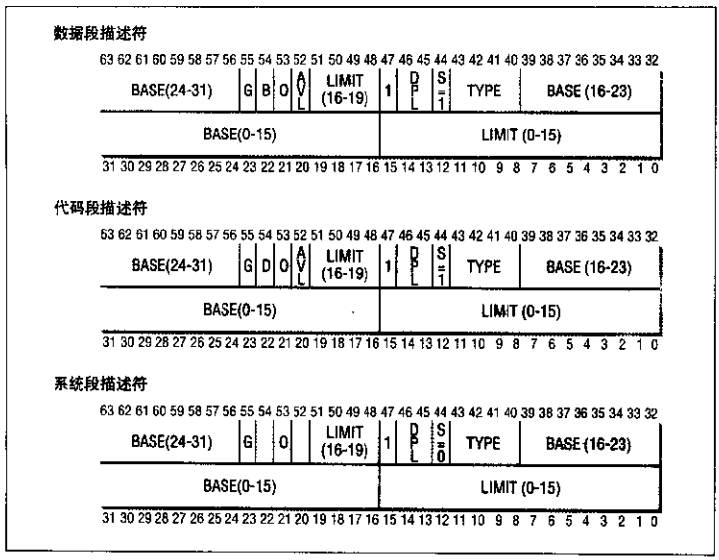


图 2-2 段描述符格式

- 20 位的 Limit 域，指定了以字节为单位的段长度，如果 G 被置为 0，则一个非空段的大小在 1 个字节到 1MB 间变化；否则，将在 4KB 到 4GB 之间变化。
- 系统标志 S，如果它被清 0，则这是一个系统段，存储内核数据结构，否则它是一个普通的代码段或数据段。
- 4 位 Type 域，描述了段的类型特征和它的存取权限。以下类型的段描述符被广泛采用：

  代码段描述符

  这个段描述符代表一个代码段，可以放在 GDT 或 LDT 中，S 标志为 1。

  数据段描述符

  这个段描述符代表一个数据段，可以放在 GDT 或 LDT 中，S 标志为 1。栈段是通过一般的数据段实现的。

任务状态段描述符（TSSD）

这个段描述符代表一个任务状态段（Task State Segment，TSS），也就是说这个段用于保存处理器寄存器的内容（参见第三章中的"任务状态段"一节）。它只能放在GDT中。根据相应的进程是否正在CPU上运行，其Type域的值分别为11或9，S标志为0。

局部描述符表描述符（LDTD）

这个段描述符代表一个LDT段，它只能放在GDT中，相应的Type域为2，S标志为0。

- 2位DPL（描述符特权级，Descriptor Privilege Level）域，用于限制对这个段的存取。它表示为访问这个段而要求的CPU最小的优先级。例如，一个DPL设为0的段只能当CPL为0时才能被访问，类似地，在内核态中，当一个段的DPL被设为3，可以被具有任何值的CPL访问。
- Segment-Present标志，被设为0，表示这个段当前不在主存中。Linux总是把这个域设为1，因为它从来不把整个段交换到磁盘上去。
- 一个被称为D或B的额外标志，取决于是代码段还是数据段。D或B的含义在两种情况下稍微有所区别，但是，如果段偏移量的地址是32位长，它就被置为1，如果这个偏移量是16位长，它就被清0（参见Intel使用手册的更详细描述）。
- 第53位是保留位，总是设为0。
- AVL标志，可以被操作系统使用，但是被Linux忽略掉。

段选择符

为了加速逻辑地址到线性地址的转换，Intel处理器提供一种附加的非编程的寄存器（一个不能被程序员所设置的寄存器），供6个可编程的段寄存器使用。每一个非编程的寄存器含有8个字节的段描述符（在前一节已讲述），由相应的段寄存器中的段选择符所指定。每当一个段选择符被装入段寄存器，相应的段描述符就由内存装入到对应的非编程的CPU寄存器。从那时起，针对那个段的逻辑地址转换就可以不访问主存中的GDT或LDT而进行，处理器只需直接引用存放段描述符的CPU寄存器即可。仅当段寄存器的内容改变时，才有必要访问GDT或LDT（参见图2-3）。每个段选择符包含以下域：

- 13位的索引（在下面的部分做进一步的讲述），指定了放在GDT或LDT中的相应段描述符的入口。
- TI（描述符表指示符）标志指明了段描述符是在GDT中（TI=0）或在LDT中（TI=1）。
- 两位RPL（请求特权级）域描述了当相应的段选择符装入到cs寄存器中时，CPU的当前特权级（注1）。

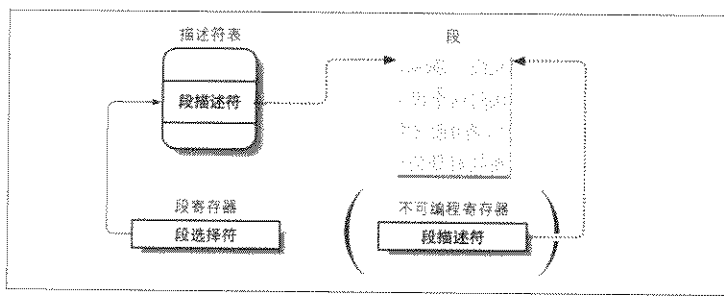


图2-3 段选择符和段描述符

由于一个段描述符是8字节长，它在GDT或LDT内的相对地址是由段选择符的最高13位的值乘以8得到的。例如：如果GDT在0x00020000（这个值保存在gdtr中），由段选择符所指定的索引号为2，那么相应的段描述符地址是0x00020000+(2×8)，或0x00020010。

GDT的第一项总是设为0，这样可以保证段选择符为空的逻辑地址被认为是无效的，因此引起一个处理器异常。能够保存在GDT中的段描述符的最大数目是8191，即$2^{13}-1$。

段单元

图2-4详细显示了一个逻辑地址是怎样转换成相应的线性地址的。段单元（segmentation unit）执行以下操作：

注1：RPL域还可用于当访问数据段时有选择地削弱处理器的特权级。详情请参见Internet的有关文档资料。

- 先检查段选择符的TI域，用于决定段描述符保存在哪一个描述符表中。TI域指明了描述符是在GDT中（在这种情况下，段单元从gdtr寄存器中得到GDT的线性基地址）还是在激活的LDT中（在这种情况下，段单元从ldtr寄存器中得到LDT的线性基地址）。
- 从段选择符的索引域计算段描述符的地址，索引域的值乘以8（一个段描述符的大小），这个结果被放到gdtr或ldtr寄存器中。
- 把逻辑地址的偏移量与段描述符基地址域的值相加，就得到了线性地址。

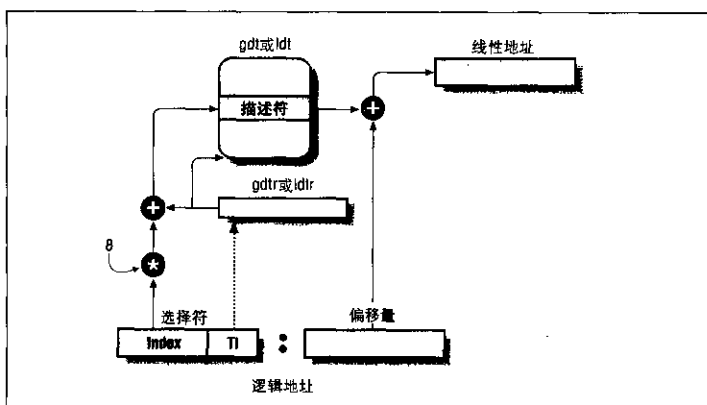


图2-4 逻辑地址的转换

请注意，由于与段寄存器相关的不可编程的寄存器的存在，只有当段寄存器的内容被改变时才需要执行前两个操作。

Linux中的段

Intel微处理器中的段方案鼓励程序员把他们的程序化分成逻辑实体，例如子程序或者全局与局部数据区。然而，Linux非常有限地使用段方案。实际上，段和页在某种程度上有点多余，因为它们都可以划分进程的物理空间：段可以给每一个进程分

配不同的线性地址空间，而页可以把同样的线性地址空间映射到不同的物理空间。与分段相比，Linux 更喜欢使用分页方式，因为：

- 当所有的进程使用同样的段寄存器值时，内存管理变得更简单，也就是说它们能共享同样的线性地址。
- Linux设计的一个目标是可以把它移植到绝大多数流行的处理器平台上。然而，许多RISC 处理器支持的段功能很有限。

2.2 版的Linux 只有在 Intel 80x86 结构下才需要使用分段。尤其是，由于所有的进程使用同样的逻辑地址空间，因而需要定义的段的总数就可以被限制得很少，于是把所有的段描述符存放到全局描述符表（GDT）中就成为可能。GDT表由变量gdt所指向的数组变量gdt_table实现。如果你察看源代码索引，可以发现这些符号定义在文件 *arch/i386/kernel/head.S* 中。本书中的每一个宏、函数和其他符号都列在附录中，因此你可以在源代码中快速找到它们。

尽管系统调用允许进程创建自己的LDT，但内核并没有使用局部描述符表。事实证明这样对应用程序是很有益的，尤其是像 Wine 那样的程序，它们执行面向段的微软 Windows 应用程序。

这里是 Linux 用到的段：

- 内核代码段，在 GDT 中相应的段描述符的各个域有以下值：

  — Base = 0x00000000

  — Limit = 0xfffff

  — G（粒度标志）=1，段大小，单位以页计

  — S（系统标志）=1，正常的代码段或数据段

  — Type = 0xa，可读并可执行的代码段

  — DPL（描述符特权级）=0，内核态

  — D/B（32 位地址标志）=1，指偏移量为 32 位

  因此，与这个段相联的线性地址从 0 开始，达到 $2^{32}-1$ 这个地址限长。S 域和Type域指定这个段是一个可读和可执行代码段。它的DPL值为0，因此只能在

内核态被访问。相应的段选择符由__KERNEL_CS宏指定，为了在这个段内寻址，内核只需要把这个宏代表的值装进cs即可。

- 内核数据段，在GDT中相应的段描述符的各个域有以下值：

  — Base = 0x00000000

  — Limit = 0xfffff

  — G（粒度标志）=1，段大小，单位以页计

  — S（系统标志）=1，正常的代码段或数据段

  — Type = 2，可读写的数据段

  — DPL（描述符特权级）=0，内核态

  — D/B（32位地址标志）=1，指偏移量为32位

  除Type域以外，这个段和前一个段一样（实际上它们的线性地址空间重叠）。Type域指明了它是一个可读写的数据段，相应的段选择符由__KERNEL_DS宏定义。

- 用户态下由所有进程所共享的用户代码段，在GDT中相应的段描述符的各个域有以下值：

  — Base = 0x00000000

  — Limit = 0xfffff

  — G（粒度标志）=1，段大小，单位以页计

  — S（系统标志）=1，正常的代码段或数据段

  — Type = 0xa，可读并可执行的代码段

  — DPL（描述符特权级）=3，用户态

  — D/B（32位地址标志）=1，指偏移量为32位

  S和DPL域指明此段不是一个系统段，它的特权级=3，因此在内核态下和用户态下都可以访问它。相应的段选择符由__USER_CS宏定义。

- 用户态下由所有进程所共享的用户数据段，在GDT中相应的段描述符的各个域有以下值：

— Base = 0x00000000

— Limit = 0xfffff

— G（粒度标志）=1，段大小，单位以页计

— S（系统标志）=1，正常的代码段或数据段

— Type = 2，可读写的数据段

— DPL（描述符特权级）=3，用户态

— D/B（32位地址标志）=1，指偏移量为32位

这个段和前一个段是相同的，仅Type域不同。相应的段选择符由__USER_DS宏定义。

- 任务状态段（TSS），每一个进程有一个。这些段的描述符存放在GDT中。与每个进程相关的TSS描述符的Base域包含相应进程描述符的tss域的地址。G标志被清0。由于TSS段是236字节长，限长域被设为0xeb。类型域设为9或11（可用的32位TSS），DPL设为0，因为用户态的进程不允许访问TSS段。
- 一个通常被所有进程共享的缺省LDT段。这个段保存在default_ldt变量中。这个缺省LDT仅包含一个表项，即空段描述符。每一个进程都有自己的LDT段描述符，通常指向这个公用的缺省LDT段。Base域被设为default_ldt变量的地址，限长域设为7。如果一个进程要求一个真正的LDT，则创立一个4096字节的段（可以包含高达511个段描述符），与那个进程相关的缺省LDT段描述符就被GDT中的一个新的有明确基地址和限长域的描述符所取代。

对于每一个进程，GDT包含两个不同的段描述符，一个用于TSS段，一个用于LDT段。在GDT中可以使用的最多表项数是12+2×NR_TASKS，这里的NR_TASKS表示进程的最大个数。在前面部分，我们描述了由Linux使用的6个主要的段描述符。还有四个段描述符覆盖了高级电源管理功能（APM），GDT有四个表项空闲，因此总数为14。

如前所述，GDT可以最多有2^{13}=8192个表项，其中第一个总是为空。由于其中的14个或者没有被使用，或者由系统填充，NR_TASKS不能大于8180/2=4090。

当进程被创建时，其TSS和LDT描述符被加入到GDT中。我们将在第三章的“内核线程”一节中看到，内核本身已经产生了第一个进程：进程0，运行init_task。

内核初始化过程中，trap_init()函数把这个首进程的TSS的描述符用以下方法插进GDT中：

```
set_tss_desc(0, &init_task.tss);
```

第一个进程创建其他进程，因此每一个后来的进程都是一些现有进程的子进程。由clone()和fork()系统调用调用的copy_thread()函数用于创建新进程，copy_thread()执行同样的函数去为新进程设置TSS。

```
set_tss_desc(nr, &(task[nr]->tss));
```

由于每一个TSS描述符指向一个不同的进程，当然每个基地址域就有不同的值。copy_thread()函数也调用set_ldt_desc()函数为新进程在GDT中插入一个与缺省LDT相关的段描述符。

内核数据段包含每个进程的进程描述符。每个进程描述符包含它自己的TSS段和指向它自己的LDT段的一个指针。这个LDT段也位于内核数据段中。

如前所述，CPU的当前特权级反映了进程是在用户态还是内核态，并由存放在cs寄存器中的段描述符的RPL域指定。只要当前特权级被改变，一些段寄存器必须相应地被更新。例如，当CPL=3（用户态）时，ds寄存器必须含有用户数据段的段选择符，而当CPL=0时，ds寄存器必须含有内核数据段的选择符。

ss寄存器也有类似的情况，当CPL为3时，它必须指向一个用户数据段中的用户态栈，当CPL为0时，它必须指向内核数据段中的一个内核态栈。当从用户态切换到内核态时，Linux总是确保ss寄存器装有内核数据段的段选择符。

硬件的分页单元

分页单元（paging unit）把线性地址转换成物理地址。它把所请求的存取类型和线性地址的存取权限相比，如果这次内存存取是无效的，则产生一个页错误异常（参见第四章和第六章）。

为了效率起见，线性地址被分成以固定长度为单位的组，称为页。页内连续的线性地址被映射到连续的物理地址中。用这种方式内核可以指定物理地址和页的存取权

限，以代替它所包含的全部线性地址的存取权限。以下我们约定，术语“页”既指一系列线性地址，又指包含在这一组地址中的数据。

分页单元认为所有的RAM被分成固定长度的页框（page frame）（有时叫做物理页）。每一个页框包含一页（page），也就是说一个页框的长度与一个页的长度一致。页框是主存的一部分，因此也是一个存储区域。区分一页和一个页框是很重要的，前者只是一个数据块，可以被存放在任何页框或磁盘中。

把线性地址映射到物理地址的数据结构称为页表（page table）。页表存放在主存中，并在启用分页单元以前必须由内核对页表进行适当的初始化。

在Intel处理器中，通过设置CR0寄存器的PG标志启用分页。当PG=0时，线性地址就被解释成了物理地址。

常规分页

从i80386起，Intel处理器的分页单元处理4KB的页。32位的线性地址被分成3个域：

目录（Directory）
　　最高的10位

页表（Table）
　　中间的10位

偏移量（Offset）
　　最低的12位

线性地址的转换由两步完成，每一步都基于一种转换表，第一种转换表称为页目录表（page directory），第二种转换表称为页表（page table）。

正在使用的页目录表的物理地址存放在处理器的CR3寄存器中，线性地址内的目录域决定了它指向页目录表中的哪个项，即指向哪个页表。接下来，地址的页表域决定页表中的一项，此项含有此页所在页框的物理地址。偏移量域，决定了本页框内的相对位置（见图2-5）。由于它是一个12位长的域，故每一页含有4096字节的数据。

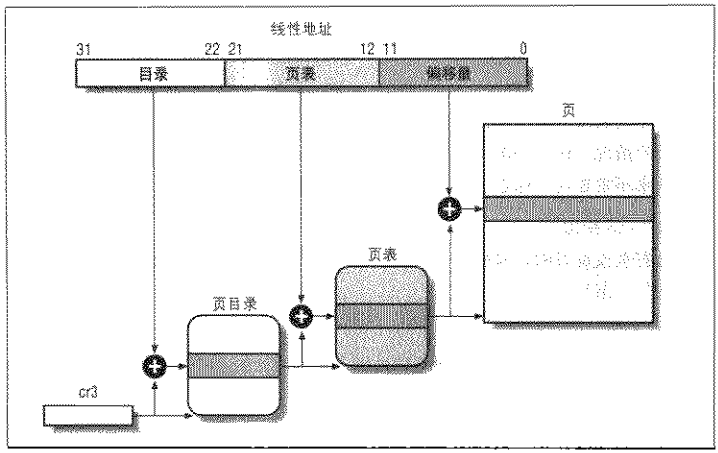


图2-5 Intel 80x86处理器的分页

目录域和页表域都是10位长，因此页目录表和页表都可以多达1024项。因此一个页目录表项可以寻址到高达1024 × 1024 × 4096=2^{32}个存储单元，和你对32位地址所希望的寻址范围一样。

页目录表项和页表项有同样的结构，每一表项都包含下面的域：

Present *标志*

如果被设为1，所指的页（或页表）就在主存中；如果为0，这一页不在主存中，此时这个表项剩余的位可由操作系统用于自己的目的（我们将在第十六章中看到Linux是如何使用这个域的）。

包含页框物理地址的最高20位的域

由于每一个页框有4KB的容量，它的物理地址必须是4096的倍数，因此物理地址的最低12位总是为0。如果这个域指向一个页目录，这个页框就含有一个页表，如果它指向一个页表，这个页框就含有一页的数据。

Accessed *标志*

分页单元每次寻找相应页框地址时，都要设置存取标志。当所选中的页被交换

出去时，这一标志就可能会被操作系统使用。分页单元从来不重置这个标志，而是必须由操作系统去做。

Dirty *标志*

只在页表项中使用。每次对一个页框进行写操作时，都要设置这个标志。和前面的情况一样，当操作系统选择一些页交换出去时会使用这个标志。分页单元从来不重置这个标志，而是必须由操作系统去做。

Read/Write *标志*

含有页或页表的存取权限（Read/Write 或只读）（参阅本章后面“硬件保护方案”一节）。

User/Supervisor *标志*

包含了访问页或页表所必须的特权级（参见后面的“硬件保护方案”一节）。

叫做 PCD 和 PWT 的两个标志

硬件高速缓存处理页或页表的控制方法（参见本章后面“硬件高速缓存”一节）。

Page Size *标志*

只适用于页目录项。如果设置为1，页目录项指的是4MB的页框（参见下一节）。

如果执行地址转换所需的页表项或页目录项中的 Present 标志为0，分页单元就把这个线性地址存放在处理器的 CR2 寄存器中，并产生 14 号异常，即“缺页”异常。

扩展分页

从“奔腾”处理器开始，Intel 80x86 微处理器引进了扩展分页，它允许页框大小为 4KB 或 4MB（参见图 2-6）。

正如前面所述，通过设置页目录表项的 Page Size 标志可以启用扩展分页功能。在这种情况下，分页单元把 32 位线性地址分成两个域：

目录域

最高 10 位

偏移量

其余 22 位

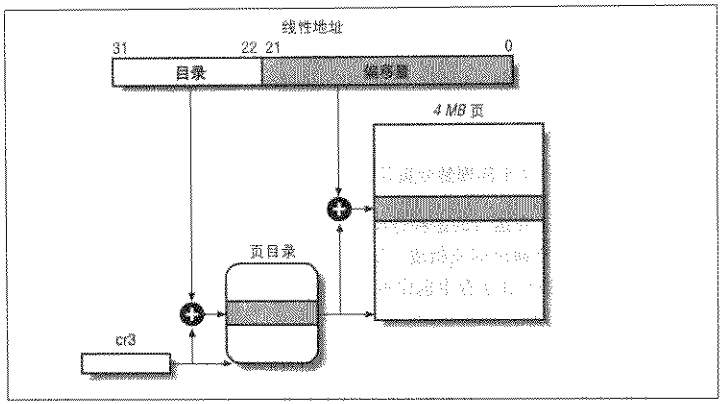


图2-6 扩展分页

扩展分页和正常分页的页目录表项基本相同，但除了：

- Page Size标志必须被设置。
- 20位物理地址域只有最高10位是有意义的。这是因为每一个物理地址都是在以4MB为边界的地方开始的，故这个地址的最低22位为0。

通过设置CR4寄存器的PSE标志能使扩展分页与常规分页共存。扩展分页用于把大段的连续线性地址转换成相应的物理地址，在这种情况下，内核可以不用中间页表而节省内存。

硬件保护方案

分页单元和分段单元的保护方案不同。Intel处理器允许分段使用四种不同的保护级别，但与页、页表相关的保护级别只有两个，因为特权级由前面“常规分页”一节中所提到的User/Supervisor标志所控制。若这个标志为0，只有当CPL小于3（对于Linux，处理器处于内核态）时才能对此页寻址；若标志为1，则总能对此页寻址。

此外，与段的三种存取权限（读，写，执行）不同的是，页的存取权限只有两种（读，

写)。如果页目录或页表项的读写标志等于0，说明相应的页表或页是只读的，否则是可读写的。

分页举例

这个简单的例子将有助于你理解分页是怎么工作的。

假如内核已给一个正在运行的进程分配的线性地址空间范围是 0x20000000 到 0x2003ffff。这个空间由64页组成。我们不必关心包含这些页的页框的物理地址，实际上，其中的一些页在主存中也许不连续。我们只关注页表项中的几个域。

让我们从分配给进程的线性地址的最高10位（分页单元解释成目录域）开始。这两个地址都以2开头，后面跟着0，因此高10位有相同的值，即0x080或十进制的128。因此，这两个地址的目录域都指向进程页目录的第129项。相应的目录项中必须包含分配给进程的页表的物理地址（参见图2-7）。如果没有给这个进程分配其它的线性地址，页目录的其余1023项都填为0。

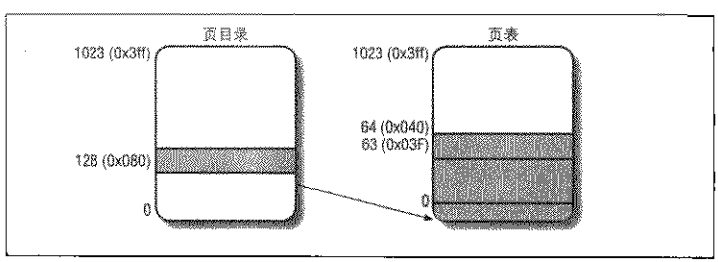


图2-7 分页的例子

中间10位的值（即页表域的值）范围从0到0x03f，或十进制的从0到63。因而只有页表的前64个表项是有意义的，其余960表项都填为0。

假设进程需要读线性地址0x20021406中的字节。这个地址由分页单元按下面的方法处理：

1. 目录域的0x80用于选择页目录的第0x80目录项，此目录项指向进程的页所在的页表。

2. 页表域的第 0x21 项用于选择页表的第 0x21 表项，此表项指向所需页的页框。

3. 最后，偏移量 0x406 用于在目标页框中读偏移量为 0x406 中的字节。

如果页表第 0x21 表项的 Present 标志为 0，此页就不在主存中。在这种情况下，分页单元在线性地址转换的同时产生一个页异常。无论何时，当进程试图访问限定在 0x20000000 到 0x2003ffff 范围之外的线性地址时，都将产生一个页异常，因为这些页表项都填充了 0，尤其是它们的 Present 标志都被清 0。

三级分页

32 位微处理器采用的是两级分页。最近几年，许多微处理器（如康柏的 Alpha，Sun 的 UltraSPARC）都采用 64 位结构的处理器。在这种情况下两级分页就不合适了，有必要升为三级分页。下面让我们用一个思维实验去看一下为什么。

开始，假想一个页在合理的情况下尽可能大（因为你必须考虑页到磁盘之间的不可避免的相互移动）。让我们选择页的大小为 16KB，由于 1KB 占用 2^{10} 地址范围，16KB 占用 2^{14} 地址范围。因此，偏移量域将是 14 位。线性地址其余的 50 位被分配给页表域和目录域。如果我们决定每一个域使用 25 位，这将意味着一个进程的页目录和页表都将含有 2^{25} 个表项，即超过 3200 万项。

即使 RAM 变得越来越便宜，我们也不能仅仅为了存放页表而浪费这么多的内存空间。

康柏的 Alpha 微处理器所选择的解决方案如下：

- 页框的大小为 8KB，因此偏移量域是 13 位。
- 只使用地址的最低 43 位。（最高 21 位总被置为 0）。
- 引入三级页表，剩余的 30 位地址被分成三个 10 位域（参见本章后面的图 2-9）。因此页表包含 $2^{10} = 1024$ 个表项，这与前面讲过的两级分页单元一样。

正如将在本章后面“Linux 的分页”一节中所看到的，Linux 设计者因受 Alpha 体系结构的影响而决定采用分页模式。

硬件高速缓存

当今的微处理器时钟频率接近GHZ，而动态RAM（DRAM）芯片的存取时间是这个时钟周期的10倍左右，这意味着，当执行的指令要求从RAM中读或向RAM中写数据时，处理器必须等待相应的时间。

为了解决二者速度不匹配的问题，引入了硬件高速缓存（cache）。硬件高速缓存基于著名的局部性原理（locality principle），用来存放程序和数据结构：由于程序的循环结构及把相关数组可以组织成线性数组，最近最常用的相邻地址在将来又被用到的可能性极大。因此，引入小而快的存储器来存放最近最常使用的代码和数据变得很有意义。为此，Intel体系结构中引入了一个叫行（line）的新单位。行由几十个连续的字节组成，以脉冲猝发模式在慢速DRAM和快速的单片静态RAM（SRAM）之间进行数据传送，用行实现硬件高速缓存。

高速缓存再被分为行的子集。在一种极端的情况下，高速缓存能够直接被映射，这时主存中的一个行总是被存放在高速缓存中完全相同的位置。在另一种极端情况下，高速缓存是完全关联的，这意味着主存中的任意一个行可以被存放在高速缓存中的任意位置。但是大多数高速缓存在某种程度上是N路组关联（N-way set associative）的，意味着主存中的任意一个行可以被存放在高速缓存N行中的任意一行中。例如，内存中的一个行可以被存放到一套2-路组关联高速缓存的两个完全不同的行中。

如图2-8所示，高速缓存单元被插入在分页单元和主存储器之间。它包含一个硬件高速缓存存储器和一个高速缓存控制器。高速缓存存储器存放内存中实际的行。高速缓存控制器有一个入口数组，每个入口对应高速缓存存储器中的一个行。每个入口有一个标签（tag）和描述高速缓存行状态的几个标志（flag）。这个标签（tag）由一些位组成，这些位允许高速缓存控制器认可由这个行当前所映射的内存单元。这种内存物理地址通常被分为以下三组：最高几位对应这个标签，中间几位对应高速缓存控制器的子集索引，最低几位对应这个行内的偏移量。

当访问一个RAM存储器单元时，CPU从物理地址中分离出子集的索引号并把子集中所有行的标签与物理地址的高几位相比较。如果发现某一个行的标签与这个物理地址的高位含有的标签相同，则CPU命中一个高速缓存（cache hit）；否则，高速缓存没有命中（cache miss）。

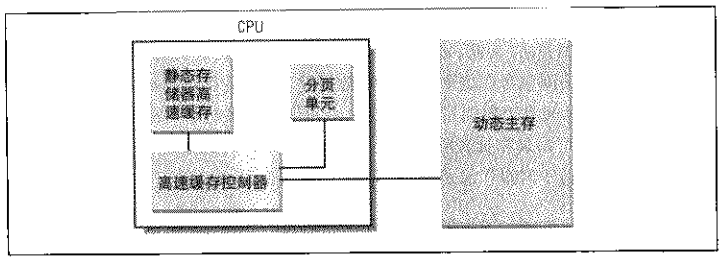


图2-8 处理器硬件高速缓存

当命中一个高速缓存时，高速缓存控制器的操作取决于存取类型。对于一个读操作，控制器从高速缓存行中选择数据并送到CPU寄存器，不需要访问内存，因此，高速缓存系统起到了其应有的作用，即节约CPU时间。对于一个写操作，控制器可能采用以下两个基本策略之一、分别称之为写透（write-through）和写回（write-back）。在写透中，控制器总是既写RAM也写高速缓存行，为了提高写操作的效率关闭高速缓存。写回方式只更新高速缓存行，不改变RAM的内容，提供了更快的功效。当然，写回结束以后，RAM最终必须被更新。只有当CPU执行一条要求刷新高速缓存项的指令时，或者当一个刷新硬件信号产生时（通常当高速缓存不命中发生后），高速缓存控制器才把高速缓存行写回到RAM中。

当高速缓存没有命中时，高速缓存行被写回到内存中，如果有必要的话，把正确的行从RAM中取出放到高速缓存的表项中。

多处理器系统的每一个处理器都有一个单独的硬件高速缓存，因此它们需要额外的硬件电路用于保持高速缓存内容的同步。参见第十一章中“对高速缓存同步的硬件支持”一节。

高速缓存技术正在快速向前发展。例如，第一代奔腾芯片包含一颗称为*L1-cache*的单片式高速缓存。近期的芯片又包含另外一个容量更大速度较慢称之为*L2-cache*的单片式高速缓存。两级高速缓存之间的一致性是由硬件实现的。Linux忽略这些硬件细节并假定只有一个单独的高速缓存。

处理器的cr0寄存器的CD标志位用来打开或关闭高速缓存电路。这个寄存器中的NW标志，指明了高速缓存是使用写透还是写回策略。

奔腾处理器高速缓存的另一个有趣的特点是，让操作系统把不同的高速缓存管理策略与每一个页框相关联。为此，每一个页目录和每一个页表的表项都包含两个标志：PCD标志指明当访问包含在这个页框中的数据时，高速缓存功能必须被打开还是被禁用。PWT标志指明当把数据写到页框时，必须使用的策略是写回策略还是写透策略。Linux清除3页目录和页表所有表项的这两个标志，其结果是：对于所有的页框启用高速缓存，对于写操作总是采用写回策略。

L1_CACHE_BYTES宏定义了奔腾处理器上高速缓存行的大小，即32字节。为了最大程度提高高速缓存的命中率，内核采用以下规则：

- 数据结构最常使用的域放在这个数据结构的低地址，以便它们能被放到高速缓存的同一行中。
- 当给大量数据结构分配空间时，内核尽力把它们都存放在内存中，以便以一种统一的方式使用所有的高速缓存行。

转换后援缓冲器（TLB）

除了通用硬件高速缓存之外，Intel 80x86处理器还包含了另外一个称之为转换后援缓冲器或TLB（Translation Lookaside Buffer）的高速缓存用于加快线性地址的转换。当一个线性地址被第一次使用时，通过访问慢速存储器中的页表，计算出相应的物理地址。同时，物理地址被存放在TLB的一项中，因此以后对同一个线性地址的引用就可以快速地得到转换。

invlpg指令用于使TLB中的一个单项失效（即释放）。为了使所有的TLB项都无效，处理器可以简单地写cr3寄存器（指向当前使用的页目录）。

由于TLB是用作存放页表内容的高速缓存，因此无论何时修改一个页表项，内核必须使TLB的相应项也无效。为了做到这一点，Linux使用flush_tlb_page(addr)函数，这个函数调用__flush_tlb_one()。后者执行invlpg汇编指令：

```
movl $addr,%eax
invlpg (%eax)
```

有时有必要使TLB所有表项都无效，如在内核初始化期间。在这样的情况下，内核调用__flush_tlb()函数，把cr3的当前值重新写进去：

```
movl %cr3, %eax
movl %eax, %cr3
```

Linux 的分页

正像我们在“三级分页”一节中所解释的，Linux 采用三级分页模式，这样的分页在 64 位的系统上也是可行的。图 2-9 展示了这个模式，它定义了三种类型的页表：

- 页全局目录（Page Global Directory）
- 页中间目录（Page Middle Directory）
- 页表（Page Table）

页全局目录包含了许多页中间目录的地址，而页中间目录又包含了许多页表的地址。每一个页表项指向一个页框。因此线性地址被分为四个部分。图 2-9 没有显示位数，因为每一部分的大小与具体的计算机体系结构有关。

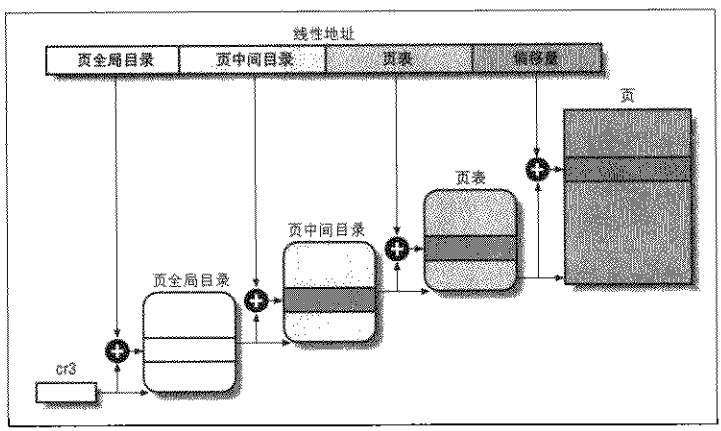


图 2-9 Linux 分页模式

Linux 对进程的处理很大程度上依赖于分页。实际上，线性地址到物理地址的自动转换使下面的设计目标变得可行：

- 给每一个进程分配一块不同的物理地址空间，这确保了对寻址错误能提供有效的保护。
- 区别页（即一组数据）和页框（即主存中的物理地址）之不同。这就允许同一页可以被存储在一个页框中，然后又被保存到磁盘上，以后重新装入该页时又可以被装在不同的页框中。这就是虚拟存储器机制的基本因素（参见第十六章）。

正如我们将在第七章看到的，每一个进程有它自己的页全局目录和自己的页表集合。当进程切换发生时（参见第三章中的“进程切换”一节），Linux把cr3控制寄存器的内容保存在一个TSS段中，然后把另外一个TSS段中的新值装入cr3寄存器中。因此，当新进程恢复在CPU上的执行时，分页单元将引用一组正确的页表。

当三级分页模式应用到只有两级页表的奔腾处理器上时会发生什么情况？Linux通过让“页中间目录”位全为0，彻底取消了中间页表域。不过，页中间目录在指针序列中的位置被保留下来，以便于同样的代码在32位系统和64位系统下都能使用。内核为页中间目录保留了一个位置，这是通过把它的页目录项数设置为1，并把这个目录项映射到页全局目录一个适当的目录项而实现的。

把逻辑地址映射为线性地址虽然有点复杂，但现在已经成了一种机械式的任务。本章下面的几节中列举了一些比较单调乏味的函数和宏，它们检索内核查找地址和管理表格所需的信息；其中大多数函数只有一二行。也许现在你就想跳过这部分，但是知道这些函数和宏的功能是非常有用的，因为后面几章的讨论中你会经常看到它们。

线性地址域

以下的宏简化了页表处理：

PAGE_SHIFT

这个宏指明偏移量域的位数。当用于奔腾处理器时，它的值为12。由于一页内的所有地址都必须放到这个偏移量域，Intel 80x86系统的页的大小是2^{12}=4096字节。PAGE_SHIFT的值为12可以看作以2为底的页的大小的对数。PAGE_SIZE使用这个宏来返回页的大小。最后，宏PAGE_MASK的值定为0xfffff000，用以屏蔽掉偏移量域的所有位。

PMD_SHIFT

决定由第二级页表映射的地址的位数。它的值定为22（12位的偏移量加上10位的页表）。宏PMD_SIZE用于计算由页中间目录的一个单独表项所映射的区域的大小，也就是说，一个页表的大小。因此，PMD_SIZE的值为2^{22}，即4MB。PMD_MASK的值为0xffc00000，用于屏蔽偏移量域与页表域的所有位。

PGDIR_SHIFT

决定第一级页表能映射的区域大小的对数。由于中间目录域长度为零，这个宏和宏PMD_SHIFT有一样的值，即22。宏PGDIR_SIZE用于计算页全局目录的一个单独表项所能映射区域的大小，也就是说，一个页目录的大小。宏PGDIR_SIZE的值为4MB。宏PGDIR_MASK的值为0xffc00000，作用和PMD_MASK一样。

PTRS_PER_PTE，PTRS_PER_PMD，和PTRS_PER_PGD

用于计算页表、页中间目录和页全局目录表项的个数，它们的值分别为1024，1和1024。

页表的处理

pte_t，pmd_t和pgd_t是32位的数据类型，分别描述页表、页中间目录和页全局目录的一个表项。pgprot_t也是一个32位的数据类型，描述与一个单独表项相关的保护标志。

四个类型转换宏（pte_val()、pmd_val()、pgd_val()和pgprot_val()）把一个32位的无符号整数转换成所需的类型。另外的四个类型转换宏（__pte()，__pmd()，__pgd()，和__pgprot()）执行相反的转换，把上面提到的四种特殊的类型转换成一个32位的无符号整数。

内核还提供了许多宏和函数用于读或修改页表表项：

- 如果相应的表项的值为0，那么，宏pte_none()、pmd_none()和pgd_none()给出的值为1，否则值为0。
- 如果相应表项的Present标志位的值为1（即相应页或页表已经被装入主存），那么，宏pte_present()、pmd_present()和pgd_present()给出的值为1。

- 宏pte_clear()、pmd_clear()和pgd_clear()清相应页表的表项。

函数使用宏pmd_bad()和pgd_bad()来检查页全局目录和页中间目录的目录项。如果目录项指向一个不能使用的页表，也就是说，如果至少出现以下条件中的一个，则这两个宏的值为1：

- 页不在主存中（Present标志被清除）。
- 页是只读页（Read/Write标志被清除）。
- Acessed或者Dirty位被清除（对于每个现有的页表，Linux总是强制这些位置1）。

没有定义pte_bad()宏，因为对一个不在主存中的页，一个不可写的页，或一个根本无法访问的页，页表项对这样页的引用都是合法的。取代这个宏的是几个函数，用来查询页表表项任意一个标志的当前值：

pte_read()
: 返回User/Supervisor标志的值（指出此页在用户态下是否可以被访问）。

pte_write()
: 如果Present标志和Read/Write标志都被设置，则返回1（指出这一页是否存在且可写）。

pte_exec()
: 返回User/Supervisor标志的值（指出此页是否可以在用户态下被访问）。请注意Intel处理器上的页框不提供代码执行保护。

pte_dirty()
: 返回Dirty标志的值（指出这一页是否被改动过）。

pte_young()
: 返回Accessed标志的值（指出这一页是否被存取过）。

另一组函数用于设置页表表项的标志值：

pte_wrprotect()
: 清除Read/Write标志。

pte_rdprotect和pte_exprotect()

清除User/Supervisor标志。

pte_mkwrite()

设置Read/Write标志。

pte_mkread()和pte_mkexec()

设置User/Supervisor标志。

pte_mkdirty()和pte_mkclean()

把Dirty标志分别置为1或0，因而把此页标记为修改过或没修改。

pte_mkyoung()和pte_mkold()

把Accessed标志置为1或者0，因而把此页标记为访问过（young）或者没访问（old）。

pte_modify(p,v)

把页表表项p的所有存取权限设置为指定的值v。

set_pte

把一个特定的值写进一个页表表项中。

下面的宏把一个页地址和一组保护标志组合成一个32位的页表表项，或者执行相反的操作，从一个页表表项中解析出页地址和保护信息。

mk_pte()

合并一个线性地址和一组存取权限以创建一个32位的页表表项。

mk_pte_phys

通过合并此页的物理地址和存取权限创建一个页表表项。

pte_page()和pmd_page()

从页表表项返回一个页的线性地址，从页中间目录目录项返回一个页表的线性地址。

pgd_offset(p,a)

输入参数为内存描述符p（参见第六章）和线性地址a。这个宏产生一个对应于地址a的一个全局目录表项的地址，页全局目录通过内存描述符p内的一个指针找到。宏pgd_offset_k(o)也很类似，不同之处在于它指向由内核线程所使用的内存描述符（参见第三章的“内核线程”一节）。

pmd_offset(p,a)

输入参数为页全局目录目录项p和线性地址a。它产生由p所指向的页中间目录内的地址a所对应的目录项的地址。宏pte_offset(p,a)也很类似，但p是一个页中间目录项，它产生由指针p所指向的页表内的地址a对应的表项的地址。

这个既长又枯燥的列表中的最后一组函数，用于简化页表表项的创建和删除。当使用两级页表时，创建或删除一个页中间目录表项是不必要的。如本节前部分所述，页中间目录仅含有一个指向下属页表的目录项。所以，页中间目录目录项只是页全局目录中的一项而已。然而当处理页表时，创建一个页表项可能很复杂，因为包含页表项的页表可能就不存在。在这种情况下，必须分配一个新的页框，把它填写为0，并最终加入这个表项。

每个页表存放在一个页框中，更进一步讲，每个进程会使用许多的页表。我们将在第六章的“页框管理”一节中看到，页框的分配和删除都是耗时的操作。因此，当内核撤消一个页表时，把相应的页框加到一个软高速缓存中。当内核必须分配一个新的页表时，就从这个高速缓存中取出一个页框。只有当这个高速缓存为空时，才从内存分配器那里申请一个新的页框。

页表高速缓存是页框的一个简单链表。宏pte_quicklist指向这个链表的头，而链表中每个页框的前四个字节用作指向下一个元素的指针。页全局目录高速缓存类似，但这个链表的表头由另外一个宏pgd_quicklist给出。当然，Intel体系结构并没有页中间目录高速缓存。

由于没有限制页表高速缓存的大小，内核必须实行一种机制防止它们变得过大。因此，内核引入了上限下限范围标志，它们被存放在pgt_cache_water数组中。check_pgt_cache()函数检查每一个高速缓存的大小是否高于范围限制的上限，如果超过，删除一些页框直到高速缓存的大小达到范围限制的下限。当系统空闲时或者内核释放某个进程的所有页表时就调用check_pgt_cache()函数。

现在介绍最后一组函数和宏：

pgd_alloc()

通过调用函数get_pgd_fast()来分配一个新的页全局目录，此函数从页全局目录高速缓存中取得一个页框。如果高速缓存为空，通过调用函数get_pgd_slow()分配一个新的页框。

pmd_alloc(p,a)

定义这个宏以使三级分页系统可以为线性地址a分配一个新的页中间目录。在Intel 80x86系统上，这个函数只是返回输入参数p的值．也就是说，返回页全局目录目录项的地址。

pte_alloc(p,a)

输入参数为页中间目录目录项的地址p和线性地址a，返回a相应的页表表项的地址。如果页中间目录目录项为空，这个函数必须分配一个新的页表。为了完成这项功能．通过调用函数get_pte_fast()从页表高速缓存中查找一个空闲的页框。如果高速缓存为空．通过调用函数get_pte_slow()分配一个页框。如果分配了一个新的页表，相应的表项被初始化，User/Supervisor标志被设置。pte_alloc_kernel()有点类似，不同之处在于，它调用函数get_pte_kernel_slow()分配一个新的页框而不是调用函数get_pte_slow()。get_pte_kernel_slow()函数清除新页表的User/Supervisor标志。

pte_free()、pte_free_kernel()和pgd_free()

释放一个页表并把这个空闲的页框插入到高速缓存合适的地方。pmd_free()和pmd_free_kernel()函数什么也不做，因为页中间目录并不真正存在于Intel 80x86系统上。

free_one_pmd()

调用pte_free()函数以释放一个页表。

free_one_pgd()

释放一个页中间目录中的所有页表。在Intel结构中，它只调用free_one_pmd()函数一次，然后通过调用pmd_free()函数来释放页中间目录。

SET_PAGE_DIR

设置一个进程的页全局目录。通过向一个进程的任务状态段（TSS）的一个域中放置页全局目录物理地址来完成这项操作。每次进程开始或恢复在CPU上的执行时，这个地址就被装入cr3寄存器中。当然，如果受影响的进程正在被执行，这个宏也直接改变cr3寄存器的值，因此这个改变马上生效。

new_page_tables()

分配页全局目录及建立进程的地址空间必需的所有页表。为了把新的页全局目录分配给这个进程，也调用SET_PAGE_DIR宏。这个话题将在第七章中详细讨论。

clear_page_tables()

通过反复调用 free_one_pgd()函数来清除一个进程页表的内容。

free_page_tables()

与 clear_page_tables()函数非常类似，但是它也释放进程的页全局目录。

保留的页框

内核代码和数据结构存放在一组保留的页框中。这些页框所含的页从不被动态地分配或者被交换到磁盘上。

作为一条常规，Linux内核被安装在RAM物理地址0x00100000开始的地方，也就是说，从第二个MB开始。所需的页框总数依赖于内核的配置方案：典型的配置所得到的内核可以被安装在小于2MB的RAM中。

为什么内核没有被安装在RAM第一个MB开始的地方？因为PC机结构有几个独特的地方必须考虑到：

- 页框0由BIOS使用，存放加电自测期间检查到的硬件配置。
- 物理地址从0x000a0000到0x000fffff的范围被留作BIOS程序使用，并且映射ISA显示卡上的存储器（在早期的MS-DOS系统上有很有名的640KB地址限制的说法）。
- 前1MB的其他页框可能被保留，用作特定的计算机模式。例如，IBM ThinkPad把0x0a页框映射到0x9f页框。

为了避免把内核装入一组不连续的页框里，Linux更愿跳过第1MB的RAM。更明确地说，Linux用PC体系结构未保留的页框来存放动态分配的页。

图2-10显示Linux怎样填满前2MB的RAM。我们假设内核需要小于1MB的RAM（比较乐观的情形）。

符号_text对应于物理地址0x00100000，表示内核代码第一个字节的地址。内核代码的结束位置由另外一个类似的符号_etext所表示。内核数据被分为两组：初始化过的数据的和没有初始化的数据。初始化过的数据在_etext后开始，在_edata处结束。再后面是没有初始化的数据并以_end结束。

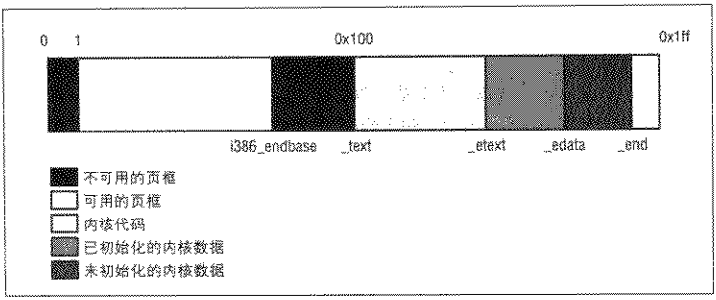


图2-10 Linux 2.2的前512个页框（2MB）

图中出现的符号并没有在源代码中定义，它们是编译内核时产生的（注2）。

给BIOS或硬件设备所保留的第一个物理地址对应的线性地址被保存在i386_endbase变量中（通常的值是0x0009f000）。在多数情况下，BIOS在加电自测时给这个变量写入一个初值。

进程页表

一个进程的线性地址空间被分成两部分：

- 从0x00000000到PAGE_OFFSET-1的线性地址，无论用户态还是内核态的进程都可以寻址。
- 从PAGE_OFFSET到0xffffffff的线性地址，只有内核态的进程才能寻址。

通常，宏PAGE_OFFSET的值是0xc0000000，这意味着线性地址的第四个GB保留给内核，而前3GB可以供内核和用户程序同时访问。当然，PAGE_OFFSET的值可以在编译时被用户重新设置。实际上，正如我们将在下一节所看到的，保留给内核的线性地址范围必须能够包含系统所安装的所有物理RAM的映射。此外，我们将在第七章看到，内核也利用这个范围内的线性地址把不连续的页框反映射为连续的

注2：你可以在*System.map*文件中找到这些符号的线性地址，*System.map*是编译内核以后所创建的。

线性地址。因此，如果Linux必须被装到容量大的RAM机器上，很可能有必要对线性地址实行一种不同的排列。

页全局目录的第一部分表项（通常是前768项）映射低于PAGE_OFFSET的线性地址，这取决于特定的进程。相反，剩余的表项对于所有的进程来说都是一样的，它们等于页全局目录swapper_pg_dir对应的目录项（参见下一节）。

内核页表

下面描述内核怎样初始化自己的页表，这个过程分为两个阶段。实际上，内核映像刚刚被装入内存之后，CPU仍然运行于实模式，所以分页功能没有被启动。

第一个阶段，内核仅创建足以把自己装入到RAM中的4MB地址空间。

第二个阶段，内核充分利用剩余的RAM并适当地建立分页页表。下一部分解释这个方案是怎样实施的。

临时内核页表

页全局目录和页表都是在内核编译过程中被静态地初始化。我们不再过多提及页中间目录，因为它等于页全局目录的表项。

页全局目录放在变量swapper_pg_dir中，映射前4MB RAM的页表放在pg0变量中。

分页第一阶段的目标是在实模式和保护模式下都能很容易地对前4MB进行寻址。因此，内核必须创建一个映射把以下两组线性地址都映射到物理地址0x00000000至0x003fffff的范围上，这两组地址是：线性地址0x00000000至0x003fffff，线性地址PAGE_OFFSET至PAGE_OFFSET+0x3fffff。换句话说，在初始化的第一个阶段，无论使用与物理地址一样的线性地址还是使用从PAGE_OFFSET开始的4M有价值的线性地址，内核都能对前4MB RAM（0x00000000至0x003fffff）寻址。

假定PAGE_OFFSET等于0xc0000000，内核通过把swapper_pg_dir的所有目录项都填为0来创建所要的映射，不过，第0项到第0x300项（十进制768）除外。后者的目录项横跨了从0xc0000000到0xc03fffff之内的所有线性地址。第0项到第0x300目录项按下面的方式初始化：

- 地址域设为pg0的地址。
- Present、Read/Write及User/Supervisor标志被设置。
- Accessed、Dirty、PCD、PWD及Page Size标志被清0。

唯一的pg0页表也被静态地初始化，因此第i个表项对第i个页框寻址。

汇编语言函数startup_32()开启分页功能。通过向cr3控制寄存器装入swapper_pg_dir的地址及设置cr0控制寄存器的PG标志来达到这一目的。如以下摘录所示：

```
movl $0x101000,%eax
movl %eax,%cr3          /* 设置页表指针.. */
movl %cr0,%eax
orl $0x80000000,%eax
movl %eax,%cr0          /* ..设置允许分页(PG)位 */
```

最终的内核页表

由内核页表所提供的最终的映射必须把从PAGE_OFFSET开始的线性地址转化为从0开始的物理地址。

宏_pa用于把从PAGE_OFFSET开始的线性地址转换成相应的物理地址，而宏_va做相反的转化。

最终的内核页全局目录仍然保留在变量swapper_pg_dir中。它由函数paging_init()初始化。这个函数需要两个输入参数：

start_mem

紧跟在内核代码段和数据段区域之后的RAM的第一个字节的线性地址。

end_mem

内存结束处的线性地址（由BIOS程序在加电自测阶段计算出来的地址）。

Linux充分挖掘奔腾处理器的扩展分页功能，启用4MB的页框。它绕过内核页表（注3），允许从PAGE_OFFSET到物理地址进行非常高效的映射。

注3：我们会在第六章的"非连续内存区管理"一节中看到，内核可以为它所用的4KB的页设置另外的映射，此时，它才使用页表。

通过如下等价的循环，页全局目录 swapper_pg_dir 被再次初始化：

```
address = 0;
pg_dir = swapper_pg_dir;
pgd_val(pg_dir[0]) = 0;
pg_dir += (PAGE_OFFSET >> PGDIR_SHIFT);
while (address < end_mem) {
    pgd_val(*pg_dir) = _PAGE_PRESENT+_PAGE_RW+_PAGE_ACCESSED
          +_PAGE_DIRTY +_PAGE_4M+__pa(address);
    pg_dir++;
    address += 0x400000;
}
```

可以看到，页全局目录的第一个目录项被置为0，因此去掉前4MB线性地址和物理地址之间的映射。第一个页表因而开始生效，因此用户态的进程也可以使用介于0和4194303之间的线性地址范围。

页全局目录中引用高于PAGE_OFFSET线性地址的所有目录项的User/Supervisor标志被清0，因此，拒绝用户态的进程访问内核的地址空间。

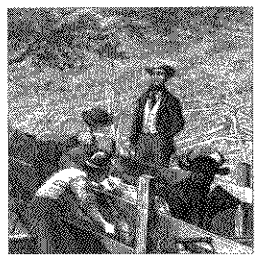
一旦 swapper_pg_dir 被初始化过，就不再使用临时页表 pg0。

对 Linux 2.4 的展望

Linux 2.4 做了两点主要的改动。与所有现有的进程相关的 TSS 段描述符，不再保存在全局描述符表中，这项改变去除了对现有进程数目的硬编码限制，这个限制因而成为可用 PID 数目。简而言之，在内核代码中将再找不到 NR_TASKS 宏了，所有依赖这个宏的数据结构都被删除或者替换掉。

另一个主要的改动和物理存储器的寻址有关。比较新的Intel 80x86处理器包含一种称之为物理地址扩展（PAE）的功能，它给标准的32位物理地址外加了4位地址。Linux 2.4利用PAE的优势支持高达64GB的RAM。线性地址仍然保持32位。因此仅有4GB的RAM可以“持久性地被映射”，并且在任一时刻被访问。因而Linux 2.4认可三个不同部分的RAM：适用于ISA直接内存存取（DMA）的物理内存，不适用于ISA直接内存存取但被内核永久映射的物理内存，以及高端内存，也就是没有被内核永久映射的物理内存。

第三章
进程



进程是任何多道程序设计的操作系统中的基本概念。进程通常被定义为程序执行时的一个实例，例如，如果16个用户同时运行vi，那么，就有16个独立的进程（尽管它们共享同一个可执行代码）。在Linux源代码中，进程常被称为“任务”。

在这一章，我们将首先讨论进程的静态特性，然后描述内核如何进行进程切换。最后两节研究进程的动态特性，即如何创建和撤消进程。这一章还将讲述Linux对多线程应用的支持，正如第一章中所提到的，它依赖所谓的轻量级进程（LWP）。

进程描述符

为了管理进程，内核必须对每个进程所做的事情进行清楚的描述。例如，内核必须知道进程的优先级，它是正在CPU上运行还是因某些事件被阻塞，给它分配什么样的地址空间，允许它访问哪个文件等等。这正是进程描述符（process descriptor）的作用。进程描述符都是task_struct类型结构，它的域包含了与一个进程相关的所有信息。因为进程描述符中存放了那么多信息，所以它是相当复杂的。它本身不仅包含了很多域，而且一些域还包括了指向其他数据结构的指针，依此类推。图3-1示意性地描述了Linux的进程描述符。

在图右边的五个数据结构涉及进程所拥有的特殊资源，这些资源将在以后的章节中涉及到。这一章将集中讨论两个域，进程的状态和进程的父/子间关系。

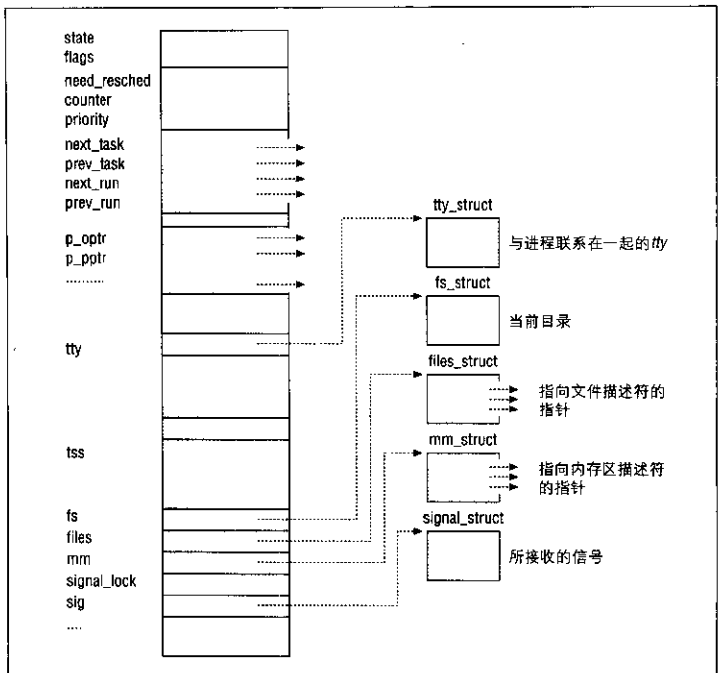


图 3-1 Linux 进程描述符

进程状态

顾名思义，进程描述符中的状态域描述了进程当前所处的状态，它是一组标志，其中，每一个标志描述了一种可能的进程状态。在当前的Linux版本中，这些状态是互斥的，因此，严格意义上说，只能设置一种状态；其余的标志将被清除。下面是进程可能的状态：

可运行状态（TASK_RUNNING）

进程要么在CPU上执行，要么准备执行。

可中断的等待状态（TASK_INTERRUPTIBLE）

进程被挂起（睡眠），直到一些条件变为真，这些条件包括：产生一个硬件中断，释放进程正等待的系统资源，或传递一个信号，它们都能唤醒进程，即让进程的状态回到TASK_RUNNING。

不可中断的等待状态（TASK_UNINTERRUPTIBLE）

与前一个状态类似，但有一个例外，把信号传递到睡眠的进程不能改变它的状态。这种状态很少用到，但在一些特定的情况下这种状态是很有用的：进程必须等待，不能被中断，直到给定的事件发生。例如，当进程打开一个设备文件，其相应的设备驱动程序开始探测相应的硬件设备时会用到这种状态，探测完成以前，设备驱动程序不能被中断，否则，硬件设备会处于不可预知的状态。

暂停状态（TASK_STOPPED）

进程的执行被暂停。当进程接收到SIGSTOP、SIGTSTP、SIGTTIN或SIGTTOU信号后，进入暂停状态。当一个进程被另一个进程监控时［例如debugger执行ptrace()系统调用监控一个测试程序］，任何信号都可以把这个进程置于TASK_STOPPED状态。

僵死状态（TASK_ZOMBIE）

进程的执行被终止，但是，父进程还没有发布wait()类系统调用［wait()，wait3()，wait4()或waitpid()］以返回有关死进程的信息。发布wait()类系统调用前，内核不能丢弃包含在死进程描述符中的数据，因为父进程可能还需要它。（参见本章结尾的“删除进程”一节）。

标识一个进程

尽管Linux进程能共享其内核的大部分数据结构（通过一种有效的方法：轻量级进程），但每个进程还是拥有它自己的进程描述符。可以被独立调度的每一个执行上下文必须拥有它自己的进程描述符。

不应该把轻量级进程与用户态的线程混淆起来，后者是由用户级的库函数所处理的不同的执行流。例如，旧的Linux系统依靠pthread库完全在用户空间实现了POSIX线程（thread）。因此，多线程的程序是作为单独的Linux进程来执行的。目前，*pthread*库已被合并到标准的C库中，并吸收了轻量级进程的优点。

进程和进程描述符之间有非常严格的一一对应关系，这使得用32位进程描述符地址（注1）标识进程成为一种方便的工具。进程描述符指针指向这些地址，内核对进程的大部分引用是通过进程描述符指针进行的。

另一方面，任何类Unix操作系统允许用户使用一个叫做进程标识符（Process ID，PID）的数来标识进程。PID是32位的无符号整数，存放在进程描述符的pid域中。PID被顺序编号，新创建进程的PID通常是前一个进程的PID加1。然而，为了与16位硬件平台的传统Unix系统保持兼容，在Linux上允许的最大PID号是32767。当内核在系统中创建第32768个进程时，就必须重新开始使用已闲置的PID号。

在本节的结尾，我们将向你说明如何从进程的PID中有效地导出它的描述符指针。效率是非常重要的，因为很多如kill()这样的系统调用用PID表示受影响的进程。

任务数组

进程是动态实体，它在系统中的生命周期可以从几秒到几个月，因此，内核必须能同时处理很多进程。事实上，我们从前面的章节中知道，Linux能处理多达NR_TASKS个进程。内核在它自己的地址空间保存了一个全局静态数组task，其大小为NR_TASKS。数组中的元素就是进程描述符指针，空指针表示数组项中没有进程描述符。

进程描述符的存放

task数组仅仅包含了进程描述符的指针，而不是描述符本身。因为进程是动态实体，因此，进程描述符被存放在动态内存中，而不是存放在永久性地分配给内核的内存区。Linux把每个进程的两个不同的数据结构存放在一个单独8KB的内存区：进程描述符和内核态的进程栈。

在第二章“Linux中的段”一节中我们已经知道，内核态的进程访问包含在内核数据段中的栈，这个栈不同于用户态的进程所用的栈。因为内核控制路径所用的栈很

注1：从技术上说，这32位仅仅是一个逻辑地址偏移量的组成部分，因为Linux利用单独的内核数据段，我们可以把这个偏移量看作与这个逻辑地址是等价的。更进一步地说，因为代码段和数据段的基地址被置为0，因此我们可以把这个偏移量当作一个线性地址来对待。

少(即使考虑到多个内核控制路径代表同一进程交错执行),也只需要几千个字节的内核态堆栈。因此,对栈和进程描述符来说,8KB 足够了。

图 3-2 显示了在内存区中存放两种数据结构的方式。进程描述符起始于这个内存区的开始,而栈起始于末端。

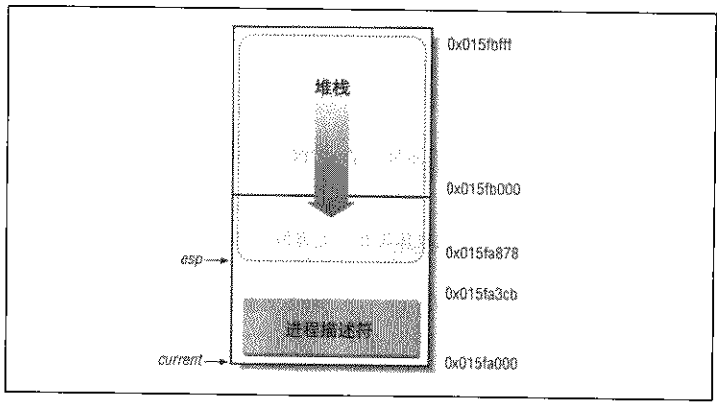


图 3-2 进程描述符和进程内核态堆栈的存放

esp 寄存器是 CPU 栈指针,用来存放栈顶的地址。在 Intel 系统中,栈起始于末端,并朝这个内存区开始的方向增长。从用户态刚切换到内核态以后,进程的内核态堆栈总是空的,因此,esp 寄存器直接指向这个内存区的顶端。

C 语言允许使用下列的联合结构来方便地表示这样一个混合结构:

```
union task_union {
    struct task_struct task;
    unsigned long stack[2048];
};
```

在图 3-2 中,从用户态转换到内核态以后,esp 寄存器包含的地址为 0x015fc000。进程描述符存放在从 0x015fa000 开始的地址。只要把数据写进栈中,esp 的值就递减。因为进程描述符的长度小于 1000 字节,因此内核态堆栈可以扩展到 7200 字节。

current宏

从效率的观点来看，刚才所讲的进程描述符与内核态堆栈之间的配对，其主要好处是：内核很容易从esp寄存器的值获得当前在CPU上正在运行的进程描述符指针。事实上，因为这个内存区是8KB（2^{13}字节）长，内核必须做的事是让esp至少有13位有效位，以获得进程描述符的基地址，这项工作由current宏来完成，它产生如下一些汇编指令：

```
movl $0xfffffe000, %ecx
andl %esp, %ecx
movl %ecx, p
```

执行了这三条指令以后，局部变量p包含了在CPU上运行的进程的进程描述符指针（注2）。

用栈存放进程描述符的另一个优点体现在多处理器系统上：对于每个硬件处理器，仅仅通过检查前面所示的栈就可以得出当前正确的进程。Linux 2.0没有把内核态堆栈与进程描述符存放在一起，而是引入了全局静态变量current来标识正在运行进程的进程描述符。在多处理器系统上，有必要把current定义为一个数组，每一个元素对应一个可用CPU。

current宏经常作为进程描述符域的前缀出现在内核代码中，例如，current->pid返回在CPU上正在执行的进程的PID。

由EXTRA_TASK_STRUCT（这里，把这个宏通常设置为16）个内存区组成的小型高速缓存用来避免无谓的内存分配器调用。为了理解设置这个高速缓存的目的，举一个例子。假设刚撤消某个进程，才过一会儿，又创建了一个新的进程，如果没有高速缓存，内核就必须先把8KB的内存区释放给内存分配器，紧接着，又请求另外同样大小的内存区。这是内存高速缓存的一个例子，是绕过内核内存分配器而引入的一种软件机制。在其他章节你还会发现很多内存高速缓存的例子。

task_struct_stack数组包含高速缓存中指向进程描述符的指针。这个数组名字来自于进程描述符的释放和请求，是通过对数组分别进行“push”及“pop”操作而实现的：

注2：共享存储方法的一个缺点是，为提高效率起见，内核将8KB内存区的内容存储在两个连续的页框中，第一个页框的大小是2^{13}（即8KB）的倍数。当可用动态内存空间很少时，这也许会成为一个问题。

free_task_struct()

这个函数释放8KB的task_union内存区，如果高速缓存未满，并把它们放进高速缓存中。

alloc_task_struct()

这个函数分配8KB的task_union内存区。如果高速缓存至少有一半填满，或者如果没有一对空闲连续可用的页框时，这个函数从高速缓存中获得内存区。

进程链表

为了对给定类型的进程（例如，在可运行状态的所有进程）进行有效的搜索，内核建立了几个进程链表。每个进程链表由指向进程描述符的指针组成。进程描述符的数据结构中包含了一个链表指针（即每个进程用来指向下一个进程的域）。当你查看task_struct结构中C语言的声明时，你就会发现这个描述符似乎以复杂递归的方式又拐到自己那儿去了。不过，概念没有链表复杂，这个链表是一种数据结构，它包含的指针指向它自己的下一个实例。

一个双向循环链表（参见图3-3）把所有现有的进程联系起来，我们叫它为进程链表(process list)。每个进程的prev_task和next_task域用来实现链表。链表的头是init_task描述符，由task数组的第一个元素指向，它是所有进程的祖先，把它叫做进程0（process 0）或*swapper*（参见本章的“内核线程”一节）。init_task的rev_task域指向链表中最后插入的进程描述符。

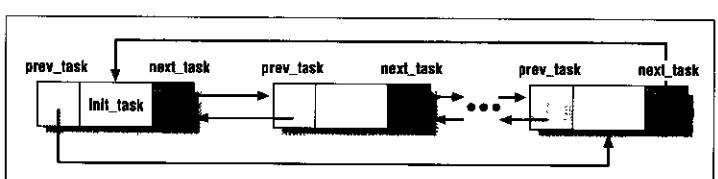


图3-3 进程链表

SET_LINKS和REMOVE_LINKS宏用来分别在进程链表中插入和删除一个进程描述符。这两个宏也考虑了进程之间的亲属关系（参见本章的“进程之间的亲属关系”一节）。

另一个叫 for_each_task的宏非常有用，它扫描整个进程链表，定义如下：

```
#define for_each_task(p) \
    for (p = &init_task ; (p = p->next_task) != &init_task ; )
```

这个宏是循环控制语句，注意init_task进程描述符是如何起到链表头的作用的。这个宏首先从 init_task移到下一个任务，并继续直到又到 init_task为止（使用循环链表的优点）。

TASK_RUNNING 状态的进程链表

当内核要寻找一个新的进程在 CPU 上运行时，必须只考虑可运行进程（即处在TASK_RUNNING状态的进程），因为扫描整个进程链表是相当低效的，所以引入了可运行状态进程的双向循环链表，也叫运行队列（runqueue）。进程描述符包含的next_run及prev_run域可以用来实现可运行队列。与前面的情况一样，init_task进程描述符起链表头的作用，nr_running变量存放可运行队列中进程的总个数。

add_to_runqueue()函数把一个进程描述符插入到链表的开始，del_from_runqueue()从队列中删除一个进程描述符。为了调度，提供了两个函数，move_first_runqueue()和move_last_runqueue()，把进程描述符分别移到运行队列的开始或末尾。

最后，利用wake_up_process()函数使一个进程可运行，这个函数把进程的状态设置为 TASK_RUNNING，调用 add_to_runqueue()把进程插入到运行队列的链表中，并给 nr_running 加 1。当这个进程是实时进程，或其动态优先级大于当前进程时，该函数就强迫调用调度程序（scheduler）（参见第十章）。

pidhash 表及链接表

在几种情况下，内核必须能从进程的 PID 导出对应的进程描述符指针。例如，为kill()系统调用提供服务：当进程 P1 希望向另一个进程 P2 发送一个信号时，P1调用 kill()系统调用，其参数为 P2 的 PID，内核从这个 PID 导出其对应的进程描述符，然后从 P2 的进程描述符中取出指向记录有挂起信号的数据结构指针。

顺序扫描进程链表并检查进程描述符的pid域，这种方法可行但效率相当低。为了加速查找，引入了pidhash散列表，它由 PIDHASH_SZ 个元素组成（PIDHASH_SZ

通常设置为NR_TASKS/4)。表项包含进程描述符指针。用pid_hashfn宏把PID转换成表的索引:

```
#define pid_hashfn(x) \
    ((((x) >> 8) ^ (x)) & (PIDHASH_SZ - 1))
```

正如计算机基础科学课程所解释的那样，散列（hash）函数并不总能确保PID与表的索引一一对应。两个不同的PID散列到相同的表索引称为冲突（colliding）。

Linux利用链接表（chained list）来处理冲突的PID：每一个表项是由冲突的进程描述符组成的双向链表。这些链表是由进程描述符中的pidhash_next和pidhash_pprev域来实现的。图3-4显示了有两个表的pidhash表：进程号（PID）为228和27535的两个进程散列到这个表的第101个元素，而进程号（PID）为27536的进程散列到这个表的第124个元素。

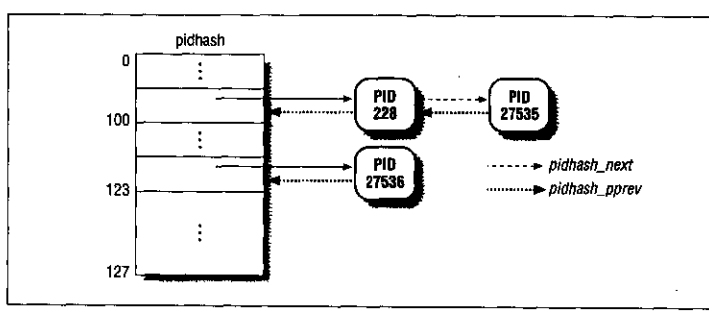


图3-4 pidhash表及链接表

具有链接表的散列法比PID到表索引的线性转换更优越，这是因为可以假设PID的值从0到32767。因为NR_TASKS是进程的最大号，但通常设置为512，因此，把表定义为32768项将会是一种存储的浪费。

调用hash_pid()和unhash_pid()函数在pidhash表中分别插入和删除一个进程。find_task_by_pid()函数查找散列表并返回给定PID进程的进程描述符指针（如果没找到则返回一个空指针）。

task 空闲表项的链表

每当一个进程被创建或撤消时，就要更新task数组，与前面所说明的其他链表一样，这里用到的链表是为了加速增加和删除。把一个新项有效地加到数组中去的方式，不是线性地搜索数组并寻找第一个空闲项，而是内核维持了一个独立的、包含空闲项的非循环双向链表。这个数组中的每一个空闲项指向另一个空闲项，而链表中的最后一个元素包含空指针。当撤消一个进程时，task中对应的元素被加到这个表头。

在图3-5中，如果第一个元素为0，tarray_freelist变量就指向元素4，因为它是最后一个被释放的元素。进程对应的元素2和元素1曾被依次撤消，元素2指向task中的另一个空闲元素，没有在图中显示出来。

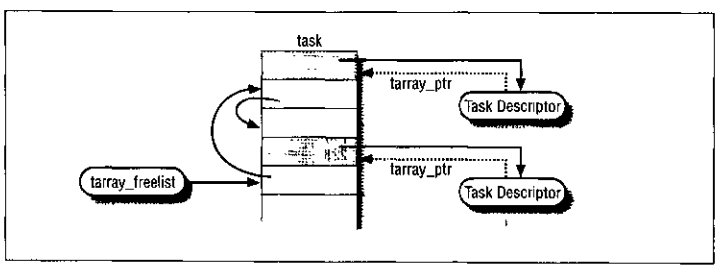


图3-5 具有空闲项的task数组举例

也可以有效地完成从数组中删除一个数组项。每个进程描述符p包括一个tarray_ptr域，而这个域指向的task数组项又包含指向p的指针。

get_free_taskslot()和add_free_taskslot()函数分别用于得到一个空闲项和释放一个项。

进程之间的亲属关系

程序创建的进程具有父/子关系。因为一个进程能创建几个子进程，而子进程之间有兄弟关系。在进程描述符中引入几个域来表示这些关系。进程0和进程1是由内核创建的，我们将在本章看到，进程1（*init*）是所有进程的祖先。一个进程P的描述符包含下列一些域：

p_opptr — *祖先（original parent）*

p_opptr指向创建了进程P的进程描述符，或如果父进程不存在，指向进程1（*init*）的描述符。因此，当一个shell用户启动一个后台进程并从shell退出时，后台进程变成*init*的子进程。

p_pptr — *父进程（parent）*

p_pptr指向P的当前父进程。它的值通常与p_oppt一致，但偶尔也可以不同，例如，当另一个进程发布ptrace()系统调用请求监控P时（参见第十九章中“执行跟踪”一节）。

p_cptr — *子进程（child）*

p_cptr指向P年龄最小的子进程的描述符，即指向刚刚由P创建的进程的进程描述符。

p_ysptr — *弟进程（younger sibling）*

p_ysptr指向在P之后由P的父进程马上创建的进程的进程描述符。

p_osptr — *兄进程（older sibling）*

p_osptr指向在P之前由P的父进程马上创建的进程的进程描述符。

图3-6显示了一组进程之间的亲属关系。进程P_0接连创建了P_1、P_2和进程P_3，而P_3创建了进程P_4。从p_cptr指针出发，利用p_osptr指向兄弟进程，P_0能检索到它所有的子进程。

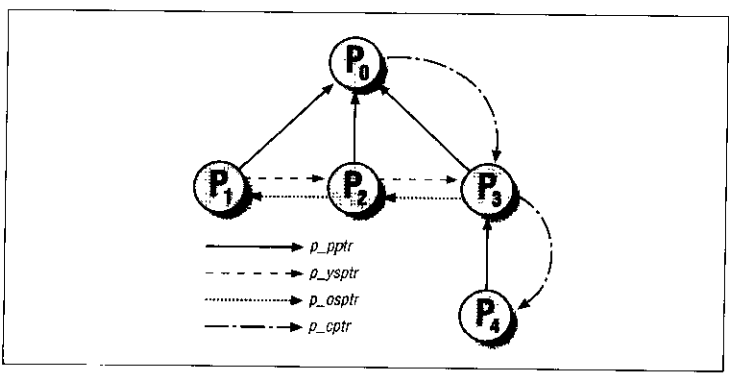


图3-6 五个进程间的亲属关系

等待队列

运行队列链表把处于TASK_RUNNING状态的所有进程组织在一起。当要把其他状态的进程分组时，不同的状态要求不同的处理，Linux选择了下列方式之一：

- TASK_STOPPED或TASK_ZOMBIE状态的进程不链接在专门的链表中，也没必要把它们分组，因为父进程可以通过进程的PID，或进程间的亲属关系检索到子进程。
- 把TASK_INTERRUPTIBLE或TASK_UNINTERRUPTIBLE状态的进程再分成很多类，每一类对应一个特定的事件。在这种情况下，进程状态提供的信息满足不了快速检索进程，因此，有必要引入另外的进程链表。这些附加的链表叫等待队列（wait queue）。

等待队列在内核中有很多用途，尤其对中断处理、进程同步及定时用处更大。因为这些主题在以后的章节中讨论，我们只在这里说明，进程必须经常等待某些事件的发生，例如，等待一个磁盘操作的终止，等待释放系统资源，或等待时间经过固定的间隔。等待队列实现在事件上的条件等待：希望等待特定事件的进程把自己放进合适的等待队列，并放弃控制权。因此，等待队列表示一组睡眠的进程，当某一条件变为真时，由内核唤醒它们。等待队列由循环链表实现，其元素包括指向进程描述符的指针。等待队列中的每个元素都是wait_queue类型：

```
struct wait_queue {
    struct task_struct * task;
    struct wait_queue * next;
};
```

每一个等待队列由一个等待队列指针来标识，等待队列指针或者指向链表中第一个元素的地址，或者为空（如果链表为空的话）。wait_queue数据结构的next域指向链表中的下一个元素，除了最后一个元素［它的next域指向一个空（dummy）的链表元素］。这个空元素的next域包含的值是标识等待队列的变量或域的地址减去一个指针的大小（在Intel平台上，一个指针的大小是4字节）。因此，内核函数把等待队列链表看成一个真正的循环链表，因为最后一个元素指向空的等待队列结构，而它的next域与这个等待队列指针一致（参见图3-7）。

init_waitqueue()函数初始化一个空的等待队列，它接受等待队列指针的地址q作为它的参数，然后把那个指针设置为q - 4。add_wait_queue(q,entry)函数

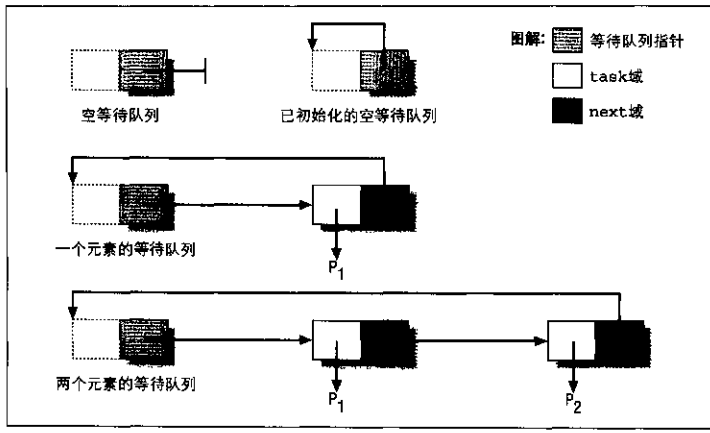


图3-7 等待队列的数据结构

把地址为entry的一个新元素插入到等待队列，等待队列指针q来标识这个等待队列。因为等待队列是由中断处理程序及主要的内核函数来修改，因此，这个函数以关中断执行下列操作（参见第四章）：

```
if (*q != NULL)
    entry->next = *q;
else
    entry->next = (struct wait_queue *)(q-1);
*q = entry;
```

因为把等待队列的指针设置为entry，因此，把新元素放在等待队列链表的第一个位置。如果等待队列不为空，把新元素的next域设置成原来第一个元素的地址，否则，把next域设置成等待队列指针的地址减4，即指向空元素。

remove_wait_queue()函数从等待队列删除由entry指向的元素。再次声明，执行下列操作前，函数必须关中断：

```
next = entry->next;
head = next;
while ((tmp = head->next) != entry)
```

```
        head = tmp;
    head->next = next;
```

这个函数扫描循环链表，查找在entry之前的元素head。通过让head的next域指向紧跟着entry的元素，就能把entry从链表中分离出来。等待队列循环链表的特殊形式简化了代码。此外，基于下列原因，这段代码是很高效的。

- 大多数等待队列仅有一个元素，这就意味着整个循环体从来不执行。
- 当扫描链表时，没必要区分等待队列指针（空等待队列元素）与wait_queue数据结构。

希望等待一个特定事件的进程能调用下列函数中的任一个：

- sleep_on()函数对当前的进程起作用，我们把当前进程叫做P：

```
void sleep_on(struct wait_queue **p)
{
    struct wait_queue wait;
    current->state = TASK_UNINTERRUPTIBLE;
    wait.task = current;
    add_wait_queue(p, &wait);
    schedule();
    remove_wait_queue(p, &wait);
}
```

  函数把P的状态设置为TASK_UNINTERRUPTIBLE，并把P插入等待队列，这个等待队列的指针被指定为参数。然后，它调用调度程序恢复另一个程序的执行。当P被唤醒时，调度程序重新开始执行sleep_on()函数，把P从等待队列中删除。

- interruptible_sleep_on()与sleep_on()函数是一样的，但稍有不同，前者把进程P的状态设置为TASK_INTERRUPTIBLE而不是TASK_UNINTERRUPTIBLE，因此，通过接受一个信号可以唤醒P。
- sleep_on_timeout()和interruptible_sleep_on_timeout()与前面情况类似，但它们允许调用者定义一个时间间隔，过了这个间隔以后，内核唤醒进程。为了做到这点，它们调用schedule_timeout()函数而不是schedule()函数（参见第五章中“动态定时器的应用”一节）。

利用wake_up或者wake_up_interruptible宏，让插入等待队列中的进程进入TASK_RUNNING状态。两个宏都使用__wake_up()函数，其参数为等待队列指针的地址q和指定一个或多个状态的位掩码（bitmask）mode。指定状态的进程将被唤醒，而其他进程的状态保持不变。从本质上说，函数执行下列指令：

```
if (q && (next = *q)) {
    head = (struct wait_queue *)(q-1);
    while (next != head) {
        p = next->task;
        next = next->next;
        if (p->state & mode)
            wake_up_process(p);
    }
}
```

函数通过与mode做“与”操作来检查每个进程的状态p->state，以决定调用者是否要让进程唤醒，只有状态包含在mode的位掩码中的进程才被实际唤醒。wake_up宏在mode中既指定TASK_INTERRUPTIBLE标志，也指定了TASK_UNINTERRUPTIBLE标志，因此，它能唤醒所有睡眠的进程。相反，wake_up_interruptible宏只唤醒TASK_INTERRUPTIBLE状态的进程，当然在mode中只指定了这个状态的标志。注意唤醒的进程并没有从等待队列中删除。已被唤醒的进程并不意味着等待条件已变为真，因此，进程有可能再次把自己挂起。

进程的使用限制

进程与一组使用限制（usage limit）相关联，使用限制指定了进程能使用的系统资源数量。Linux 特别指定了以下的使用限制：

RLIMIT_CPU

进程使用 CPU 的最长时间。如果进程超过了这个限制，内核就向它发一个SIGXCPU信号，然后如果进程还不终止，再发一个SIGKILL信号（参见第九章）。

RLIMIT_FSIZE

允许文件大小的最大值。如果进程试图把一个文件的大小扩充到大于这个值，内核就给这个进程发SIGXFSZ信号。

RLIMIT_DATA

堆大小的最大值。在扩充进程的堆之前，内核检查这个值（参见第七章中“堆的管理”一节）。

RLIMIT_STACK

栈大小的最大值。在扩充进程的用户态堆栈之前，内核检查这个值（参见第七章中“缺页异常处理程序”一节）。

RLIMIT_CORE

内存信息转储文件的大小。当一个进程异常终止时，内核要在进程的当前目录下创建一个内存信息转储文件，在这个文件创建之前，内核检查这个值（参见第九章的“接收信号之前所执行的操作”一节）。如果这个限制为0，那么，内核就不创建这个文件。

RLIMIT_RSS

进程所拥有的页框的最大数。实际上，内核从来不检查这个值，因此，没有实现这个使用限制。

RLIMIT_NPROC

用户能拥有的进程最大数［参见本章“clone()，fork()及vfork()系统调用”一节］。

RLIMIT_NOFILE

打开文件的最大数。当打开一个新文件或复制一个文件描述符时，内核检查这个值（参见第十二章）。

RLIMIT_MEMLOCK

非交换内存的最大尺寸。当进程试图通过mlock()或mlockall()系统调用锁住一个页框时，内核检查这个值（参见第七章中“分配线性地址区间”一节）。

RLIMIT_AS

进程地址空间的最大尺寸。当进程使用malloc()或相关函数扩大它的地址空间时，内核检查这个值（参见第七章中“进程的地址空间”一节）。

使用限制被存放在进程描述符的rlim域，这个域是类型为struct rlimit的一个基本的数组，对每个使用限制：

```
struct rlimit {
```

```
        long rlim_cur;
        long rlim_max;
    };
```

rlim_cur域是资源的当前使用限制。例如，current->rlim[RLIMIT_CPU].rlim_cur表示在CPU上正运行进程所花时间的当前限制。

rlim_max域是资源限制所允许的最大值。利用getrlimit()和setrlimit()系统调用，用户总能把一些资源的rlim_cur限制增加到rlim_max。然而，只有系统管理员才能改变rlim_max域，或把rlim_cur域设置成大于相应的rlim_max域的一个值。

通常，大多数使用限制包含值RLIMIT_INFINITY(0x7fffffff)，这个值意味着对相应的资源没有强加限制。然而，系统管理员可以选择来给一些资源强加更强的限制。只要用户注册进系统，内核就创建一个由系统管理员拥有的进程，系统管理员能调用setrlimit()以减少某些资源的rlim_max和rlim_cur域的值，随后，同一进程执行一个login shell，这个进程就变为由用户拥有。由用户创建的每个新进程都继承其父进程rlim数组的内容，因此，用户不能忽略系统强加的限制。

进程切换

为了控制进程的执行，内核必须有能力挂起正在CPU上运行的进程，并恢复以前挂起的某个进程的执行。这种行为被称为进程切换（process switching），任务切换，或上下文切换。下面描述了在Linux中进程切换的主要内容：

- 硬件上下文
- 硬件支持
- Linux代码
- 保存浮点寄存器

硬件上下文

每个进程可以拥有属于自己的地址空间，但所有进程必须共享CPU寄存器。因此，恢复一个进程的执行之前，内核必须确保每个寄存器装入了挂起进程时的值。

进程恢复执行前必须装入寄存器的一组数据称为硬件上下文（hardware context）。硬件上下文是进程可执行上下文的一个子集，可执行上下文包含了进程执行需要的所有信息。在Linux中，把一个进程的硬件上下文部分存放在TSS段，而剩余部分保存在内核态堆栈。正如我们在第二章中“Linux中的段”一节所认识到的那样，TSS段与进程描述符的tss域一致。

我们假定用prev局部变量表示切换出的进程的进程描述符，next表示切换进的进程的进程描述符。因此，我们把进程切换定义为这样的行为：保存prev硬件上下文，用next硬件上下文代替prev。因为进程切换经常发生，因此减少保存和装入硬件上下文所花费的时间是非常重要的。

在早期的Linux版本中，利用Intel体系结构所提供的硬件支持的优势，通过far jmp指令（注3）指向next进程的TSS描述符的选择符，实现了进程切换。当执行这条指令时，CPU通过自动保存原来的硬件上下文，装入新的硬件上下文来执行硬件上下文的切换。但基于以下原因，Linux 2.2使用了软件方法来执行进程切换：

- 通过一组mov指令的有序执行逐步执行切换，这样能较好地控制被装入数据的合法性。尤其是，这使检查段寄存器的值成为可能，当用单独的far jmp指令时，这类检查是不可能的。
- 旧方法和新方法所需时间大致相同。然而，当当前切换的代码在将来有可能再增强时，旧方法不可能优化硬件上下文切换。

进程切换只发生在内核态。在进程切换执行之前，用户态下的进程使用的所有寄存器的内容都已被保存，这也包括指定用户态堆栈指针地址的ss和esp寄存器的内容。

任务状态段

Intel 80x86体系结构包括了一个特殊的段类型，叫任务状态段（TSS）。正如我们在第二章的“Linux中的段”一节中所看到的那样，每个进程包含有它自己最小长

注3：far jump指令既修改cs寄存器，也修改eip寄存器，而简单的jump指令只修改eip寄存器。

度为104字节的TSS段。操作系统还需要额外的字节来存放硬件不能自动保存的寄存器以及I/O的访问权位图（permission bitmap）。需要这种位图是因为ioperm()及iopl()系统调用可以允许用户态的进程直接访问特殊的I/O端口。尤其是，如果把eflag寄存器中的IOPL域设置为3，就允许用户态的进程访问对应的I/O访问权位图位为0的任何一个I/O端口。

thread_struct结构描述了Linux中TSS的格式。还引入一个额外的区域来存放tr和cr2寄存器、浮点寄存器、调试寄存器及指定给Intel 80x86处理器的其他各种各样的信息。

每个TSS有它自己8字节的任务段描述符（Task State Segment Descriptor，TSSD）。这个描述符包括指向TSS起始地址的32位基地址域，20位限制域，限制域值不能小于0x67（十进制的103，由以前提到的TSS段的最小长度决定）。TSSD的S标志位被清0，以表示相应的TSS是系统段（system segment）。

如果TSSD指向当前正在CPU上运行的进程的TSS，那么Type域被置为11；否则被置为9（注4）。Type域最低第2位叫做忙位（Busy bit），就是这一位区分值9和11（注5）。

由Linux创建的TSSD存放在全局描述符表（GDT）中，GDT的基地址存放在gdtr寄存器中。tr寄存器包含了当前正在CPU上运行的进程的TSSD选择符，也包含了两个隐藏的非编程域：TSSD的Base域和Limit域。通过这种方式，处理器就能直接对TSS寻址，而不用从GDT中检索TSS的地址。

如前所述，Linux把硬件上下文部分存放在进程描述符的tss域，这就意味着当内核创建一个新的进程时，必须初始化TSSD以便能指向这个新的tss域。即使通过软件来保存硬件上下文，TSS段仍起一个重要的作用，因为它包含了I/O访问权位图。事实上，当一个进程在用户态下执行in或out I/O指令时，控制单元执行下列操作：

注4：可以以特定的方式使用Type域来允许以前被挂起的进程自动恢复执行，但Linux没有利用这个硬件特点。更详细的信息可以在Pentium的手册中找到。

注5：因为在对这一位进行修改前，处理器执行“忙锁定”，因此，多任务操作系统可以测试这一位以检查CPU是否试图切换到正在执行的进程。但是Linux没有利用这个硬件特点（参见第十一章）。

1. 检查eflag寄存器的IOPL域，如果把这一位置为3（用户态的进程能访问I/O端口），执行下一个检查；否则，产生一个“一般保护性错误”异常。
2. 访问tr寄存器以确定当前的TSS，也因此确定了合适的I/O访问权位图。
3. 检查I/O指令中指定的I/O端口在I/O访问权位图中对应的位，如果该位为0，则执行这条I/O指令；否则，控制单元产生一个“一般保护性错误”异常。

switch_to 宏

switch_to宏执行进程切换。它利用了prev和next两个参数：第一个参数是挂起进程的进程描述符指针，而第二个参数是在CPU上要执行的进程的进程描述符指针。schedule()函数调用这个宏以调度一个新的进程在CPU上运行（参见第十章）。

switch_to宏是最依赖硬件的内核例程之一。这里描述了在Intel 80x86微处理器上它所做的事情：

1. 在eax和edx寄存器中分别保存prev和next的值（这些值以前存放在ebx和ecx中）：

```
movl %ebx, %eax
movl %ecx, %edx
```

2. 在prev内核态堆栈中保存esi，edi，及ebp寄存器的内容。必须保存这些内容，这是因为编译程序假定switch_to 结束以前它们将保持不变。

```
pushl %esi
pushl %edi
pushl %ebp
```

3. 在prev->tss.esp中保存esp的内容，以便这个域指向prev内核态堆栈的顶部：

```
movl %esp, 532(%ebx)
```

4. 把next->tss.esp装入esp。从现在开始，内核对next的内核态堆栈操作，因此，这条指令执行从prev到next真正的上下文切换。因为进程描述符的地址与其内核态堆栈的地址紧紧地联系在一起（如本章前面“标识一个进程”一节中解释的那样），改变这个内核态堆栈就意味着改变当前进程：

```
movl 532(%ecx), %esp
```

5. 在prev->tss.eip中保存标号为1的地址（在本节的后面说明）。当被取代的进程恢复执行时，进程将执行标号为1的这条指令：

```
movl $1f, 508(%ebx)
```

6. 在next的内核态堆栈顶，把next->tss.eip的值推进栈，在大多数情况下，这个值是标号为1的地址：

```
pushl 508(%ecx)
```

7. 跳转到__switch_to()的C函数：

```
jmp __switch_to
```

这个函数作用于prev和next参数，这两个参数分别表示前一个进程和新进程。这个函数的调用不同于一般函数的调用，因为__switch_to()从eax和edx取参数prev和next（我们在前面已看到这些参数就是保存在那里），而不像大多数函数一样从栈中取参数。为了强迫这个函数到寄存器取它的参数，内核利用__attribute__和regparm关键字，这两个关键字是C语言非标准的扩展名，由gcc编译程序实现。在*include/asm-i386 /system.h*头文件中，__switch_to()函数的声明如下：

```
__switch_to(struct task_struct *prev,
            struct task_struct *next)
  __attribute__ (regparm(3))
```

这个函数完成了由switch_to()宏开始的进程切换，它包含了扩展的内联汇编语言代码，读起来非常复杂，因为代码引用的寄存器用到特殊的符号。为了简化下面的讨论，我们将描述由编译程序产生的汇编语言指令：

a. 把esi和ebx寄存器的内容保存在next的内核态堆栈，然后把参数prev和next分别装入ecx和ebx寄存器：

```
pushl %esi
pushl %ebx
movl %eax, %ecx
movl %edx, %ebx
```

b. 执行由unlazy_fpu()宏产生的代码（参见本章的“保存浮点寄存器”一节），以确保数学协处理器工作时能保存其寄存器的内容。正如我们将在后面看到的那样，当执行上下文切换时，没有必要装入next的浮点寄存器：

```
unlazy_fpu(prev);
```

c. 清除next的忙位（参见本章前面“任务状态段”一节），并且在tr寄存器中装入TSS的选择符：

```
movl 712(%ebx), %eax
andb $0xf8, %al
andl $0xfffffdff, gdt_table+4(%eax)
ltr 712(%ebx)
```

前述的代码相当紧凑，它作用于：

进程的TSSD选择符，把它从next->tss.tr拷贝到eax。

选择符的低8位被存放在al（注6）中。al的低3位包含TSSD的RPL和TI域。

清al的低3位，把剩余的TSSD索引左移3位（即乘以8）。因为TSSD是8字节长，索引值乘以8产生TSSD在GDT中的相对地址。gdt_table+4(%eax)符号指向TSSD中第5个字节的地址，andl指令清第5个字节的忙位，而ltr把next->tss.tr选择符放在tr寄存器中，并置忙位（注7）。

d. 把fs和gs段寄存器的内容分别存放在prev->tss.fs和prev->tss.gs中：

```
movl %fs,564(%ecx)
movl %gs,568(%ecx)
```

e. 把next->tss.ldt的值装到ldtr寄存器。只有当prev使用的局部描述符表（LDT）与next所使用的不同时，才有必要进行这种操作：

```
    movl 920(%ebx),%edx
    movl 920(%ecx),%eax
    movl 112(%eax),%eax
    cmpl %eax,112(%edx)
    je 2f
    lldt 572(%ebx)
2:
```

注6：ax寄存器由eax的低16位组成。此外，al寄存器由ax的低8位组成，而ah由ax的高8位组成。类似的表示可以应用到ebx、ecx及edx寄存器。ax的低13位指定GDT内TSSD的索引。

注7：在向tr寄存器中装入值之前，Linux必须清忙位，否则控制单元将产生一个异常。

在实际中，这种检查是由指向tss.segments的域（在进程描述符中偏移为112）进行的，而不是由tss.ldt域进行的。

f. 把next->tss.cr3的值装入cr3寄存器。但是，如果prev和next是轻量级进程，就要避免这种操作，因为轻量级进程共享相同的页全局目录（PGD）。因为prev的PGD从不改变，就没必要保存它。

```
    movl 504(%ebx),%eax
    cmpl %eax,504(%ecx)
    je 3f
    movl %eax,%cr3
3:
```

g. 分别把包含在next->tss.fs和next->tss.gs中的值装入fs和gs段寄存器。从逻辑上说，这一步补充了在第7d步执行的操作。

```
movl 564(%ebx),%fs
movl 568(%ebx),%gs
```

实际上这段代码更复杂，因为当CPU检测到一个无效的段寄存器值时，应该能产生一个异常。这段代码采用“修正（fix-up）”方法来处理这个异常（参见第八章中“动态地址检查：修正代码”一节）。

h. 用next->tss.debugreg[i]的值（0 i 7）装载八个调试寄存器（注8）。如果next挂起时使用了调试寄存器才这么做（即域next->tss.debugreg[7]不为0）。正如我们将在第十九章看到的那样，通过写TSS才能修改这些寄存器的值，因此，没有必要保存它们：

```
    cmpl $0,760(%ebx)
    je 4f
    movl 732(%ebx),%esi
    movl %esi,%db0
    movl 736(%ebx),%esi
    movl %esi,%db1
    movl 740(%ebx),%esi
    movl %esi,%db2
    movl 744(%ebx),%esi
    movl %esi,%db3
```

注8：Intel 80x86调试寄存器允许硬件监控进程。最多可定义4个断点区。只要被监控进程所产生的线性地址包含在其中一个断点区中，就产生一个异常。

```
    movl 756(%ebx),%esi
    movl %esi,%db6
    movl 760(%ebx),%ebx
    movl %ebx,%db7
4:
```

i. 在第7a步，把ebx和esi寄存器的值压进了堆栈，在这里，通过恢复这两个寄存器的值结束函数：

```
popl %ebx
popl %esi
ret
```

当执行ret指令时，控制单元从栈取出eip程序计数器中的值，这个值通常是标号为1的指令地址，在下一步将说明标号为1的地址是由switch_to宏存放在栈中的。然而，如果next因为是第一次执行还从未被挂起过，这个函数将找到ret_from_fork()函数的起始地址［参见本章的“clone()，fork()及vfork()系统调用”一节］。

8. switch_to宏的剩余部分包含的几条指令用来恢复esi、edi及ebp寄存器的内容。这三条指令的第一条标号为1：

```
1:  popl %ebp
    popl %edi
    popl %esi
```

注意这些pop指令用到prev进程的内核态堆栈。当调度程序选择prev作为新进程在CPU上执行时，这些指令被执行，因此把prev作为switch_to的第二个参数来调用switch_to。此时，esp寄存器指向prev的内核态堆栈。

保存浮点寄存器

从Intel 80486开始，算术浮点单元（FPU）已被集成到CPU中。数学协处理器这个名词只是使人想起使用昂贵的专用芯片执行浮点计算的岁月。然而，为了维持与旧模式的兼容，采用*ESCAPE*指令执行浮点算术函数，这个指令的一些前缀字节在0xd8和0xdf之间。这些指令作用于包含在CPU中的浮点寄存器集。显然，如果一个进程用了ESCAPE指令，那么，浮点寄存器的内容属于它的硬件上下文。

最近，Intel在它的微处理器中引入了一个新的汇编指令集，叫做MMX指令，用来加速多媒体应用程序的执行。MMX指令作用于FPU的浮点寄存器。选择这种体系

结构的明显缺点是编程者不能把浮点指令与MMX指令混在一起使用。优点是操作系统设计者能忽视新指令集，因为保存浮点单元状态的任务切换代码可以方便地应用到保存MMX状态。

Intel 80x86微处理器并不在TSS中自动保存浮点寄存器。然而，处理器中包含了一些硬件支持，能在需要时保存这些寄存器的值。硬件支持由cr0寄存器中的一个TS（任务切换）标志组成，它遵循以下规则：

- 每当执行硬件上下文切换时，设置TS位。
- 每当设置TS位时执行ESCAPE或MMX指令，控制单元就产生一个“设备不可用”的异常（参见第四章）。

TS标志允许内核只有在真正需要时才保存和恢复浮点寄存器。为了说明它如何工作，让我们假设进程A使用数学协处理器。当发生上下文切换时，内核置TS标志并把浮点寄存器保存在进程A的TSS中。如果新进程B不利用协处理器，内核不必恢复这个浮点寄存器的内容。但是，只要B打算执行ESCAPE或MMX指令，CPU就产生一个“设备不可用”的异常，并且相应的处理程序用保存在进程B中的TSS的值装载浮点寄存器。

现在，让我们描述一下为了对浮点寄存器有选择地保存而引入的数据结构。这些数据结构存放在tss.i387的子域中，这个子域的形式由i387_hard_struct结构描述。进程描述符也存放了两个附加标志的值：

- PF_USEDFPU标志包含在flags域中，它指明当进程最后在CPU上执行时是否使用浮点寄存器。
- used_math域。这个域在两种情况下被清0（没有意义）：
  - 当进程调用execve()系统调用（参见第十九章），开始一个新进程的执行时。因为控制将不再返回到前一个程序，存放在tss.i387中的当前数据也不再使用。
  - 当在用户态下执行一个程序的进程开始执行一个信号处理程序时（参见第九章）。因为信号处理程序与程序的执行流是异步的，因此，浮点寄存器对信号处理程序来说是毫无意义的。然而，开始执行这个处理程序之前，内核在tss.i387中保存浮点寄存器，处理程序结束以后恢复它们。因此，允

许信号处理程序使用数学协处理器，但是，在正常的程序流执行期间，协处理器不能继续进行已开始的浮点计算。

如前所述，__switch_to()函数执行unlazy_fpu宏，这个宏产生下列的代码：

```
if (prev->flags & PF_USEDFPU) {
    /* 保存浮点寄存器 */
    asm("fnsave %0" : "=m" (prev->tss.i387));
    /* 等待，直到所有的数据被传递 */
    asm("fwait");
    prev->flags &= ~PF_USEDFPU;
    /* 把cr0中TS的标志设置为1 */
    stts();
}
```

stts()宏设置cr0的TS标志，实际上，它产生下面的汇编语言指令：

```
movl %cr0, %eax
orb $8, %al
movl %eax, %cr0
```

当一个进程恢复执行以后，浮点寄存器的内容还没有被完全保存，然而，cr0的TS标志位已由unlazy_fpu()设置。因此，进程第一次执行ESCAPE或MMX指令时，控制单元产生一个“设备不能用”的异常，内核（更确切地说，由异常调用的异常处理程序）运行math_state_restore()函数：

```
void math_state_restore(void) {
    asm("clts"); /* 清除cr0的TS标志 */
    if (current->used_math)
        /* 装入浮点寄存器 */
        asm("frstor %0": :"m" (current->tss.i387));
    else {
        /* 初始化浮点单元 */
        asm("fninit");
        current->used_math = 1;
    }
    current->flags |= PF_USEDFPU;
}
```

因为进程正在执行ESCAPE指令，因此，这个函数设置PF_USEDFPU标志。此外，这个函数还清除cr0的TS标志，因此，进程以后执行ESCAPE或MMX指令时将不触发“设备不可用”的异常。如果存放在tss.i387域中的数据是有效的，这个

函数用合适的值装载浮点寄存器。否则，FPU被重新初始化，并且所有的寄存器被清0。

创建进程

Unix操作系统为了满足用户的请求，非常依赖进程创建。例如，只要用户输入一条命令，shell进程就创建一个新进程，新进程执行shell的另一个拷贝。

传统的Unix操作系统以统一的方式对待所有的进程：父进程所拥有的资源被复制，被复制的那份用于运行子进程。这种方法使进程的创建非常慢且效率低，因为子进程需要拷贝父进程的整个地址空间。实际上，子进程几乎不必读或修改父进程拥有的所有资源，在很多情况下，子进程立即调用execve()，并清除父进程很仔细地保存过的地址空间。

现代Unix内核通过引入三种不同的机制解决了这个问题：

- 写时复制技术允许父子进程能读相同的物理页。只要两者中有一个进程试图写一个物理页，内核就把这个页的内容拷贝到一个新的物理页，并把这个新的物理页分配给正在写的进程。第七章将全面地解释这种技术在Linux中的实现。
- 轻量级进程允许父子进程共享每一进程在内核的数据结构，如页表（也就是整个用户态地址空间）及打开文件表。
- vfork()系统调用创建的一个进程能共享其父进程的内存地址空间。为了防止父进程重写子进程需要的数据，阻塞父进程的执行，一直到子进程退出或执行一个新的程序。我们将在后面了解到有关vfork()系统调用更多的知识。

clone()，fork()及vfork()系统调用

在Linux中，轻量级进程是由名为__clone()的函数创建的，这个函数用了四个参数：

fn

指定一个由新进程执行的函数。当这个函数返回时，子进程终止。函数返回一个整数，表示子进程的退出代码。

arg

指向传递给 fn() 函数的数据的指针。

flags

各种各样的信息。低字节指定子进程结束时发送到父进程的信号编号，通常选择 SIGCHLD 信号。剩余的3个字节给一组克隆标志编码，编码指定父子进程间共享的资源。设置的标志具有以下含义：

CLONE_VM

内存描述符和所有的页表（参见第七章）。

CLONE_FS：

识别根目录和当前工作目录的表。

CLONE_FILES：

识别打开文件表（参见第十二章）。

CLONE_SIGHAND：

识别信号处理程序的表（参见第九章）。

CLONE_PID：

PID（注9）。

CLONE_PTRACE：

如果 ptrace() 系统调用引起父进程被跟踪，那么，子进程也被跟踪。

CLONE_VFORK：

用在 vfork() 系统调用（参见本节后面部分）。

child_stack

指定把用户态堆栈指针赋给子进程的 esp 寄存器。如果这个参数为0，内核把当前父进程的栈指针赋给子进程。因此，父子进程暂时共享同一用户态堆栈。但是，借助于写时复制机制，通常只要父子进程中有一个试图去改变栈，则立即各能得到用户态堆栈的一份拷贝。当然，如果子进程和父进程共享同一地址空间，这个参数必须有一个非空值。

注9：我们在后面将看到，只有PID为0的进程使用CLONE_PID标志。在单处理器系统上，没有两个轻量级进程的PID是相同的。

实际上，__clone()是一个在C语言库中定义的封装（wrapper）函数（参见第八章的“POSIX API和系统调用”一节），C语言库又使用了对编程者隐藏的一个系统调用，名叫clone()。clone()系统调用只利用了flags和child_stack两个参数，新进程总是从系统调用指令的下一条指令开始它的执行。当这个系统调用返回到__clone()函数时，它先确定自己是在父进程中还是在子进程中，然后强迫子进程执行fn()函数。

Linux用clone()实现了传统的fork()系统调用，clone()的第一个参数指定为SIGCHLD信号，并把所有的克隆标志清0，第二个参数为0。

前面描述的vfork()系统调用在Linux中是由clone()实现，clone()的第一个参数指定为SIGCHLD信号和CLONE_VM及CLONE_VFORK标志，第二个参数为0。

不管是发布clone()，fork()还是vfork()系统调用，内核都调用do_fork()函数，其执行步骤如下：

1. 如果指定CLONE_PID标志，do_fork()函数检查父进程的PID是否不为空，如果不为空，就返回一个错误码。只有*swapper*进程能设置CLONE_PID，在初始化多处理器系统时需要设置该标志（参见第十一章中“主要的SMP数据结构”一节）。

2. 调用alloc_task_struct()函数以获得8KB的union task_union内存区来存放进程描述符和新进程的内核态堆栈。

3. do_fork()接着让当前指针指向父进程描述符，并把父进程描述符的内容拷贝到刚刚分配的内存区的新进程描述符中。

4. 有几个检查以确认用户具有开始执行一个新进程所必须的资源。首先，do_fork()检查current->rlim[RLIMIT_NPROC].rlim_cur的值是否小于或等于用户拥有的当前进程数，如果是，则返回一个错误代码，否则，从名为user_struct的数据结构获得这个值。在进程描述符的user域通过一个指针就能找到这个数据结构。

5. 调用find_empty_process()函数。如果父进程的拥有者不是超级用户，这个函数检查nr_tasks（系统中进程总数）的值是否小于NR_TASKS减去MIN_

TASKS_LEFT_FOR_ROOT（注10），如果是，find_empty_process()调用get_free_taskslot()在task数组中找到一个空闲项，否则，返回一个错误。

6. do_fork()把新进程描述符指针写到前面获得的task数组项，并把进程描述符中的tarray_ptr域设置成那个项的地址（参见前面"标识一个进程"一节）。

7. 如果新进程利用了一些内核模块，do_fork()就增加相应模块的引用计数器（reference counter）。每个模块都有自己的引用计数器，用来表示有多少个进程使用这个模块。直到一个模块的引用计数器为空，才能删除这个模块（参见附录二）。

8. 然后，do_fork()更新一些从父进程拷贝来的标志域的标志。

   a. 清PF_SUPERPRIV标志，这个标志表示进程是否使用了其超级用户的任何特权。

   b. 清PF_USEDFPU标志。

   c. 除非设置了CLONE_PTRACE参数标志，否则清PF_PTRACED标志。当设置了CLONE_PTRACE标志，就意味着父进程正由ptrace()函数跟踪，因此，子进程也被跟踪。

   d. 除非再一次设置CLONE_PTRACE参数标志，否则清PF_TRACESYS标志。

   e. 设置PF_FORKNOEXEC标志，这个标志表示子进程还没有发布execve()系统调用。

   f. 根据CLONE_VFORK标志的值设置PF_VFORK标志。这说明只要子进程发布PF_VFORK系统调用或终止，必须唤醒父进程。

9. 现在，do_fork()已经获得它从父进程能利用的几乎所有的东西，剩下的活动就是集中建立子进程的新资源，并让内核知道这个新进程已经诞生。首先，do_fork()调用get_pid()函数获得一个新的PID，这个PID将赋给子进程（除非设置了CLONE_PID标志）。

10. 然后，更新不能从父进程继承的进程描述符的所有域，例如指定进程间亲属关系的域。

注10：有一些进程，通常是4个，被保留给超级用户；MIN_TASKS_LEFT_FOR_ROOT引用这个数字。这样，即使有一个用户被允许使用"fork bomb"（一个可以永远派生自己的在线程序）迁载系统，超级用户也可以登录，杀死一些进程，并开始搜寻此违法用户。

11. 除非flag参数有不同的指定，否则，do_fork()调用copy_files()，copy_fs()，copy_sighand()及copy_mm()来创建新的数据结构，并把父进程相应的数据结构拷贝到这几个新创建的数据结构中。

12. 当发出clone()调用时，do_fork()调用copy_thread()，用包含在CPU寄存器中的值初始化子进程的内核态堆栈（如第八章描述的那样，这些值已经被保存在父进程的内核态堆栈中）。然而，do_fork()强迫eax寄存器对应域的值为0。用内核态堆栈的基地址初始化子进程TSS的tss.esp域，并且把汇编语言函数［ret_from_fork()］的地址保存在tss.eip域。

13. 用SET_LINKS宏把新的进程描述符插入进程链表。

14. 用hash_pid()函数把新的进程描述符插入pidhash散列表。

15. 增加nr_tasks和current->user->count的值。

16. 把子进程描述符的状态域设置成TASK_RUNNING，并调用wake_up_process()，把子进程插入到运行队列链表。

17. 如果指定了CLONE_VFORK标志，do_fork()挂起父进程，直到子进程释放它的内存地址空间（即，直到子进程要么结束，要么执行一个新程序）。为了做到这点，进程描述符包括了一个叫vfork_sem的内核信号量（参见第十一章的“使用内核信号量加锁”一节）。

18. 返回子进程的PID，这个PID最终由用户态下的父进程读取。

现在，我们有了处于可运行状态的完整的子进程。但是，它还没有实际运行，调度程序要决定何时把CPU交给这个子进程。在以后的进程切换中，调度程序继续完善子进程：用子进程描述符tss中域的值装载几个CPU寄存器。特别是用tss.esp装载esp寄存器（即子进程内核态堆栈的地址），把函数ret_from_fork()的地址装入eip寄存器。接下来，这个汇编语言函数调用ret_from_sys_call()函数（参见第八章），此函数用存放在栈中的值再装载所有的寄存器，并强迫CPU返回到用户态。然后，在fork()，vfork()，或clone()系统调用结束时，新进程将开始执行。由系统调用返回的值包含在eax中，值为0，就给子进程，等于PID，就给子进程的父进程。

除非fork返回一个空的PID，否则，子进程将与父进程执行相同的代码。熟悉Unix的编程者可以在基于PID值的程序中插入一个条件语句使子进程与父进程有不同的行为。

内核线程

传统的Unix系统把一些重要的任务委托给周期性执行的进程，这些任务包括刷新磁盘高速缓存，交换出不用的页框，维护网络链接等等。事实上，以严格线性的方式执行这些任务的确效率不高，如果把他们放在后台调度，不管是对它们的函数还是对终端用户进程都能得到较好的响应。因为一些系统进程只在内核态运行，现代操作系统把它们的函数委托给内核线程（kernel thread），内核线程不受不必要的用户态上下文的拖累。在Linux中，内核线程在以下几方面不同于普通进程：

- 每个内核线程执行一个单独指定的内核函数，而普通进程只有通过系统调用执行内核函数。
- 内核线程只运行在内核态，而普通进程既可以运行在内核态，也可以运行在用户态。
- 因为内核线程只运行在内核态，它们只使用大于PAGE_OFFSET的线性地址空间。另一方面，不管在用户态还是在内核态，普通进程可以用4GB的线性地址空间。

创建一个内核线程

kernel_thread()函数创建一个新的内核线程，并只能由另一个内核线程来执行。这个函数所包含的代码大部分是内联式汇编语言，但在某种程度上等价于下面的代码：

```
int kernel_thread(int (*fn)(void *), void * arg,
                  unsigned long flags)
{
    pid_t p;
    p = clone( 0, flags | CLONE_VM );
    if ( p )          /* parent */
        return p;
    else {            /* child */
        fn(arg);
        exit();
    }
}
```

进程 0

所有进程的祖先叫做进程0，或因为历史的原因叫做*swapper*进程，它是在Linux的初始化阶段由start_kernel()函数从无到有创建的一个内核线程（参见附录一）。这个祖先进程利用下列数据结构：

- 存放在init_task_union变量中的一个进程描述符和一个内核态堆栈。init_task和init_stack宏分别产生进程描述符和栈的地址。
- 进程描述符指向下列表：
  - init_mm
  - init_mmap
  - init_fs
  - init_files
  - init_signals

  由下列宏分别初始化这些表：
  - INIT_MM
  - INIT_MMAP
  - INIT_FS
  - INIT_FILES
  - INIT_SIGNALS
- 一个TSS段，由INIT_TSS宏初始化。
- 两个段描述符，叫做TSSD和LDTD，存放在GDT中。
- 页全局目录，存放在swapper_pg_dir，也可以认为它是内核页全局目录，因为它由所有的内核线程使用。

start_kernel()函数初始化内核需要的所有数据结构，开中断，创建另一个内核线程，这个线程命名为进程1，更一般的叫法为*init*进程：

```
kernel_thread(init, NULL,
              CLONE_FS  CLONE_FILES | CLONE_SIGHAND);
```

新创建内核线程的PID为1，并与进程0共享每个进程在内核中的所有数据结构。此外，当调度程序选择到它时，*init*进程开始执行init()函数。

创建*init*进程后，进程0执行cpu_idle()函数，该函数主要是在允许中断的情况下重复执行hlt汇编语言指令（参见第四章）。只有当没有其他进程处于可运行（TASK_RUNNING）状态时，调度程序才选择进程0。

进程1

由进程0创建的内核线程执行init()函数，init()四次轮流调用kernel_thread()函数为常规内核任务初始化四个必要的内核线程：

```
kernel_thread(bdflush, NULL,
              CLONE_FS | CLONE_FILES | CLONE_SIGHAND);
kernel_thread(kupdate, NULL,
              CLONE_FS | CLONE_FILES | CLONE_SIGHAND);
kernel_thread(kpiod, NULL,
              CLONE_FS | CLONE_FILES | CLONE_SIGHAND);
kernel_thread(kswapd, NULL,
              CLONE_FS | CLONE_FILES | CLONE_SIGHAND);
```

结果是，创建了四个额外的内核线程来处理内存高速缓存及交换活动：

kflushd（即bdflush）

刷新“脏”缓冲区中的内容到磁盘以归还内存，正如第十四章中“把脏缓冲区写入磁盘”一节所描述的那样。

Kupdate

刷新旧的“脏”缓冲区中的内容到磁盘以减少文件系统不一致的风险，参考第十四章中“把脏缓冲区写入磁盘”一节。

kpiod

把属于共享内存映射的页面交换出去，如第十六章中“从共享内存映射中换出页”一节中所述。

Kswapd

执行内存回收功能，如第十六章中“kswapd内核线程”一节中所述。

init()调用execve()系统调用装入可执行程序*init*。因此，*init*内核线程变成…

个普通的进程，它拥有自己的内核数据结构。*init*进程从不终止，因为它创建和监控操作系统外层的所有进程的活动。

撤消进程

很多进程终止了它们本该执行的代码，从这种意义上说，这些进程“死”了。当这种情况发生时，必须通知内核以便内核释放进程所拥有的资源，包括内存，打开文件及其他我们在本书碰到的零碎东西，如信号量。

进程终止的一般方式是调用exit()系统调用。这个系统调用可能由编程者明确地插入。另外，当控制流到达主过程［C程序中的main()函数］的最后一条语句时，执行exit()系统调用。

内核可以有选择地强迫进程死。发生在以下两种典型情况下：当进程接受到一个不能处理或忽视的信号时，或者当在内核态产生一个不可到达的CPU异常时而内核正在代表进程运行。

进程终止

所有进程的终止都是由do_exit()函数来处理，这个函数从内核数据结构中删除对终止进程的大部分引用。do_exit()函数执行下列动作：

1. 在进程描述符的flag域设置PF_EXITING标志，表示进程正在被删除。
2. 如果必要，删除进程描述符，这可以通过sem_exit()函数从IPC信号量队列中删除（参见第十八章），或者通过del_timer()函数（参见第五章）从定时器队列中删除。
3. 分别检查与分页、文件系统、打开文件描述符及信号处理相关的数据结构，要使用的函数分别是__exit_mm()，__exit_files()，__exit_fs()及__exit_sighand()，如果没有其他进程共享这些数据结构，这些函数也删除对应的数据结构。
4. 把进程描述符的state域设置成TASK_ZOMBIE，我们将在下一节中看到僵死进程会出现什么情况。

5. 把进程描述符的exit_code域设置成进程终止代号，这个值或者是exit()系统调用参数（正常终止），或者是由内核提供的一个错误代号（异常终止）。

6. 调用exit_notify()函数更新父子进程间的亲属关系。由终止进程创建的所有子进程都成为*init*的子进程。

7. 调用schedule()函数（参见第十章），选择一个新的进程运行。因为调度程序忽略处于TASK_ZOMBIE状态的进程，因此，在调用schedule()时，原先的进程正好在switch_to宏之后将停止执行。

删除进程

Unix允许一个进程查询内核以获得其父进程的PID,或者其任何子进程的执行状态。例如，进程可以创建一个子进程执行特定的任务，然后调用wait()类系统调用检查子进程是否终止。如果子进程已经终止，那么，它的终止代号将告诉父进程这个任务是否已成功地进行。

为了遵循这些设计，允许Unix内核在进程终止后丢弃包含在进程描述符域中的数据。但是，只有父进程发布了一个与已终止的进程相关的wait()类系统调用之后，才允许这样做。这就是为什么引入僵死状态，尽管从技术上来说进程已死，但它的描述符必须保存，直到父进程得到通知。

如果父进程在子进程之前结束会发生什么情况呢？在这种情况下，已僵死进程可能用可用的task项塞满了系统。如前所述，这个问题的解决必须强迫所有的孤儿进程成为*init*进程的子进程。以这种方式，*init*进程在用wait()类系统调用检查它合法的子进程终止时，就可以撤消僵死的进程。

通过执行下列步骤，release()函数释放一个僵死进程的描述符：

1. 调用free_uid()函数，对到目前为止终止进程的拥有者所创建的进程数减1。这个值存放在本章前面提到的user_struc数据结构中。

2. 调用add_free_taskslot()函数，释放task中要释放的进程描述符项。

3. 减nr_tasks变量的值。

4. 调用unhash_pid()，从pidhash散列表中删除该进程描述符。

5. 调用 REMOVE_LINKS 宏，从进程链表中删除该进程描述符。

6. 调用 free_task_struct()函数，释放 8KB 包含该进程描述符和进程内核态堆栈的内存区。

对 Linux 2.4 的展望

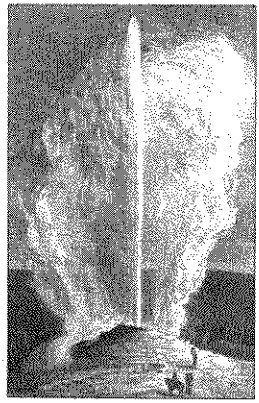
新内核支持大量的用户和组，因为它使用 32 位的 UID 和 GID。

为了提高对进程个数的硬编码界限，Linux 2.4 去掉了 tasks 数组，在以前的版本中该数组包含了指向所有进程描述符的指针。

此外，Linux 2.4 不再为每个进程包含一个任务状态段（TSS）。因此，进程描述符表中的tss域被指向一个数据结构的指针代替，这个数据结构存放的信息就是以前存放在 TSS 中的内容，即寄存器内容和 I/O 位图。Linux 2.4 为系统中每个 CPU 只用一个 TSS。当发生上下文切换时，内核利用每个进程的数据结构保存和恢复寄存器的内容，并填充正在 CPU 上执行进程的 TSS 的 I/O 位图。

Linux2.4加强了等待队列。睡眠进程被存放在一个由高效的list_head数据类型所实现的链表中。此外，内核现在只能唤醒在等待队列中睡眠的单独的一个进程，因此，极大地提高了信号量的效率。

最后，Linux 2.4 给 clone()系统调用增加了一个新标志 CLONE_PARENT，允许新的轻量级进程与调用了系统调用的进程有相同的父进程。



第四章

中断和异常

中断（interrupt）通常被定义为改变处理器执行指令顺序的事件。这样的事件对应于CPU芯片内部和外部的硬件电路产生的电信号。

中断通常被分为同步（synchronous）中断和异步（asynchronous）中断：

- 同步中断是当指令执行时由CPU控制单元产生的，之所以称为同步，是因为只有在一条指令终止执行后CPU才会发出中断。
- 异步中断是由其他硬件设备依照CPU时钟信号随机产生的。

在Intel 80x86微处理器手册中，把同步和异步中断分别称为异常（exception）和中断（interrupt）。我们也采用这种分类，当然有时我们也用术语“中断信号”指这两种类型（同步及异步）。

中断是由间隔定时器和I/O设备发出的，例如，用户的一次按键将会引起一个中断。另一方面，异常由程序的错误所产生，或者由内核必须处理的异常条件所产生。第一种情况下，内核对这个异常的处理是产生一个Unix程序员非常熟悉的信号。第二种情况下，内核执行从异常条件恢复所需要的所有步骤，例如一个缺页，或对内核服务的一个请求（通过一条int指令）。

我们将通过介绍这样的信号来开始“中断信号的作用”这一节。然后，我们将说明由I/O设备发出的众所周知的IRQ是如何引起中断的，我们将详细讨论Intel 80x86微处理器是如何在硬件级处理中断和异常的。接下来，我们将在“初始化中断描述符表”一节说明Linux是如何初始化Intel中断体系结构必需的所有的数据结构。剩余的三部分描述Linux是如何在软件级处理中断信号的。

继续前进之前，需要值得注意的是：我们在本章仅涉及对所有PC都通用的“标准的”中断，而不涉及一些体系结构的非标准中断，例如，膝上型电脑产生的中断类型没有在这里讨论。专门针对多处理器体系结构的其他中断类型将在第十一章中给予简单描述。

中断信号的作用

顾名思义，中断信号提供了一种方式，使处理器转而去运行正常控制流之外的代码。当一个中断信号达到时，CPU必须停止它当前正在做的事情，并且切换到一个新的活动。为了做到这一点，就要在内核态堆栈保存程序计数器的当前值（即eip和cs寄存器的内容），并把与中断类型相关的一个地址放进程序计数器。

在本章，有些事情会使你想起在前一章描述的上下文切换，这发生在内核用一个进程替换另一个进程时。但是，中断处理与进程切换有一个明显的差异：由中断或异常处理程序执行的代码不是一个进程。更正确地说，它是一个内核控制路径，代表中断发生时正在运行的进程执行（参见本章“中断和异常处理程序的嵌套执行”一节）。作为一个内核控制路径，中断处理程序比一个进程要“轻（light）”（中断的上下文很少，建立或终止需要的时间很少）。

中断处理是由内核执行的最敏感的任务之一，因为它必须满足下列约束：

- 当内核正打算去做一些其它的事情时，中断随时会到来。因此，内核的目标就是让中断尽可能快地被处理完，尽其所能把其他更多的处理向后推迟。例如，假设一个数据块已达到了网线，当硬件中断内核时，内核只简单地标志数据到来了，让处理器恢复到它以前运行的状态，其余的处理稍后再进行（如把数据移入一个缓冲区，它的接受进程可以在缓冲区找到数据并恢复这个进程的执行）。为应答中断，可以把内核需要执行的活动分为两部分：上半部分（top half）和下半部分（bottom half）。上半部分内核立即执行，而下半部分留着稍

后处理。内核维持着一个队列，这个队列指向表示下半部分、等待被执行的所有函数，并且内核在处理到某个特定的点上时，把这些函数从队列中取出去执行。

- 因为中断随时会到来，所以内核可能正在处理其中的一个中断时，另一个中断（不同类型）又发生了。应该尽可能多地允许这种情况发生，因为这能维持更多的I/O设备处于忙状态（参见“中断和异常处理程序的嵌套执行”一节）。因此，中断处理程序必须被编写成使相应的内核控制路径能以嵌套的方式执行。当最后一个内核控制路径终止时，内核必须能恢复被中断进程的执行，或者，如果中断信号已导致了重新调度的动作，内核能切换到另外的进程。
- 尽管内核在处理前一个中断时可以接受一个新的中断，但在内核代码中还是存在一些临界区，在那里，中断必须被关闭。必须尽可能地限制这样的临界区，因为根据以前的要求，内核，尤其是中断处理程序，应该在大部分时间内以开中断的方式运行。

中断和异常

Intel文档把中断和异常分为以下几类：

- 中断：

  可屏蔽中断（Maskable interrupt）：

  被送到微处理器的INTR引脚。通过清eflag寄存器的IF标志关闭中断。所有I/O设备发出的IRQ均可引起可屏蔽中断。

  不可屏蔽中断（Nonmaskable Interrupt）

  被送到微处理器的NMI（Nonmaskable Interrupt）引脚。通过清IF标志，中断不能关闭。只有几个危急的事件，例如硬件故障，才引起不可屏蔽中断。

- 异常：

  处理器探测异常（Processor-detected exceptions）

  当CPU执行一条指令时所探测到的一个反常条件所产生的异常。可以进一步分为三组，这取决于CPU控制单元产生异常时保存在内核态堆栈eip寄存器的值：

故障（Fault）

保存在eip中的值是引起故障的指令地址，因此，当异常处理程序终止时，会重新执行那条指令。我们将在第七章的“缺页异常处理程序”一节中看到，只要处理程序能纠正引起异常的反常条件，重新执行同一指令就是必要的。

陷阱（Trap）

保存在eip中的值是一个指令地址，该指令在引起陷阱的指令地址之后。只有当没有必要重新执行已执行过的指令时，才触发陷阱。陷阱的主要用途是为了调试程序。在这种情况下，中断信号的作用是通知调试程序一条特殊指令已被执行（例如到了一个程序的断点）。一旦用户检查到调试程序所提供的数据，她就可能要求被调试程序从下一条指令重新开始执行。

异常结束（Abort）

发生一个严重的错误。控制单元出了麻烦，不能在eip寄存器中保存有意义的值。异常结束是由硬件故障或系统表中无效的值引起的。由控制单元发送的这个中断信号是紧急信号，用来把控制切换到相应的异常结束处理程序，这个异常结束处理程序除了迫使受影响的进程终止外，没有别的选择。

编程异常（Programmed exception）

在编程者发出请求时发生。是由int或int3指令触发的；当into（检查溢出）和bound（检查地址出界）指令检查的条件不为真时，也引起编程异常。控制单元把编程异常作为陷阱来处理。编程异常通常也被叫做软中断（software interrupt）。这样的异常有两种常用的用途：执行系统调用，给调试程序通报一个特定的事件（参见第八章）。

中断和异常向量

每个中断和异常是由0~255之间的一个数来标识。因为一些未知的原因，Intel把这个8位的无符号整数叫做一个向量。不可屏蔽中断的向量和异常的向量是固定的，而可屏蔽中断的向量可以通过对中断控制器的编程来改变（参见下一节）。

Linux利用了下列向量：

- 从0~31的向量对应于异常和不可屏蔽中断。
- 从32~47的向量被分配给可屏蔽中断，即由IRQ引起的中断。
- 剩余的从48~255的向量用来标识软中断。Linux只用了其中的一个，即128或0x80向量，用来实现系统调用。当用户态下的进程执行一条int 0x80汇编指令时，CPU切换到内核态，并开始执行system_call()内核函数（参见第八章）。

IRQ和中断

每个能够发出中断请求的硬件设备控制器都有一个指派为*IRQ*（Interrupt ReQuest）的输出线。所有现有的IRQ线都与一个叫做中断控制器的硬件电路的输入引脚相连，中断控制器执行下列动作：

1. 监视IRQ线，检查产生的信号。
2. 如果在IRQ线上产生了一个信号：
   a. 把接收到的信号转换成一个对应的向量。
   b. 把这个向量存放在中断控制器的一个I/O端口，从而允许CPU通过数据总线读此向量。
   c. 把产生的信号发送到处理器的INTR引脚，即发出一个中断。
   d. 等待，直到CPU确认这个中断信号，把它写进可编程中断控制器（PIC）的其中一个I/O端口，同时，清INTR线。
3. 返回到第一步。

IRQ线是从0开始顺序编号的，因此，第一条IRQ线通常被表示成IRQ0。与IRQ*n*关联的Intel的缺省向量是*n*+32。如前所述，IRQ和向量之间的映射可以通过向中断控制器端口发布合适的指令来修改。

图4-1显示了两片Intel 8259A PIC“级连”的一种典型连接，两片PIC可以处理至多15种不同的IRQ输入线。注意，第二片PIC的INT输出线与第一片PIC的IRQ2相连，在IRQ2线上的一个信号表示这样的事实，即在IRQ8 ~ IRQ15中的某一条线产生了一个信号。可使用的IRQ线的数目按惯例被限制在15，然而，越来越多新的PIC芯片能够处理更多的输入线。

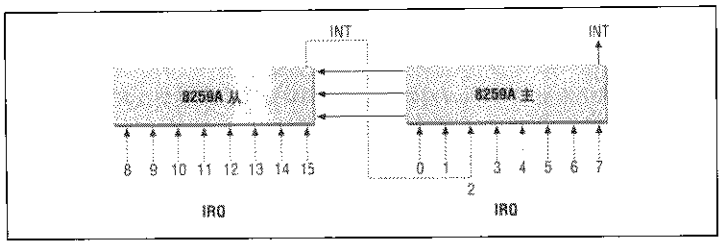


图4-1 两片8259A PIC的级连连接

在图中没有显示出来的其他线把PIC连接到总线。特别地，双向线D0-D7把I/O端口连到数据总线，而另一个输入线被连接到控制总线，并用来接收来自CPU的应答信号。

因为可用IRQ线的数量是有限的，有必要在几个不同的I/O设备之间共享同一条线。当这种情况发生时，连接到同一条线上的所有设备必须被顺序地选中。为了确定它们中的哪个发出了一个中断请求，我们将在“中断处理”一节中描述Linux是如何处理这种硬件限制的。

可以有选择地禁用每个IRQ线。因此，可以对PIC编程来禁用（disable）IRQ，也就是说，可以告诉PIC停止对给定的IRQ线发出中断，或者反之，启用（enable）它们。被禁用的中断是丢失不了的，它们一旦变为允许的，PIC就又把被禁用的中断发送到CPU。这个特点被大多数中断处理程序使用，因为这允许中断处理程序连续地处理同一类型的IRQ。

有选择地启用/禁用IRQ线不同于可屏蔽中断的全局屏蔽/不屏蔽。当eflags寄存器的IF标志被清除时，由PIC发布的可屏蔽中断被CPU临时忽略。cli和sti汇编指令分别清除和设置该标志。

异常

Intel 80x86微处理器发布了大约20种不同的异常（注1）。内核必须为每种异常提

注1： 精确数字依赖于处理器模式。

供一个专门的处理程序。对于某些异常，在开始执行异常处理程序前CPU控制单元会同时产生一个硬件错误码、并且压入内核堆栈。

在下面的列表中给出了Pentium模型中的异常的向量、名字、类型及其简单描述。更多的信息可以在Intel的技术文档中找到。

0 - "Divide error"（故障）

当一个程序试图被0除时产生。

1- "Debug"（陷阱或故障）

产生于：(1)设置eflags的T标志时（对于实现逐步执行所调试程序是相当有用的），(2)一条指令或操作数的地址落在一个活动debug寄存器的范围之内（参见第三章的"硬件上下文"一节）。

2 - 未用

为不可屏蔽中断保留（利用NMI引脚的那些中断）。

3 - "Breakpoint"（陷阱）

由int3（断点）指令（通常由debugger插入）引起。

4 - "Overflow"（陷阱）

当设置eflags的OF（overflow）标志时，into（check for overflow）指令被执行。

5 - "Bounds check"（故障）

对于有效地址范围之外的操作数，bound（check on address bound）指令被执行。

6 - "Invalid opcode"（故障）

CPU执行单元检测到一个无效的操作码（决定执行操作的机器指令部分）。

7 - "Device not available"（故障）

随着cr0的TS标志被设置，ESCAPE或MMX指令被执行（参见第三章的"保存浮点寄存器"一节）。

8 - "Double fault"（故障）

正常情况下，当CPU正试图为前一个异常来调用处理程序时，同时又检测到一个异常，两个异常能被串行地处理。然而，在少数情况下，处理器不能串行地处理它们，因而产生这种异常。

9 - "Coprocessor segment overrun"（故障）

因外部的数学协处理器引起的问题（仅用在80386微处理器）。

10 - "Invalid TSS"（故障）

CPU试图让一个上下文切换到有无效的TSS的进程。

11 - "Segment not present" (fault)

引用一个不存在的内存段（段描述符的Segment-Present标志被清0）。

12 - "Stack segment"（故障）

试图超过栈段界限的指令，或者由ss标识的段不在内存。

13 - "General protection"（故障）

违反了Intel 80x86保护模式下的保护规则之一。

14 - "Page fault"（故障）

寻址的页不在内存，相应的页表项为空，或者违反了一种分页保护机制。

15 - 由Intel保留。

16- "Floating point error"（故障）

集成到CPU芯片中的浮点单元用信号通知一个错误情形，如数字溢出，或被0除。

17 - "Alignment check"（故障）

操作数的地址没有被正确地排列（例如，一个长整数的地址不是4的倍数）。

18 ~ 31

这些值由Intel保留，为将来的扩充用。

如表4-1所示，每个异常都由专门的异常处理程序来处理（参见本章后面的"异常处理"一节），它们通常把一个Unix信号发送到引起异常的进程。

表4-1 由异常处理程序发送的信号

编号	异常	异常处理程序	信号
0	"Divide error"	divide_error()	SIGFPE
1	"Debug"	debug()	SIGTRAP
2	NMI	nmi()	None
3	"Breakpoint"	int3()	SIGTRAP

表 4-1 由异常处理程序发送的信号（续）

编号	异常	异常处理程序	信号
4	"Overflow"	overflow()	SIGSEGV
5	"Bounds check"	bounds()	SIGSEGV
6	"Invalid opcode"	invalid_op()	SIGILL
7	"Device not available"	device_not_available()	SIGSEGV
8	"Double fault"	double_fault()	SIGSEGV
9	"Coprocessor segment overrun"	coprocessor_segment_overrun()	SIGFPE
10	"Invalid TSS"	invalid_tss()	SIGSEGV
11	"Segment not present"	segment_not_present()	SIGBUS
12	"Stack exception"	stack_segment()	SIGBUS
13	"General protection"	general_protection()	SIGSEGV
14	"Page fault"	page_fault()	SIGSEGV
15	Intel reserved	None	None
16	"Floating point error"	coprocessor_error()	SIGFPE
17	"Alignment check"	alignment_check()	SIGSEGV

中断描述符表

中断描述符表（Interrupt Descriptor Table，IDT）是一个系统表，它与每一个中断或异常向量相联系，每一个向量有相应的中断或异常处理程序的入口地址。在内核允许中断发生前，必须适当地初始化 IDT。

在第二章中，我们介绍了GDT和LDT，IDT的格式与这两种表的格式非常相似，表中的每一项对应一个中断或异常向量，每个向量由 8 个字节组成。因此，最多需要 256 × 8=2048 字节来存放 IDT。

idtr CPU 寄存器允许 IDT 位于内存的任何地方，它指定了 IDT 的物理基地址及其限制（最大长度）。在允许中断之前，必须用 lidt 汇编指令初始化 idtr。

IDT 包含了三种类型的描述符，图 4-2 显示了每种描述符中的 64 位的含义。尤其值得注意的是，在编码 40~43 位的 Type 域的值标识了描述符的类型。

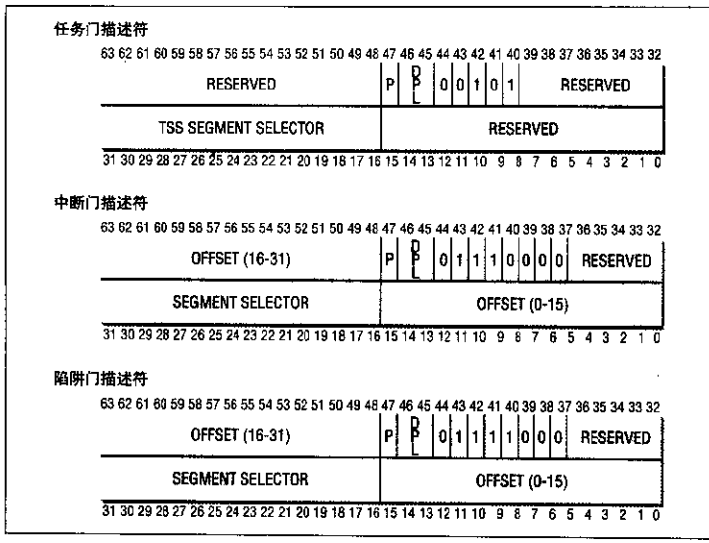


图4-2 门描述符的格式

这些描述符是:

任务门 (Task gate)

包含了一个进程的TSS段选择符（sgment selector），当中断信号发生时，它被用来取代当前进程的那个TSS段选择符。Linux并没有使用任务门。

中断门 (Interrupt gate)

中断门包含了段选择符和一个中断或异常处理程序的段内偏移量。当控制器转移到一个适当的段时，处理器清IF标志，从而关闭接下来会发生的可屏蔽中断。

陷阱门 (Trap gate)

与中断门相似，除非把控制权传递到一个适当的段，否则处理器不修改IF标志。

正如我们将在“中断，陷阱及系统门”一节中所看到的那样，Linux利用中断门处理中断，利用陷阱门处理异常。

中断和异常的硬件处理

我们现在描述CPU控制单元如何处理中断和异常。我们假定内核已被初始化，因此，CPU在保护模式下运行。

当执行了一条指令后，cs和eip这对寄存器包含了下一条将要执行的指令的逻辑地址。在处理这条指令之前，控制单元会检查在运行前一条指令时是否已经发生了一个中断或异常。如果发生了一个中断或异常，那么控制单元将：

1. 确定与中断或异常关联的向量 i（$0 \le i \le 255$）。
2. 读由idtr寄存器指向的IDT表中的第 i 项（在下面的描述中，我们假定IDT表项中包含的是一个中断门或一个陷阱门）。
3. 从gdtr寄存器获得GDT的基地址，并在GDT中查找，以读取在IDT表项中的选择符所标识的段描述符。这个描述符指定了中断或异常处理程序所在段的基地址。
4. 确信中断是由被授权的（中断）发生源发出的。首先将当前特权级CPL（存放在cs寄存器的低两位）与段描述符（存放在GDT中）的描述符特权级DPL比较，如果CPL小于DPL，就产生一个“一般性保护”异常，因为中断处理程序的特权不能低于引起中断的程序的特权。对于编程异常，则做进一步的安全检查：比较CPL与处于IDT中的门描述符的DPL，如果DPL小于CPL，就产生一个“一般性保护”异常。这最后一个检查可以避免以下情况的发生：用户应用程序访问特殊的陷阱门或中断门。
5. 检查是否发生了特权级的变化，也就是说，CPL是否不同于所选择的段描述符的DPL。如果是，控制单元必须开始使用与新的特权级相关的栈。通过执行以下步骤来做到这点：
   a. 读tr寄存器，以访问当前进程的TSS段。
   b. 用与新特权级相关的栈段和栈指针的正确值装载ss和esp寄存器。这些值可以在TSS中找到（参见第三章的“任务状态段”一节）。

c. 在新的栈中保存ss和esp以前的值，这些值定义了与旧特权级相关的栈的逻辑地址。

6. 如果故障已发生，用引起异常的指令地址装载cs和eip寄存器，从而使得这条指令能再次被执行。

7. 在栈中保存eflag、cs及eip的内容。

8. 如果异常产生了一个硬件错误码，则将它保存在栈中。

9. 装载cs和eip寄存器，其值分别是IDT表中第i项门描述符的段选择符和偏移量域。这些值给出了中断或者异常处理程序的第一条指令的逻辑地址。

控制单元所执行的最后一步就是跳转到中断或者异常处理程序。换句话说，控制单元处理完中断信号后，所执行的指令就是被选中的中断处理程序的第一条指令。

中断或异常被处理完后，相应的处理程序必须产生一条iret指令，把控制权转交给被中断的进程，这将迫使控制单元：

1. 用保存在栈中的值装载cs、eip、或eflag寄存器。如果一个硬件错误码曾被压入栈中，并且在eip内容的上面，那么，执行iret指令前必须先弹出这个硬件错误码。

2. 检查处理程序的CPL是否等于cs中最低两位的值（这意味着被中断的进程与处理程序运行在同一特权级）。如果是，iret终止执行；否则，转入下一步。

3. 从栈中装载ss和esp寄存器，因此，返回到与旧特权级相关的栈。

4. 检查ds、es、fs及gs段寄存器的内容，如果其中一个寄存器包含的选择符是一个段描述符，并且其DPL值小于CPL，那么，清相应的段寄存器。控制单元这么做是为了禁止用户态的程序（CPL=3）利用内核以前所用的段寄存器（DPL=0）。如果不清这些寄存器，怀有恶意的用户程序就可能利用它们来访问内核地址空间。

中断和异常处理程序的嵌套执行

一条内核控制路径由运行在内核态的指令序列组成，这些指令处理一个中断或一个异常。例如，当进程发出一个系统调用的请求时，相应的内核控制路径的第一部分

指令就是那些把寄存器的内容保存在内核堆栈的指令，而最后一部分指令就是恢复寄存器内容并让CPU返回到用户态的那些指令。

假定内核没有bug，大多数异常就只会在CPU处于用户态时发生。事实上，异常要么是由编程错误引起，要么是由调试程序触发。然而，“缺页”异常可以发生在内核，即当进程试图对属于它地址空间的页进行寻址，而该页现在不在RAM中时发生。当处理这样的一个异常时，内核可以挂起当前进程，并用另一个进程代替它，直到请求的页可以使用。只要被挂起的进程又获得处理器，处理缺页异常的内核控制路径就将恢复执行。

因为“缺页”异常处理程序从不进一步引起异常，至多与异常相关的两个内核控制路径会堆叠在一起（一个在另一个的上面）。

与异常形成对照的是，由I/O设备所产生的中断并不涉及对当前进程而言特殊的数据结构，尽管处理中断的内核控制路径代表当前进程运行。事实上，当一个给定的中断发生时，无法预测哪个进程当前在运行。

当CPU正在执行一个与中断有关的内核控制路径时，Linux的设计不允许发生进程切换。但是，这样的内核控制路径可以随意地嵌套，即一个中断处理程序可以由另一个中断处理程序进行中断，如此等等。

一个中断处理程序也可以推迟一个异常处理程序的执行，反之则不然。在内核态能触发的唯一异常就是刚刚描述的“缺页”。但是，中断处理程序从不执行可能导致缺页的操作，因此也就意味着，从不执行进程切换操作。

基于两个主要的原因，Linux交错执行内核控制路径：

- 为了提高可编程中断控制器和设备控制器的吞吐量。假定设备控制器在一条IRQ线产生了一个信号，PIC把这个信号转换成一个INTR请求，然后PIC和设备控制器保持阻塞，一直到PIC从CPU接收到一条应答信息。而正是由于内核控制路径的交错执行，内核才能在处理前一个中断的同时，发送应答。
- 为了实现一种没有优先级的中断模型。因为每个中断处理程序可以被另一个中断处理程序延缓，因此，在硬件设备之间没必要建立预定义优先级。这就简化了内核代码，提高了内核的可移植性。

初始化中断描述符表

现在，你知道了Intel微处理器在硬件级对中断和异常做了些什么，接下来，我们可以继续描述如何初始化中断描述符表。

回忆一下，内核启用中断以前，必须把IDT表的初始地址装到idtr寄存器，并初始化表中的每一项。这个动作是在初始化系统时完成的（参见附录一）。

int指令允许用户进程发出一个中断信号，其值可以是0~255的任意一个向量。因此，为了防止用户通过int指令模拟非法的中断和异常，IDT的初始化必须非常小心。这可以通过把中断或陷阱门描述符的DPL域设置成0来实现。如果进程试图发出这样的一个中断信号，控制单元将检查出CPL的值与DPL域有冲突，并且发布一个“一般保护性”异常。

然而，在少数情况下，用户进程必须能发出一个编程异常。为了做到这点，只要把中断或陷阱门描述符的DPL域设置成3，即特权级尽可能一样高。

现在，让我们来看一下Linux是如何实现这种策略的。

中断、陷阱及系统门

与在前面“中断描述符表”中所提到的一样，Intel提供了三种类型的中断描述符：任务门、中断门及陷阱门描述符。任务门描述符与Linux无关，但是它的中断描述符表包含了几个中断和陷阱门描述符。Linux使用与Intel稍有不同的细目分类和术语学，把它们按如下进行分类：

中断门（Interrupt gate）

用户态的进程不能访问的Intel的中断门（门的DPL域为0）。所有的中断处理程序都由中断门激活，并全部限制在内核态。

系统门（System gate）

用户态的进程可以访问的Intel的陷阱门（门的DPL域为3）。通过系统门来激活四个Linux异常处理程序，它们的向量是3，4，5及128，因此，在用户态下，可以发布int3、into、bound及int 0x80四条汇编指令。

陷阱门（Trap gate）

用户态的进程不能访问Intel的陷阱门（门的DPL域为0）。除了前一段所描述的四个Linux异常处理程序，其他所有的异常处理程序都通过陷阱门来激活。

下列函数用来给IDT插入门：

`set_intr_gate(n,addr)`

在第*n*个IDT表项中，插入一个中断门。这个门中的段描述符被设置成内核代码段的选择符，偏移域设置成addr，addr是中断处理程序的地址。把DPL域设置成0。

`set_system_gate(n,addr)`

在第*n*个IDT项中，插入一个陷阱门。这个门中的段描述符被设置成内核代码段的选择符，偏移域设置成addr，addr是异常处理程序的地址。把DPL域设置成3。

`set_trap_gate(n,addr)`

与前面的函数相似，只不过DPL的域被设置成0。

IDT的初步初始化

当计算机还运行在实模式时，IDT被初始化并由BIOS例程使用。然而，一旦Linux接管，IDT就被移到RAM的另一个区域，并进行第二次初始化，因为Linux没有利用任何BIOS例程（参见附录一）。

IDT被存放在idt_table表中，有256个表项（注2）。6字节的idt_descr变量指定了IDT的大小和它的地址，只有当内核用lidt汇编指令初始化idtr寄存器时，才用到这个变量。在其他所有的情况下，内核对idt变量的引用是为了获得IDT的地址。

在内核初始化过程中，由汇编语言写成的setup_idt()函数首先用同一个中断门[即指向ignore_int()的中断处理程序]来填满所有这256个idt-table表项：

注2：一些Pentium模式有声名狼藉的"f00f" bug，能让用户态程序冻结系统。当Linux在这样的CPU上执行时，就使用一个工作区，这个工作区基于在写保护页框中存放的IDT。当用户编译内核时，给这种bug提供这样的工作区作为一个可选项。

```
setup_idt:
    lea ignore_int, %edx
    movl $(__KERNEL_CS << 16), %eax
    movw %dx, %ax        /* selector = 0x0010 = cs */
    movw $0x8e00, %dx    /* interrupt gate, dpl=0, present */
    lea idt_table, %edi
    mov $256, %ecx
rp_sidt:
    movl %eax, (%edi)
    movl %edx, 4(%edi)
    addl $8, %edi
    dec %ecx
    jne rp_sidt
    ret
```

用汇编语言写成的 ignore_int() 中断处理程序，可以被看作一个空的处理程序，它执行下列动作：

1. 在栈中保存一些寄存器的内容。
2. 调用 printk() 函数打印“Unknown interrupt”系统消息。
3. 从栈恢复寄存器的内容。
4. 执行 iret 指令以恢复被中断的程序。

ignore_int() 处理程序应该从不被执行，在控制台或日志文件中出现的“Unknown interrupt”消息标志着要么是出现了一个硬件的问题（一个I/O设备正在产生没有预料到的中断），要么就是出现了一个内核的问题（一个中断或异常未被适当地处理）。

紧接着这个预初始化，内核将在IDT中进行第二遍初始化，用有意义的陷阱和中断处理程序替换这个空处理程序。一旦这个过程完成了，对由控制单元产生的每一个不同的异常，IDT都有一个专门的陷阱或系统门，而对于可编程中断控制器确认的每一个IRQ，IDT都将包含一个专门的中断门。

在接下来的两节中，将分别针对异常和中断来详细地说明这个工作是如何完成的。

异常处理

Linux 利用异常来达到两个差别很大的目标：

- 向进程发一个信号以通报一个反常情况。
- 处理请求分页。

对第一种情况，可以举一个例子。比如，如果进程执行了一个被0除的操作，CPU产生一个“除法错误”的异常，并由相应的异常处理程序向当前进程发送一个SIGFPE信号，这个进程将采取若干必要的步骤来（从错误中）恢复或者终止运行（如果没有为这个信号设置处理程序的话）。

异常处理程序有一个标准的结构，由以下三部分组成：

1. 在内核堆栈中保存大多数寄存器的内容（这部分用汇编语言实现）。
2. 用高级的C函数处理异常。
3. 通过ret_from_exception()函数从高级语言退出。

为了利用异常，必须对IDT进行适当的初始化，使得每个被确认的异常都有一个异常处理程序。trap_init()函数的工作是将一些最终值（即处理异常的函数）插入到IDT的一些表项中，这些表项与不可屏蔽中断及异常有关。这是由宏set_trap_gate和set_system_gate完成的：

```
set_trap_gate(0,&divide_error);
set_trap_gate(1,&debug);
set_trap_gate(2,&nmi);
set_system_gate(3,&int3);
set_system_gate(4,&overflow);
set_system_gate(5,&bounds);
set_trap_gate(6,&invalid_op);
set_trap_gate(7,&device_not_available);
set_trap_gate(8,&double_fault);
set_trap_gate(9,&coprocessor_segment_overrun);
set_trap_gate(10,&invalid_TSS);
set_trap_gate(11,&segment_not_present);
set_trap_gate(12,&stack_segment);
set_trap_gate(13,&general_protection);
set_trap_gate(14,&page_fault);
set_trap_gate(16,&coprocessor_error);
set_trap_gate(17,&alignment_check);
set_system_gate(0x80,&system_call);
```

现在，我们看一下一个典型的异常处理程序一旦被调用都做些什么。

为异常处理程序保存寄存器的值

让我们用handler_name来表示一个通用的异常处理程序的名字。(所有异常处理程序的实际名字都出现在前一部分的宏列表中。)每一个异常处理程序开始于下列的汇编指令:

```
handler_name:
   pushl $0 /* only for some exceptions */
   pushl $do_handler_name
   jmp error_code
```

当异常发生时，如果控制单元没有自动地把一个硬件错误代码插入到栈中，相应的汇编语言片段会包含一条pushl $0指令，在栈中垫上一个空值。然后，把高级C函数的地址压进栈中，它的名字由异常处理程序名与do_前缀组成。

标号为error_code的汇编语言片段对所有的异常处理程序都是相同的，除了“设备不可用”这一个异常(参见第二章的“保存浮点寄存器”一节)。这段代码执行一下步骤:

1. 把高级C函数可能用到的寄存器保存在栈中。
2. 产生一条cld指令来清eflag的方向标志DF，以确保调用字符串指令时会自动增加edi和esi寄存器的值。
3. 把栈中位于esp+36处的硬错误码拷贝到eax中，给栈中同一位置存上值-1，正如我们将在第九章的“系统调用的重新执行”一节中所看到的那样，这个值用来把0x80异常与其他异常隔离开。
4. do_handler_name()这个C函数的地址保存在栈中的esp+32位置，把这个地址装到ecx寄存器中，然后，在栈的这个位置写入es的值。
5. 把内核数据段选择符装进ds和es寄存器，然后把ebx寄存器的值设置为当前进程描述符的地址(参见第三章中的“标识一个进程”一节)。
6. 把要传给C函数的参数保存在栈中，也就是异常硬错误码和栈中某个位置的地址，在这个位置上保存了用户态寄存器的内容。
7. 调用这个C函数，它的地址现在存在ecx中。

执行了最后一步后，被调用的函数将在栈的顶部找到:

- 返回地址（即C函数结束后将执行的指令地址）（参见下一节）。
- 所保存的用户态寄存器栈地址。
- 硬件错误码。

从异常处理程序返回

当执行异常处理的C函数终止时，控制权被转移到下列汇编语言片段：

```
addl $8, %esp
jmp ret_from_exception
```

这段代码从栈中弹出用户态下寄存器的值和硬错误码，然后执行一个jmp指令跳转到ret_from_exception()函数。这个函数将在后面的“从中断和异常返回”一节中进行描述。

调用异常处理程序

如前面解释的那样，执行异常处理程序的C函数名总是由do_前缀和处理程序名组成。其中的大部分函数把硬错误码和异常向量保存在当前进程的描述符中，然后，向当前进程发送一个适当的信号。代码如下：

```
current->tss.error_code = error_code;
current->tss.trap_no = vector;
force_sig(sig_number, current);
```

当ret_from_exception()函数被调用时，它检查当前进程是否接收到一个信号。如果是，要么由进程自己的信号处理程序（如果存在的话）来处理这个信号，要么由内核来处理。在后面这种情况中，内核一般会杀死这个进程。异常处理程序发送的信号已在表4-1中列出。

最后，处理程序调用die_if_kernel()或die_if_no_fixup()：

- die_if_kernel()函数检查异常是否发生在内核态，如果是，它会调用die()函数，die()函数在控制台打印CPU所有寄存器的内容，并通过调用do_exit()终止当前的进程（参见第十九章）。

- die_if_no_fixup()函数相似，但是在调用die()之前，它检查异常是否由于一个无效的系统调用参数产生的，如果得到的结论是肯定的，它会使用“fixup”方式，这将在第八章的“动态地址检查：修正代码”一节中描述。

内核使用两种异常来更有效地管理硬件资源。相应的处理程序也更复杂，因为异常没必要表示一种错误情况：

- “设备不可用”：与在第三章的“保存浮点寄存器”一节中所讨论的一样，用这个异常推迟装载浮点寄存器，直到最后可能的时刻。
- “缺页”：正如我们将在第七章的“缺页异常处理程序”一节中看到的那样，用这个异常来推迟把新页框分配给进程，直到最后可能的时刻。

中断处理

正如前面解释的那样，大多数异常的处理是仅仅给引起异常的进程发送一个Unix信号，要采取的操作因此被延迟，直到进程接收到这个信号。所以，内核能很快地处理异常。

这种方法并不适合中断，因为与中断相关的进程（如请求传送数据的进程）已被挂起，且一个完全无关的进程正在运行，常常很久以后这些中断才到达。所以，给当前进程发送一个Unix信号是毫无意义的。

此外，由于硬件的限制，几个设备也许需要共享同一个IRQ线（回想一下PC只提供了几个IRQ）。这就意味着单单中断向量并不能提供中断产生的所有信息，例如，一些PC配置可以把同一个向量分配给网卡和图形卡。因此，一个中断处理程序必须足够灵活以为几个设备服务。为了做到这一点，几个中断服务例程（Interrupt Service Routine，ISR）要能与同一个中断处理程序相关联。其中每个ISR是一个函数，这个函数与共享这个IRQ线的单个设备相关。因为不可能预先知道哪一个特定的设备发出了IRQ，因此，执行每个ISR以检查相应的设备是否需要关注，如果是，当这个设备产生中断时，ISR执行必需执行的所有操作。

当一个中断发生时，并不是所有的操作都具有相同的急迫性。事实上，把所有的操作都放进中断处理程序本身并不合适。需要时间长的、非重要的操作应该推后，因

为当一个中断处理程序正在运行时，相应的IRQ线上再发出的信号就被忽略。更重要的是，中断处理程序是代表进程执行的，它所代表的进程必需总处于TASK_RUNNING状态，否则，就可能出现系统僵死情形。因此，中断处理程序不能执行任何阻塞过程，如I/O设备操作。因此，Linux把中断后面要执行的操作分为三类：

紧急的（Critical）

这样的操作诸如：向PIC应答一个中断，对PIC或设备控制器重编程，或者修改由设备和处理器同时访问的数据结构。这些都能被很快地执行，而之所以说它们是紧急的是因为它们必须被尽快地执行。紧急操作要在一个中断处理程序内立即执行，而且是在关中断的状态下。

非紧急的（Noncritical）

这样的操作如，修改那些只有处理器才会访问的数据结构（例如，按下一个键后，读扫描码）。这些操作也要很快地完成，因此，它们由中断处理程序立即执行，但在开中断的状态下。

非紧急可延迟的（Noncritical deferrable）

这样的操作如，把一个缓冲区的内容拷贝到一些进程的地址空间（例如，把键盘行缓冲区的内容发送到终端处理程序的进程）。这些操作可能被延迟较长的时间间隔而不影响内核操作，有兴趣的进程会等待需要的数据。非紧急可延迟的操作由一些被称为“下半部分（bottom half）”的函数来执行。我们将在后面讨论“下半部分”。

所有的中断处理程序执行四个相同的基本操作：

1. 在内核态堆栈中保存IRQ的值和寄存器的内容。
2. 为正在给这个IRQ线服务的PIC发送一个应答，这将允许PIC进一步发出中断。
3. 执行共享这个IRQ的所有设备的中断服务例程（ISR）。
4. 跳到ret_from_intr()的地址后终止。

当发生一个中断时，需要几个描述符来表示IRQ线的状态和需要执行的函数。图4-3以示意图的方式展示了处理一个中断的硬件电路和软件函数。下面将讨论这些函数。

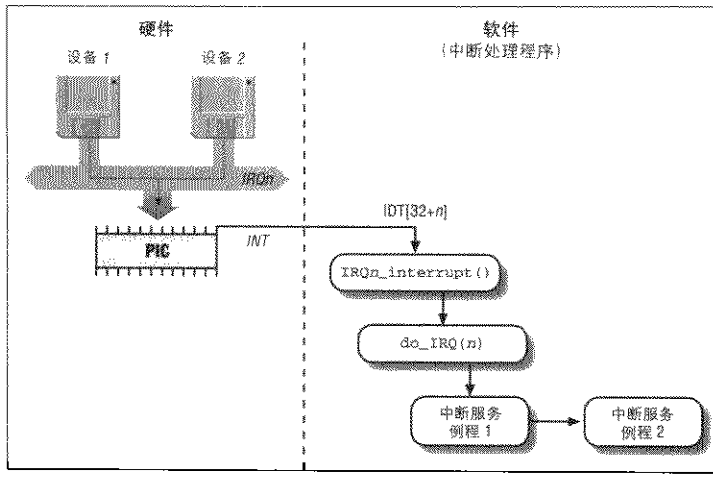


图 4-3 中断处理

中断向量

正如在前面的“IRQ和中断”一节中所解释的那样，给16个物理IRQ分配的向量是32~47。IBM PC兼容体系结构要求，一些设备必须被静态地连接到指定的IRQ线。尤其是：

- 间隔定时设备必须连到IRQ0线（参见第五章）。
- 从8259A PIC必须与IRQ2线相连（参见图4-1）。
- 必须把外部数学协处理器连接到IRQ13线（尽管最近的Intel 80x86处理器不再使用这样的设备，但Linux仍然支持历史悠久的80386模式）。

对剩余的所有IRQ，在允许中断前，内核必须建立IRQ号与I/O设备之间的对应，否则，内核在不知道哪个向量对应一个设备（如SCSI硬盘）的情况下，怎么能处理来自这个设备的信号呢？

现代I/O设备能把自己连接到几个IRQ线。最佳的选择依赖于系统有多少个设备及是否一些设备必须只响应特定的IRQ。对每个设备来说，选择一条线的方式有两种：

- 安装设备时执行一个通用程序。这样的程序可以让用户来选择一个可用的IRQ号，或者自己确定一个可用的IRQ号。
- 在系统启动时执行一个硬件协议。在这种系统下，外设宣布它们准备使用哪个中断线，然后协商一个最终的值以尽可能减少冲突。该过程一旦完成，每个中断处理程序可以利用一个能访问设备I/O端口的函数，来读取被赋予的IRQ。例如，遵循外设部件互连（Peripheral Component Interconnect，PCI）标准的设备的驱动程序利用一组函数，如`pci_read_config_byte()`和`pci_write_config_byte()`访问设备的配置空间。

在这两种情况下，当内核初始化相应的驱动程序时，能检索到一个设备挑选的IRQ线。表4-2显示了设备和IRQ之间一种相当随意的排列，你或许能在某个PC中找到同样的排列。

表4-2 把IRQ分配给I/O设备的一个例子

IRQ	INT	硬件设备
0	32	时钟
1	33	键盘
2	34	PIC级联
3	35	第二串口
4	36	第一串口
6	38	软盘
8	40	系统时钟
11	43	网络接口
12	44	PS/2鼠标
13	45	数学协处理器
14	46	EIDE磁盘控制器的一级链接
15	47	EIDE磁盘控制器的二级链接

IRQ 数据结构

与往常一样，当讨论到涉及状态转换的复杂操作时，数据结构有助于我们首先理解关键数据存放在什么地方。因此，这部分将解释支持中断处理的数据结构以及怎样把它们放在各种描述符中。图 4-4 示意性地显示了几个主要描述符之间的关系，这些描述符表示IRQ线的状态。(该图没有显示出处理“下半部分”所需的数据结构，后面将对它们进行讨论)。

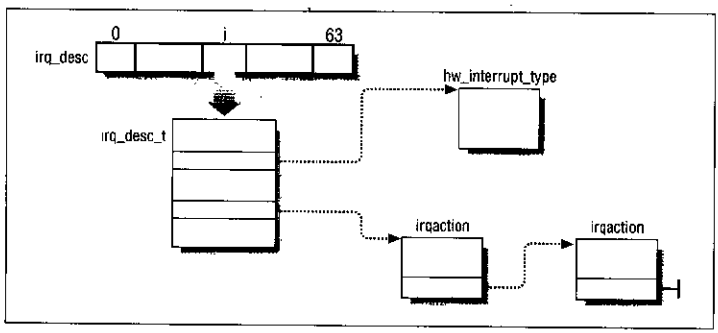


图 4-4 IRQ描述符

irq_desc_t 描述符

一个 irq_desc数组包含了 NR_IRQS 个 irq_desc_t描述符，这个描述符包含下列域：

status

描述 IRQ 线状态的一组标记。

IRQ_INPROGRESS

正在执行 IRQ 的一个处理程序。

IRQ _DISABLED

由一个设备驱动程序故意地禁用 IRQ 线。

IRQ_PENDING

一个IRQ已经出现在线上，这个IRQ的出现也已被应答，但是内核还没有为它提供服务。

IRQ_REPLAY

这条IRQ线已被禁用，但是前一个出现的IRQ还没有被应答。

IRQ_AUTODETECT

当执行一个硬件设备探测时，内核使用IRQ线。

IRQ_WAITING

当执行一个硬件设备探测时，内核使用IRQ线；此外，还没有产生相应的中断。

handler

指向hw_interrupt_type描述符，这个描述符描述为IRQ线提供服务的PIC电路。

action

指定当出现IRQ时被调用的中断服务例程。该域指向这个IRQ的irqaction描述符链表的第一个元素。在本章后面将会简单地描述一下irqaction描述符。

depth

如果启用IRQ线，则为0，如果禁用这个IRQ线不止一次，则为一个正数。每当disable_irq()函数被调用一次，它就增加这个域的值。如果depth等于0，这个函数就禁用这个IRQ线。相反，每当调用enable_irq()函数时，这个域的值就减少。如果depth变为0，这个函数就启用这个IRQ线。

在系统初始化期间，init_IRQ()函数把每个IRQ主描述符的status域设置成IRQ_DISABLED，代码如下：

```
for (i=0; i<NR_IRQS; i++)
    irq_desc[i].status = IRQ_DISABLED;
```

然后，更新IDT，用最终的中断门来代替临时使用的中断门。这是通过下列语句完成的：

```
for (i = 0; i < NR_IRQS; i++)
    set_intr_gate(0x20+i,interrupt[i]);
```

这段代码在interrupt数组中查找中断处理程序的地址，用这个地址建立中断门。IRQ*n*中断处理程序的名字是IRQ*n*_interrupt()(参见后面“为中断处理程序保存寄存器的值”一节)。

hw_interrupt_type 描述符

这个描述符包含一组指针，指向与特定PIC电路打交道的低级I/O例程。Linux除了支持本章前面已提到的8259A芯片外，也支持其他的几个PIC电路，如SMP IO-APIC、PIIX4的内部8259 PIC及SGI的Visual Workstation Cobalt (IO-)APIC。但是，为了简单起见，我们在本章假定，我们的计算机是有两片8259A PIC的单处理机，它提供16个标准的IRQ。在这种情况下，有16个irq_desc_t描述符，其中每个描述符的handler域指向i8259A_irq _type变量，这个变量描述8259A PIC。对其进行如下的初始化：

```
struct hw_interrupt_type i8259A_irq_type = {
    "XT-PIC",
    startup_8259A_irq,
    shutdown_8259A_irq,
    do_8259A_IRQ,
    enable_8259A_irq,
    disable_8259A_irq
};
```

在这个结构中的第一个域"XT-PIC"是一个名字。接下来，i8259A_irq_type包含的指针指向五个不同的函数，这些函数就是对PIC编程的函数。前两个函数分别启动和关闭这个芯片的IRQ线。但是，在使用8259A芯片的情况下，这两个函数的作用与后两个函数是一样的，后两个函数是启用和禁用IRQ线。在后面的“do_IRQ()函数”中将描述do_8259A_IRQ()函数。

irqaction 描述符

与前面描述的一样，多个设备能共享一个IRQ。因此，内核要维护多个irqaction描述符，其中的每个描述符涉及一个特定的硬件设备和一个特定的中断。这个描述符包含下列域。

Handler

指向一个I/O设备的中断服务例程。这是允许多个设备共享同一IRQ的关键域。

Flags

用一组标志描述IRQ与I/O设备之间的关系。

SA_INTERRUPT

处理程序必须以关中断执行。

SA_SHIRQ

设备允许它的IRQ线与其他设备共享。

SA_SAMPLE_RANDOM

可以把这个设备看作是随机事件发生源，因此，内核可以用它做随机数产生器。(用户可以从*/dev/random*和*/dev/urandom*设备文件中取得随机数而访问这种特征)

SA_PROBE

内核在执行硬件设备探测时正在使用的IRQ线。

Name

I/O设备名(通过读*/proc/interrupts*文件，在列出所服务的IRQ时也显示设备名)。

dev_id

指定I/O设备的主设备号和次设备号(参见第十三章中的“设备文件”一节)。

Next

指向irqaction描述符链表的下一个元素。链表中的元素涉及共享同一IRQ的硬件设备。

为中断处理程序保存寄存器的值

与其他上下文切换一样，需要保存寄存器这一点给内核开发者留下有点杂乱的编码工作，因为寄存器的保存和恢复必须用汇编语言代码，但是，在这些操作内部，又期望处理器从C函数调用和返回。在这一节，我们将描述处理寄存器的汇编语言任务，而下一节，我们将在随后调用的C函数中说明一些需要的技巧。

保存寄存器是中断处理程序做的第一件事情。如前所述，IRQ*n*中断处理程序的名字是IRQ*n*_interrupt，它的地址包含在中断门中，而中断门存放在IDT相应的表项中。

为了产生16个不同的中断处理程序入口点，同一个BUILD_IRQ宏被复制16次，每个IRQ号对应一次。每个BUILD_IRQ展开成下面的汇编语言片段：

```
IRQn_interrupt:
    pushl $n-256
    jmp common_interrupt
```

把中断号减256的结果保存在栈中（注3）；然后，当引用这个号时，可以对所有的中断处理程序都执行相同的代码。这种通用的代码可以在BUILD_COMMON_IRQ宏中找到，把这个宏可以展开成下列的汇编语言片段：

```
common_interrupt:
    SAVE_ALL
    call do_IRQ
    jmp ret_from_intr
```

SAVE_ALL宏依次展开成下列片段：

```
cld
push %es
push %ds
pushl %eax
pushl %ebp
pushl %edi
pushl %esi
pushl %edx
pushl %ecx
pushl %ebx
movl $_ _KERNEL_DS,%edx
mov %dx,%ds
mov %dx,%es
```

SAVE_ALL可以在栈中保存中断处理程序可能会使用的所有CPU寄存器，但除了eflags、cs、eip、ss及esp，因为这几个寄存器已经由控制单元自动保存了（参

注3：从IRQ号减256得到一个负数。正数保留给系统调用使用（参见第八章）。

见前面“中断和异常的硬件处理”一节)。然后，这个宏把内核数据段的选择符装到ds和es寄存器。

保存寄存器以后，BUILD_COMMON_IRQ宏调用do_IRQ()函数，并跳转到ret_from_intr()地址（参见后面的“从中断和异常返回”一节)。

do_IRQ()函数

调用do_IRQ()函数执行与一个中断相关的所有中断服务例程。当这个函数执行时，内核态堆栈从栈顶到栈底包含:

- do_IRQ()的返回地址
- 由SAVE_ALL推进栈中的一组寄存器的值
- IRQ号的编码
- 当控制单元识别出中断时自动保存的寄存器

因为C编译程序把所有这些参数都放在栈顶，所以do_IRQ()函数的声明如下:

```
void do_IRQ(struct pt_regs regs)
```

这里，pt_regs 结构由15个域组成:

- 前九个域与SAVE_ALL推进栈中的寄存器的值对应。
- 第十个域，通过引用一个叫做orig_eax的域，给IRQ号编码。
- 其余的域与控制单元自动推进栈中的寄存器值相对应（注4)。

因此，do_IRQ()函数读取作为参数传递过来的IRQ，并把它解码如下:

```
irq = regs.orig_eax & 0xff;
```

然后，执行:

```
irq_desc[irq].handler->handle(irq, &regs);
```

注4: pt_regs结构中不包括ret_from_intr()的返回地址，因为C编译器可以从栈顶获得返回地址，并在产生处理参数的指令时考虑进去。

handler域指向hw_interrupt_type描述符，这个描述符就是为IRQ线服务的PIC模型（参见前面“IRQ数据结构”一节）。假定那个PIC是一片8259A，handle域就指向do_8259A_IRQ()函数，这个函数由此被执行。

do_8259A_IRQ()函数首先调用mask_and_ack_8259A()函数，而后者对PIC应答这个中断，并禁止对同一个IRQ号的进一步中断。

接着do_8259A_IRQ()函数检查该中断处理程序是否愿意处理这个中断，是否已经在处理这个中断。最后，读存放在IRQ主描述符status域的RQ_DISABLED和IRQ_INPROGRESS标志值。如果两个标志都被清0，那么从action域挑选指向第一个irqaction描述符的指针，并设置IRQ_INPROGRESS标志。然后，调用handle_IRQ_event()，通过下列代码依次执行每个中断服务例程。如前所述，如果几个设备共享IRQ，每个相应的中断服务例程都必须被调用，因为内核并不知道哪个设备发出了中断：

```
do {
   action->handler(irq, action->dev_id, regs);
   action = action->next;
} while (action);
```

注意，只要还有一个中断服务例程没有执行，内核就不能跳出这个循环，因为同一IRQ线上的另一设备也许需要得到服务。

最后，do_8259A_IRQ()函数通过清除刚才提到的IRQ_INPROGRESS标志来清理所有事情。此外，如果没有设置IRQ_DISABLED标志，还得调用低级函数enable_8259A_irq()启用来自这个IRQ线的中断。

现在，控制权返回到do_IRQ()，do_IRQ()会检查“下半部分”任务是否等待执行。（我们会看到，这样的一个下半部分队列由内核维护）。如果下半部分在等待，就会调用我们将简短描述的do_bottom_half()函数。最后，do_IRQ()终止，并把控制权转移到ret_from_intr 地址。

中断服务例程

如前所述，一个中断服务例程（ISR）实现一种特定设备的操作。所有的中断服务例程都有相同的参数：

irq

IRQ号

dev_id

设备标识符

regs

指向内核态堆栈区域，栈中含有中断发生后随即保存的寄存器

第一个参数允许一个单独的ISR处理几条IRQ线，第二个参数允许一个单独的ISR照顾几个同类型的设备，第三个参数允许ISR访问被中断的内核控制路径的执行上下文。实际上，大多数ISR不使用这些参数。

当do_IRQ()函数调用一个ISR时，主IRQ描述符的SA_INTERRUPT标志决定是开中断还是关中断。可以用一条汇编指令将ISR所使用的中断状态转成其相反的状态：cli关中断，sti开中断。

ISR的结构依赖于所处理的设备的特点。我们将在第五章和第十三章给出几个ISR的例子。

下半部分

下半部分是一个低优先级的函数，通常与中断处理相关，也就是说，等待内核找到一个方便的时刻来运行它。下半部分一直会等到下面事件之一发生时才会被执行：

- 内核处理完一个系统调用。
- 内核处理完一个异常。
- 内核终止do_IRQ()函数，即处理完一个中断。
- 内核执行schedule()函数以选择一个新进程在CPU上运行。

因此，当中断服务例程激活一个下半部分时，会出现一个较长的事件间隔才执行这个下半部分（注5）。但是，正如我们已看到的，下半部分对于内核迅速完成多个设

注5：但是，下半部分的执行不会永远地延迟下去，直到没有下半部分执行时CPU才切换回用户态。参看后面的从“中断和异常返回”。

备的中断服务是至关重要的。这本书不谈及太多的下半部分的内容—因为它们依赖于设备服务所需要的特殊任务——而只涉及内核是怎样维护和调用下半部分的。在第五章的“TIMER_BH下半部分函数”一节中，你将找到一个特殊的下半部分的例子。

Linux利用一个叫做bh_base表的数组来把所有的下半部分组织在一起。bh_base是一个指向下半部分的指针数组，并能包含多达32个项，每一项就是一种下半部分。实际上，Linux只利用了其中大约一半。表4-3列出了这些类型。正如你从表中看到的那样，一些下半部分与硬件设备相关，而这些硬件设备未必装在系统中或是针对IBM PC兼容机之外的某些平台的。但是，TIMER_BH、CONSOLE_BH、TQUEUE_BH、SERIAL_BH、IMMEDIATE_BH及KEYBOARD_BH看来有着广泛的用途。

表4-3 Linux的下半部分

下半部分	外部设备
AURORA_BH	Aurora多端口卡（SPARC）
CM206_BH	CD-ROM Philips/LMS cm206磁盘
CONSOLE_BH	虚拟控制台
CYCLADES_BH	Cyclades Cyclom-Y串行多端口
DIGI_BH	DigiBoard PC/Xe
ESP_BH	Hayes ESP串行卡
IMMEDIATE_BH	立即任务队列
ISICOM_BH	MultiTech的ISI卡
JS_BH	游戏杆（PC IBM）
KEYBOARD_BH	键盘
MACSERIAL_BH	Power Macintosh的串行端口
NET_BH	网络接口
RISCOM8_BH	RISCom/8
SCSI_BH	SCSI接口
SERIAL_BH	串行接口
SPECIALIX_BH	Specialix IO8+
TIMER_BH	定时器
TQUEUE_BH	周期任务队列

激活和跟踪下半部分的状态

第一次调用下半部分前，必须首先对它进行初始化。这是通过调用 init_bh(n, routine)函数完成的，这个函数把例程的地址插入到 bh_base 的第 n 项。相反，remove_bh(n)从表中删除第 n 个下半部分。

一旦下半部分已进行过初始化、它就能被“激活”，因此，每当前面提到的一个事件发生时，就会执行下半部分。中断处理程序用 mark_bh(n)函数激活第 n 个下半部分。为了保持对所有这些下半部分的状态进行跟踪，在bh_active变量中存放了32个标志，这些标志指定当前被激活的下半部分。当一个下半部分结束了它的执行时，内核清 bh_active 相应的标志，因此，任何活动只能引起一次执行。

用 do_bottom_half()函数开始执行当前激活的、未屏蔽的下半部分。开中断后调用run_bottom_halves()函数。这个函数确保每次只有一个下半部分曾是激活的，这是通过执行下列C代码片断实现的：

```
active = bh_mask & bh_active;
bh_active &= active;
bh = bh_base;
do {
    if (active & 1)
        (*bh)();
    bh++;
    active >>= 1;
} while (active);
```

在bh_active中的标志涉及一组必须要执行的下半部分，把这些标志位清0，就确保每个下半部分激活后只引起相应的函数执行一次。

可以个别地“屏蔽”每个下半部分。在这种情况下，即使激活这个屏蔽的下半部分，也不能执行它。bh_mask 变量的 32 位分别指定哪些下半部分当前是屏蔽的。disable_bh(n)和enable_bh(n)函数作用于 bh_mask 的第 n 个标志，用它们分别对一个下半部分屏蔽和去掉屏蔽。

这里解释一下为什么屏蔽下半部分是有用的。假如当一个异常（如一个“缺页”）发生时，一个内核函数正在修改某个内核数据结构。当内核处理完这个异常以后，所有激活的、未屏蔽的下半部分都将被执行。如果其中之一与所挂起的那个内核函数访问同一个内核数据结构，那么，这个下半部分与那个内核函数都会发现这个数据

结构处于不一致状态。为了避免这种竞争条件（race condition），内核函数必须屏蔽所有访问这一数据结构的下半部分。

很不幸的是，bh_mask变量并不总能确保下半部分始终正确地被屏蔽。例如，让我们假定内核控制路径P1屏蔽某个下半部分B，然后，另一个内核控制路径P2中断了P1。P2一旦又屏蔽下半部分B，就执行它自己拥有的操作，然后在结束前清除对B的屏蔽。现在，P1恢复它的执行，而B已经成为未屏蔽的（不正确）。

因此，有必要使用计数器，而不是用一个简单的二进制标志跟踪屏蔽，并且增加一个新的表bb_mask_count，这个表的表项包含了每个下半部分的屏蔽层数。在对bh_mask的第n个标志操作之前，disable_bh(n)和enable_bh(n)函数会首先更新bh_mask_count[n]。

扩充下半部分

引入下半部分的目的是，允许一些与中断处理相关的有限个函数以推迟的方式得到执行。这种方法可以从两个方面得到延伸：

- 允许一个普通的内核函数，而不仅仅是服务于中断的一个函数，能以下半部分的身份运行。
- 允许几个内核函数，而不是单独的一个函数，能与一个下半部分相关联。

函数组用任务队列（task queue）表示，任务队列是元素为struct tq_struct的链表，其结构如下：

```
struct tq_struct {
    struct tq_struct *next;    /* 激活下半部分链表 */
    unsigned long sync;        /* 必须初始化为0 */
    void (*routine)(void *);   /* 要调用的函数 */
    void *data;                /* 参数 */
};
```

我们将在第十三章看到，当一个特定的中断发生时，I/O设备驱动程序充分利用任务队列来请求一些函数执行。

DECLARE_TASK_QUEUE宏用来分配一个新的任务队列，而queue_task()则是把一

个新函数插入到任务队列中。run_task_queue()执行给定任务队列中的所有函数。值得一提的是，有两个特殊的任务队列，每个都与一个特定的下半部分相关：

- 由IMMEDIATE_BH下半部分运行tq_immediate任务队列，该队列中包括要执行的内核函数和标准的下半部分。只要一个函数加入到tq_immediate队列，内核就激活IMMEDIATE_BH下半部分。
- 由TQUEUE_BH下半部分运行tq_timer任务队列，每个定时中断都激活TQUEUE_BH下半部分。我们将在第五章看到，这意味着大约每隔10ms这个队列运行一次。

IRQ线的动态处理

除了IRQ0、IRQ2及IRQ13，其余的13根IRQ都被动态地处理。因此存在一种方式，同一个中断可以让几个硬件设备使用，即使这些设备不允许共享IRQ。技巧就在于使这些硬件设备的活动串行化，以便一次只能有一个设备拥有这个IRQ线。

在激活一个准备利用IRQ线的设备之前，其相应的驱动程序调用request_irq()。这个函数建立一个新的irqaction描述符，并用参数值初始化它。然后调用setup_x86_irq()函数把这个描述符插入到合适的IRQ链表。如果setup_x86_irq()返回一个错误码，设备驱动程序终止操作，这意味着IRQ线已由另一个设备所使用，而这个设备不允许中断共享。当设备操作结束时，驱动程序调用free_irq()函数从IRQ链表中删除这个描述符，并释放相应的内存。

让我们用一个简单的例子看一下这种方案是怎么工作的。假定一个程序想对*/dev/fd0*设备文件（与第一个软盘对应的设备文件，注6）进行访问。程序要做到这点，可以通过直接访问*/dev/fd0*，也可以通过在系统上安装一个文件系统。通常将IRQ6分配给软盘控制器，给定这个号，软盘驱动程序就发出下列请求：

```
request_irq(6, floppy_interrupt,
            SA_INTERRUPT|SA_SAMPLE_RANDOM, "floppy", NULL);
```

我们可以观察到，floppy_interrupt()中断服务例程必须以关中断（设置SA_INTERRUPT）的方式来执行，并且不允许共享这个IRQ（清SA_SHIRQ标志）。当软

注6：软盘是通常不允许IRQ共享的“旧”设备。

盘的操作被终止时（要么终止对*/dev/fd0*的I/O操作，要么卸载这个文件系统），驱动程序就释放IRQ6：

```
free_irq(6, NULL);
```

为了把一个irqaction描述符插入到适当的链表中，内核调用setup_x86_irq()函数，传递给这个函数的参数为irq_nr（即IRQ号）和new（即刚才分配的irqaction描述符的地址）。这个函数将：

1. 检查另一个设备是否已经在用irq_nr这个IRQ，如果是，检查两个设备的irqaction描述符中的SA_SHIRQ标志是否都指定了IRQ线能被共享。如果不能使用这个IRQ线，则返回一个错误码。
2. 把*new（新irqaction描述符）加到由irq_desc[irq_nr]->action指向的链表的末尾。
3. 如果没有其他设备共享同一个IRQ，清*new的flags域的IRQ_DISABLED和IRQ_INPROGRESS标志，并重新对PIC编程以确保允许产生IRQ信号。

举一个如何使用setup_x86_irq()的例子，它是从系统初始化的代码中抽出的。通过执行time_init()函数中下面的指令，内核初始化间隔定时器设备的irq0描述符（参见第五章）。

```
struct irqaction irq0  =
    {timer_interrupt, SA_INTERRUPT, 0, "timer", NULL,};
setup_x86_irq(0, &irq0);
```

首先，初始化类型为irqaction的irq0变量：把handler域设置成timer_interrupt()函数的地址，flags域设置成SA_INTERRUPT，name域设置成"timer"，最后一个域设置成NULL以表示没有用dev_id值。接下来，内核调用setup_x86_irq()，把irq0插入到与IRQ0相关的irqaction描述符的链表中。

类似地，内核初始化与IRQ2和IRQ13相关的irqaction描述符，并把它们插入到irqaction描述符合适的链表中，这是通过执行init_IRQ()函数中下面的指令完成的：

```
struct irqaction irq2 =
    {no_action, 0, 0, "cascade", NULL,};
struct irqaction irq13 =
```

```
        {math_error_irq, 0, 0, "fpu", NULL,};
    setup_x86_irq(2, &irq2);
    setup_x86_irq(13, &irq13);
```

从中断和异常返回

我们通过考察中断和异常处理程序的终止阶段来结束本章。尽管终止阶段的主要目的很清楚，即恢复一些程序的执行，但是，在这样做之前，还需要考虑几个问题：

- 内核控制路径并发执行的数量：如果仅仅只有一个，那么CPU必须切换到用户态。
- 激活的、等待被执行的下半部分：如果有一些，必须执行它们。
- 挂起进程的切换请求：如果有任何请求，内核就必须执行进程调度；否则，把控制权还给当前进程。
- 挂起的信号：如果信号已经发送到当前进程，就必须处理它。

从技术上说，完成所有这些事情的内核汇编语言代码并不是一个函数，因为控制权从不返回到调用它的函数。它只是一个代码片段，有三个不同的入口点，分别叫做ret_from_intr、ret_from_sys_call和ret_from_exception。我们将以三个不同的函数提到它们，因为这使得描述变得简单。因此，我们常常以函数的形式提到下面这三个入口点：

ret_from_intr()
    终止中断处理程序。

ret_from_sys_call()
    终止系统调用，即由0x80异常引起的内核控制路径。

ret_from_exception()
    终止除了0x80的所有异常。

图4-5显示了相应的有三个入口点的大体流程图。除了这三个标记，还增加了其他几个标记，这就使得你更容易地把汇编语言代码与流程图联系起来。现在，让我们仔细地看一下在每种情况下终止是如何发生的。

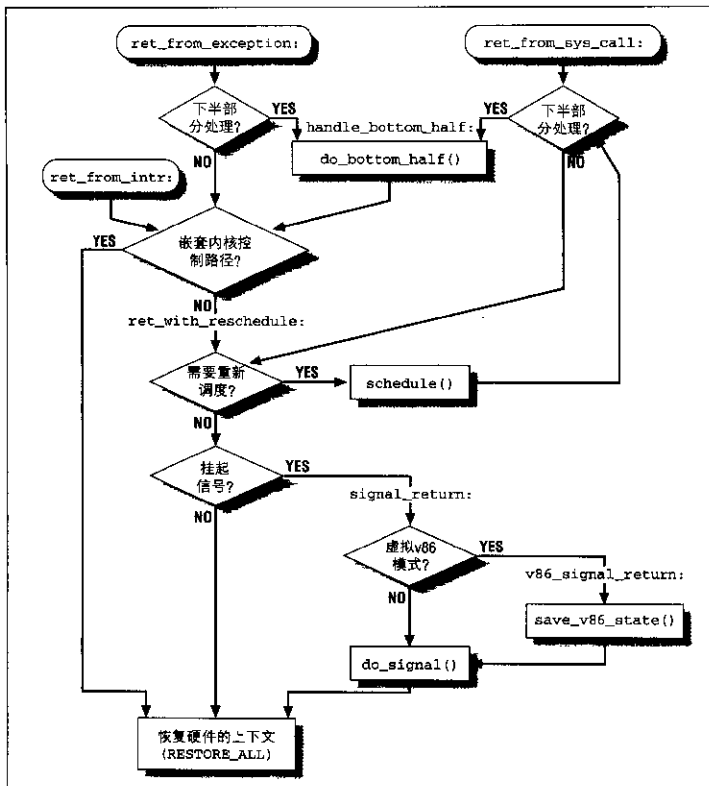


图 4-5 从中断和异常返回

ret_from_intr()函数

当ret_from_intr()被调用时，do_IRQ()函数已经执行了所有激活的下半部分[参见前面“do_IRQ()函数”一节]。ret_from_intr()函数的初始化部分是由下列代码实现的:

```
ret_from_intr:
  movl %esp, %ebx
  andl $0xffffe000, %ebx
  movl 0x30(%esp), %eax
  movb 0x2c(%esp), %al
  testl $(0x00020000 | 3), %eax
  jne ret_with_reschedule
  RESTORE_ALL
```

当前进程描述符的地址被保存在ebx中（参见第三章的“标识一个进程”一节）。接着函数使用在中断发生时压入栈中的cs及eflags寄存器的值，以决定被中断的程序是否正运行在内核态。如果是，就发生了一个中断嵌套，并且通过执行下列的代码（由RESTORE_ALL宏产生）、恢复被中断的内核控制路径：

```
popl %ebx
popl %ecx
popl %edx
popl %esi
popl %edi
popl %ebp
popl %eax
popl %ds
popl %es
addl $4,%esp
iret
```

这个宏把SAVE_ALL宏保存的值载入各个寄存器，并通过执行iret指令将控制权交给被中断的程序。

另一方面，如果被中断的程序正运行在用户态，或者如果设置了eflags的VM标志（注7），就跳转到ret_with_reschedule地址：

```
ret_with_reschedule:
  cmpl $0,20(%ebx)
  jne reschedule
  cmpl $0,8(%ebx)
  jne signal_return
  RESTORE_ALL
```

注7： 这个标志允许程序在Virtual-8086模式下执行。

如前所述，ebx寄存器指向当前进程的描述符。在这个描述符内，need_resched域在偏移量20处，由第一条cmpl指令检查。因此，如果need_resched域是1，schedule()函数被调用以执行进程切换。

sigpending域在进程描述符内的偏移量是18。如果这个域为空，current恢复到用户态下执行。否则，代码跳到signal_return处理current的挂起信号：

```
signal_return:
  sti
  testl $(0x00020000),0x30(%esp)
  movl %esp,%eax
  jne v86_signal_return
  xorl %edx,%edx
  call do_signal
  RESTORE_ALL
v86_signal_return:
  call save_v86_state
  movl %eax,%esp
  xorl %edx,%edx
  call do_signal
  RESTORE_ALL
```

如果被中断的进程处于VM86模式，save_v86_state()函数被调用。然后，调用do_signal()函数（参见第九章）处理挂起的信号。最后，current可以恢复到用户态下执行。

ret_from_sys_call()函数

ret_from_sys_call()函数等价于下列的汇编语言代码：

```
ret_from_sys_call:
  movl bh_mask, %eax
  andl bh_active, %eax
  je ret_with reschedule
handle_bottom_half:
  call do_bottom_half
  jmp ret_from_intr
```

首先，检查bh_mask和bh_active变量，以决定是否存在激活的、未屏蔽的下半部分。如果没有必须执行的下半部分，就跳转到ret_with_reschedule地址。否则，调用do_bottom_half()函数，然后把控制权转移到ret_from_intr。

ret_from_exception()函数

ret_from_exception()函数等价于下列汇编语言代码:

```
ret_from_exception:
  movl bh_mask,%eax
  andl bh_active,%eax
  jne handle_bottom_half
  jmp ret_from_intr
```

首先，检查bh_mask和bh_active全局变量，以决定是否存在激活的、未屏蔽的下半部分。如果是，执行它们。然后无条件跳转到ret_from_intr地址。因此，异常与中断以相同的方式终止。

对 Linux 2.4 的展望

Linux 2.4引入一种新的机制叫软中断。软中断类似于Linux 2.2的下半部分，也就是说它们也可用来推迟一个内核函数的执行。然而，下半部分被严格地串行执行(因为即使在不同的CPU上，也没有两个下半部分能被同时执行)，软中断却在任何时候都不需要串行化。同一软中断的两个实例完全有可能在两个CPU上同时运行。当然，在这种情况下，软中断必须是可重入（reentrant）的。尤其是网络部分可以从软中断中获得极大的好处：在多处理机上，它会变得更高效，因为可以使用两个下半部分来替代一个老的NET_BH的下半部分。

Linux 2.4还引入另一个类似于下半部分的机制叫做*tasklet*。tasklet建立在软中断之上，但是，对它们自己而言，又是被串行执行，两个CPU可以同时执行两个tasklet，但这两个tasklet必须不同。编写tasklet比编写一般的软中断更容易，因为它们不必是可重再入的。

下半部分在Linux 2.4中继续存在，但是它们现在建立在tasklet之上。一如既往，没有两个下半部分同时被执行，即使在一个多处理机系统的两个不同CPU上也不行。设备驱动程序的开发者将会更新他们旧的驱动程序，用tasklet代替下半部分，因为下半部分极大地降低了多处理机系统的性能。

从硬件方面来说，现在即使在单处理机系统上，Linux 2.4也支持IO-APIC芯片，并且能够在多处理机系统上处理几个外部的IO-APIC芯片。(将Linux移植到大型企业级系统上时需要这种功能。)

第五章
定时测量

计算机处理的很多活动都是由定时测量（timing measurement）来驱动的，这常常对用户是透明的。例如，当你停止使用计算机的控制台以后，屏幕会自动关闭，这得归因于定时器，它允许内核跟踪你按键或移动鼠标后到现在过了多少时间。如果你收到了一个来自系统的警告信息，希望你删除一组不用的文件，这是由于有一个程序能识别很长时间内都没有被访问的用户文件。为了做这些事情，程序必须能从每个文件中检索到一个时间标记(timestamp)，这个标记识别文件的最后访问时间，因此，这样的一个时间标记必须由内核自动地设置。尤其值得注意的是，定时机制通过一些更基本的内核操作，如检查超时（time-out）来驱使进程切换。

我们可以辨别出内核所必需的两种主要的定时测量：

- 持续记录当前的时间和日期，以便能通过time()、ftime()和gettimeofday()系统调用返回给用户程序［见本章后面的“time()，ftime()及gettimeofday()系统调用”一节］，也可以用内核本身来作文件和网络包的时间标记。
- 维持定时器，这种机制能够提醒内核或用户程序某一时间间隔已经过去了［分别参见后面的“定时器的作用”一节和“setitimer()和alarm()系统调用”一节］。

定时测量是由几个硬件电路完成的，这些电路基于固定频率的振荡器和计数器。本章由三个不同的部分组成。第一部分描述定时基础的硬件设备；接下来的部分描述为测量时间而引入的内核数据结构和函数；最后一部分讨论与定时测量相关的系统调用及相应的服务例程。

硬时钟

内核必须显式地与三种时钟打交道：实时时钟（Real Time Clock，RTC）、时间标记计数器（Time Stamp Counter，TSC）及可编程间隔定时器（Programmable Interval Timer，PIT）。前两种硬件设备允许内核跟踪当前的时间；后一种设备由内核编程，以便它能以固定的、预先定义的频率发出中断。对于内核和用户程序使用的定时器来说，这样的周期性中断是至关重要的。

实时时钟

所有的PC都包含了一个叫实时时钟（RTC）的时钟，它是独立于CPU和所有其他芯片的。

即使当PC关掉电源，RTC还继续走，因为它靠一个小电池或蓄电池供电。CMOS RAM和RTC被集成在一个芯片上，即Motorola 146818或一个其他等价的芯片。

RTC能在IRQ8上发出周期性的中断，中断的频率在2 Hz～8192 Hz之间。也可以对它进行编程以使得当RTC到达某个特定的值时激活IRQ8线，也就是作为一个闹钟来工作。

Linux只用RTC来获得时间和日期，然而，通过作用于*/dev/rtc*设备文件，也允许进程对RTC编程（参见第十三章）。内核通过0x70和0x71 I/O端口存取RTC。通过执行*/sbin/clock*系统程序（它直接作用于这两个I/O端口），系统管理员可以配置时钟。

时间标记计数器

所有的Intel 80x86微处理器都包含一个CLK输入引线，它接收一个外部振荡器的时钟信号。

从Pentium开始，很多新近的Intel 80x86微处理器就都包含了一个64位的时间标记计数器（TSC）的寄存器，可以通过汇编语言指令`rdtsc`读这个寄存器。这个寄存器是一个计数器，它在每个时钟信号到来时加1，例如，如果时钟节拍的频率是400 MHz，那么，时间标记计数器每2.5纳秒增加一次。

与可编程间隔定时器传递来的时间测量相比，Linux利用这个寄存器可获得更精确的时间测量。为了做到这点，Linux在初始化系统的时候必须确定时钟信号的频率。事实上，因为编译内核时并没有声明这个频率，所以同一内核映像可以运行在用任何一个频率产生时钟节拍的CPU上。算出实际频率是由calibrate_tsc()函数在系统初始化期间完成，它返回的值是：

$\left\lfloor \frac{2^{32}}{f} \right\rfloor$，$f$=以MHz为单位的CPU的频率

f值是通过计算在一个相对长的时间间隔内所发生的时钟信号的个数而得到，也就是50.00077毫秒中时钟信号的个数而得到的。这个时间常量可通过适当地设置可编程间隔定时器的一些通道来产生（参见下一节）。calibrate_tsc()执行较长时间不会引起问题，因为只有在系统初始化期间才调用这个函数。

可编程间隔定时器

除了实时时钟和时间标记计数器，IBM PC兼容机还包含了第三种类型的时间测量设备，叫做可编程间隔定时器（PIT）。PIT的作用类似于微波炉的闹钟，即让用户意识到烹调的时间已经够了。所不同的是，这个设备不是通过振铃，而是发出一个特殊的中断，叫做定时中断（timer interrupt）来通知内核又一个时间间隔过去了（注1）。与闹钟的另一个区别是，PIT以某一固定的频率（由内核确定）不停地发出中断。每个IBM PC兼容机都至少包含了一个PIT，它一般是一个使用0x40~0x43 I/O端口的8254 CMOS芯片。

在下一节中我们将看到，Linux给PC的第一个PIT进行编程，使它以（大约）100 Hz的频率向IRQ0发出定时中断，即每10ms产生一次定时中断。这个时间间隔叫做一个节拍（tick），它的长度以微秒为单位存放在tick变量中。节拍为系统中的所有活动打拍子，从某种意义上说，它们象音乐家排练节目时节拍器发出的滴答声。

一般而言，短的节拍产生较好的系统响应。这是因为系统响应很大程度上取决于高优先级的进程一旦变成可运行的，它能以多快的速度抢占正在运行的进程（参见第十章），而内核一般在处理定时中断时检查正在运行的进程是否要被抢占。然而，这

注1：也用PIT来驱动连接到计算机内部扬声器的音频放大器。

是一种平衡：短的节拍需要CPU在内核态花费较多的时间，也就是在用户态花费较少的时间。结果是，用户程序的运行就慢了。因此，只有非常强大的机器才采用很短的节拍，并能承担随之而产生的系统开销。目前，只有Linux内核在康柏的Alpha的版本是每秒发出1024个定时中断，相当于大约1ms一个节拍。

在Linux的代码中，有几个宏产生的常量决定定时中断的频率：

- HZ产生每秒定时中断的个数，也就是定时中断的频率。在IBM PC及大多数其他硬件平台上，这个值被设置为100。
- CLOCK_TICK_RATE产生的值为1193180，这个值是8254芯片的内部振荡器频率。
- LATCH产生CLOCK_TICK_RATE和HZ的比值。这个值被用来对PIT编程。

第一个PIT由init_IRQ()进行如下的初始化：

```
outb_p(0x34,0x43);
outb_p(LATCH & 0xff , 0x40);
outb(LATCH >> 8 , 0x40);
```

outb() C函数等价于outb汇编语言指令：它把第一个操作数拷贝到由第二个操作数指定的I/O端口。outb_p()函数类似于outb()，除了它会通过一个空操作而产生一个暂停外。第一条outb_p()语句是对PIT的一条指令，使得PIT以新的频率发出中断。接下来的两条outb_p()和outb()语句，为设备提供新的中断频率。把16位LATCH常量作为两个连续的字节发送到设备的8位的0x40 I/O端口。最后的结果是，PIT将以（大约）100-Hz的频率发出定时中断，也就是说，每10 ms产生一次定时中断。

现在我们已经明白了硬件时钟都做了些什么，接下来一节将描述，当内核接收到一个定时中断时（即已过了一个节拍）所做的全部的工作。

定时中断处理程序

每一个定时中断的产生都触发下列主要的动作：

- 更新自系统启动以来所花费的时间。

- 更新时间和日期。
- 确定当前进程在CPU上已运行了多长时间，如果已经超过了分配给它的时间，则抢占它。时间片（也叫时限 *quantum*）将在第十章讨论。
- 更新资源使用统计数。
- 检查每个软定时器（参见后面“定时器的作用”一节）的时间间隔是否已到，如果是，则调用适当的函数。

考虑到第一个活动是紧迫的，因此，由定时中断处理程序自己来完成。其余的四个活动次之，可以通过TIMER_BH和TQUEUE_BH的下半部分（参见第四章的“下半部分”一节）调用的函数来完成。

内核利用两个基本的时间保持（timekeeping）函数：一个保持当前最新的时间，另一个计算在当前秒内走过的微秒数。有两种不同的方式来维持这些值：如果芯片有一个时间标记计数器（TSC），则可用一个更精确的方法，在其他情况下，使用精确性差一些的方法。因此，内核设置了两个变量来保存它所用的函数，如果TSC存在，就让变量指向使用TSC的函数。

- 如果CPU有TSC寄存器，用do_gettimeofday()计算当前时间，否则，用do_normal_gettime()计算。在do_get_fast_time变量存放的指针指向合适的函数。
- 当TSC寄存器可用时，用do_fast_gettimeoffset()计算微秒数，否则，用do_slow_gettimeoffset()计算。这个函数的地址存放在do_gettimeoffset变量中。

在内核启动时运行的time_init()函数能将这些变量指向正确的函数，并设置IRQ0对应的中断门。

PIT中断服务例程

一旦IRQ0中断门已初始化，IRQ0 irqaction描述符的handler域就包含timer_interrupt()函数的地址。这个函数以关中断开始运行，因为IRQ0主描述符的status域设置了SA_INTERRUPT标志。它执行下列步骤：

1. 如果CPU有一个TSC寄存器，执行下列子步骤：
   a. 执行一条rdtsc汇编指令，在last_tsc_low变量中保存TSC寄存器的值。
   b. 读8254芯片内部振荡器的状态，并计算定时中断的发生与中断服务例程的执行之间的延迟（注2）。
   c. 在delay_at_last_interrupt变量中保存这个延迟（以微秒作单位）。
2. 调用do_timer_interrupt()。

可以认为do_timer_interrupt()对所有的80x86模型是通用的，它执行下列操作：

1. 调用do_timer()函数（很快会给出充分的解释）。
2. 如果已经调用了系统调用函数adjtimex()，那么这个函数每660秒会调用一次set_rtc_mmss()函数，也就是说，每11分钟调整一次实时时钟。这个特点有助于网络上的系统同步它们的时钟（参见后面的“adjtimex()系统调用”一节）。

以关中断运行的do_timer()函数必须被尽可能快地执行。因此，它只是简单地更新一个基本的值（系统自启动以来所用的时间），而把所有剩余的活动都委托给两个下半部分处理。这个函数涉及与定时测量相关的三个主要变量：第一个变量是刚刚提到的基本的正常运行时间，而后两个变量需要存放丢失的节拍，即下半部分有机会执行之前发生了多少个定时中断。因此，第一个变量是绝对值（仅仅保持递增），而另外两个则是相对值，相对于另一个变量xtime来存放当前时间的近似值。（这个变量将在后面“更新时间和日期”一节中描述）。

三个do_timer()的变量是：

jiffies

自系统启动以来的时钟节拍数目。内核初始化期间把它设置成0，当一个定时中断发生时也就是每个节拍增1（注3）。

注2：8254振荡器启动一个连续递减的计数器。当计数器的值变为0时，该芯发送一个IRQ0信号。该计数器可以记录中断已持续了多长时间。

注3：因为jiffies被作为一个32位的无符号整数存储，所以系统启动约497天后，该值自动返回为0。

lost_ticks

自从xtime最后更新以来发生的节拍数目。

lost_ticks_system

自从xtime最后更新以来，进程在内核态运行所发生的节拍数目。user_mode宏检查保存在栈中cs寄存器的CPL域，以确定进程是否运行在内核态。

do_timer()函数等价于：

```
void do_timer(struct pt_regs * regs)
{
    jiffies++;
    lost_ticks++;
    mark_bh(TIMER_BH);
    if (!user_mode(regs))
        lost_ticks_system++;
    if (tq_timer)
        mark_bh(TQUEUE_BH);
}
```

注意，只有tq_timer任务队列不为空时TQUEUE_BH下半部分才被激活（参见第四章的“下半部分”一节）。

TIMER_BH下半部分函数

timer_bh()函数与TIMER_BH下半部分相关联，它调用了update_times()，run_old_timers()和run_timer_list()三个辅助函数，下面对它们给予描述。

更新时间和日期

用户程序获得的当前时间和日期就是从类型为struct timeval的xtime变量中得到的。内核有时也涉及这个变量，例如，当更新inode的时间标记时（参见第一章中“文件描述符与索引节点”一节）。特别是，从1970年1月1日凌晨0点（注4）以来所过去的秒数存放在xtime.tv_sec中，最后一秒内已经过去的微秒数（它的取值范围为0~999999）存放在xtime.tv_usec。

注4： 传统上，这个日期由所有的Unix系统用作计算时间的最早时刻。

在系统初始化期间，time_init()被调用来设置时间和日期：通过调用get_cmos_time()函数，从实时时钟中读时间和日期，然后初始化xtime。这一任务一旦完成了，内核将再也不需要RTC，而是依靠TIMER_BH下半部分，它会在每个节拍都被激活。

TIMER_BH下半部分调用update_times()函数，该函数会以关中断来更新xtime，并执行下列语句：

```
if (lost_ticks)
    update_wall_time(lost_ticks);
```

update_wall_time()函数连续调用update_wall_time_one_tick()函数lost_ticks次，每次调用都给xtime.tv_usec域加10000（注5）。如果xtime.tv_usec变得大于999999，update_wall_time()函数也会更新xtime的tv_sec域。

更新资源使用统计数

用lost_ticks和lost_ticks_system的值一起来更新资源使用统计数。这些统计数由各种管理实用程序来使用，如top。用户执行uptime命令后可以看到一些统计数：如相对于最后1分钟、5分钟、15分钟的“平均负载”。值0意味着没有活跃的进程（除了*swapper*进程0）在运行，而值1意味着一个单独的进程100%占有CPU，值大于1说明几个运行着的进程共享CPU。

更新了系统时钟以后，update_times()再次打开中断，并执行下列动作：

- 把lost_ticks的值保存在ticks后，清lost_ticks
- 把lost_ticks_system的值保存在system后，清lost_ticks_system
- 调用calc_load(ticks)
- 调用update_process_times(ticks, system)

calc_load()函数计算处于TASK_RUNNING或TASK_UNINTERRUPTIBLE状态的进程的个数，并利用这个数更新CPU使用统计数。

注5：事实上，这个函数要复杂得多，因为它可能稍微调整1000这个值。如果已经发出了adjtimex()系统调用，那么这种调整就可能是必要的。

update_process_times()函数更新类型为kernel_stat的kstat变量中存放的一些内核统计数；然后，它又调用update_one_process()更新存放统计数的一些域，通过times()系统调用可以把这些统计数输出到用户程序。特别强调的是，要区别CPU在用户态和内核态所花费时间之不同。update_one_process()函数执行下列操作：

- 更新current进程描述符的per_cpu_utime域，这个域中存放着进程在用户态运行期间所用的节拍数。
- 更新current进程描述符的per_cpu_stime域，这个域中存放着进程在内核态运行期间所用的节拍数。
- 调用do_ process_times()，检查总的CPU时间限制是否已到达，如果是，向当前进程发送SIGXCPU和SIGKILL信号。在第三章的“进程的使用限制”一节中，描述了每个进程描述符的rlim[RLIMIT_CPU].rlim_cur域所控制的限制。
- 调用do_it_virt()和do_it_ prof()函数，这两个函数将在后面的“setitimer()和alarm()系统调用”一节中介绍。

在进程描述符中还提供了两个附加的域times.tms_cutime和times.tms_cstime，这两个域计算进程的子进程在用户态和内核态分别花费的CPU节拍数。出于效率的考虑，这些域不在do_process_times()中更新，而是在父进程查询其中一个子进程状态的时候才更新（参见第三章“撤消进程”一节）。

CPU的分时（time-sharing）

定时中断对于可运行（即处于TASK_RUNNING状态）的进程之间共享CPU的时间是必不可少的。我们在第十章将看到，通常给每个进程分配一个时间片，如果时间片到时进程还没有终止，那么，schedule()函数选择一个新的进程投入运行。

进程描述符的counter域表示进程在CPU上运行所剩余的节拍数。时间片总是一个节拍的倍数，即大约10ms的倍数。这个记数器的值在每个节拍的时候都由update_process_times()更新：

```
if (current->pid) {
  current->counter -= ticks;
  if (current->counter < 0) {
```

```
            current->counter = 0;
            current->need_resched = 1;
        }
    }
```

在第三章的“标识一个进程”一节中我们曾提到，PID为0（*swapper*）的进程不必与其他进程共享CPU时间，因为当不存在其他的TASK_RUNNING进程时，它才在CPU上运行。

因为counter由一个下半部分以延缓的方式更新，所以递减的过程可能大于一个节拍，因此，用局部变量ticks表示从下半部分被激活以后所产生的节拍个数。当counter变得小于0时，把进程描述符的need_resched域设置为1。在这种情况下，恢复用户态的程序执行之前将调用schedule()函数，并让其他处于TASK_RUNNING状态的进程有机会恢复在CPU上运行。

定时器的作用

定时器是一种软件工具，它允许在将来的某个时刻，当给定的时间间隔用完时调用函数。超时表示一个时刻，到这一时刻，与定时器相关的时间间隔已经用完。

内核和进程广泛地使用定时器。大多数设备驱动程序利用定时器检测反常情况，例如，当有一会儿不存取软驱以后，软盘驱动程序就关闭该设备的发动机，并行打印机设备利用定时器检测错误的打印机情况。

编程人员也经常利用定时器在将来某一时刻执行特定的函数（参见后面的“setitimer()和alarm()系统调用”一节）。

相对来说，实现一个定时器并不难。每个定时器包含一个域，这个域表示定时器将需要多长时间才到期。这个域值的计算是把jiffies的当前值加上正确的节拍数得到的。这个域的值不再改变。每当内核检查一个定时器时，就把到期域的值和当前这一刻jiffies的值相比较，当jiffies大于或等于这个域所在的值时，定时器到期。这种比较是通过time_after、time_before、time_after_eq及time_before_eq宏进行的，这些宏处理jiffies可能会出现的溢出。

Linux考虑了三种类型的定时器，即静态定时器（static timer）、动态定时器（dynamic timer）及间隔定时器（interval timer）。前两种类型由内核使用，而间隔定时器可以由进程在用户态创建。

有关Linux定时器的警告：因为对定时器函数的检查总是由下半部分进行，而下半部分被激活以后很长时间才能被执行，因此，内核不能确保定时器函数正好在定时到期时开始执行，而只能保证在适当的时间执行它们，或者假定延迟到几百毫秒之后执行它们。因此，对于必须严格遵守定时时间的那些实时应用，定时器并不适合。

静态定时器

Linux的最初版本只允许32个不同的定时器（注6）。这些静态定时器(依靠静态分配的内核数据结构)现在还在使用。因为它们是最先引入的，因此，Liunx代码涉及到它们时把它们作为旧定时器。

静态定时器存放在timer_table数组中，包含32个数组项。每一项由下列timer_struct结构组成：

```
struct timer_struct {
    unsigned long expires;
    void (*fn)(void);
};
```

expires域指定定时器到期时间。这个时间被表示成自系统启动以来的时钟节拍数。expires值小于或等于jiffies值的所有定时器就认为是到期了或停止了。fn域包含定时器到期时被执行函数的地址。

尽管timer_table包含了32项，但Linux只用了表5-1所列出的那些项。

表5-1 静态定时器

静态定时器	定时对象
BACKGR_TIMER	I/O操作请求的基础
BEEP_TIMER	提示铃
BLANK_TIMER	屏幕保护
COMTROL_TIMER	Comtrol串口卡
COPRO_TIMER	i80387协处理器

注6： 之所以定为这个值是由于要使相应的激活标志能被存储在一个单一的变量中。

表5-1 静态定时器（续）

静态定时器	定时对象
DIGI_TIMER	Digiboard卡
FLOPPY_TIMER	软盘
GDTH_TIMER	GDTH SCSI驱动器
GSCD_TIMER	Goldstar CD-ROM
HD_TIMER	硬盘（老式IDE驱动器）
MCD_TIMER	Mitsumi CD-ROM
QIC02_TAPE_TIMER	QIC-02磁带机
RS_TIMER	RS-232串口卡
SWAP_TIMER	*kswapd*内核线程激活

用timer_active来识别活动的静态定时器：32位变量的每一位是一个标志，指定相应的定时器是否被激活。

为了激活一个静态定时器，内核必须简单地：

- 登记定时器的fn域要执行的函数。
- 计算到期时间(这通常是通过把某一指定的值加到jiffies的值上而完成的)，并把这一时间保存在定时器的expires域。
- 在timer_active中设置合适的标志。

对到期静态定时器的检查工作是由run_old_timers()函数完成的，这个函数由TIMER_BH下半部分调用：

```
void run_old_timers(void)
{
    struct timer_struct *tp;
    unsigned long mask;
    for (mask = 1, tp = timer_table; mask;
            tp++, mask += mask) {
        if (mask > timer_active)
            break;
        if (!(mask & timer_active))
            continue;
```

```
            if (tp->expires > jiffies)
                continue;
            timer_active &= ~mask;
            tp->fn();
            sti();
        }
    }
```

一旦识别出一个到期的定时器，就执行fn域指向的函数，但在执行前必须清timer_active中相应的标志，这就保证了run_old_timers()将来每次执行时不再调用这个定时器。

动态定时器

动态定时器被动态地创建和撤消，对当前活动动态定时器的个数也没有限制。

动态定时器存放在下列timer_list结构中：

```
struct timer_list {
    struct timer_list *next;
    struct timer_list *prev;
    unsigned long expires;
    unsigned long data;
    void (*function)(unsigned long);
};
```

function域包含定时器到期时要执行函数的地址。data域指定传递给这个定时器函数的参数。data域使得定义一个通用函数处理几个设备驱动程序的定时问题成为可能，可以在data域存放设备ID，或其他有意义的数据，这些数据可被用来区分不同设备的函数使用。

expires域的含义与静态定时器相应域的含义相同。

next和prev域实现双向循环链表的连接。事实上，每个活动的动态定时器根据expires的值被精确地插入到512个双向循环链表的其中一个中。在本章稍后将描述使用这个链表的算法。

为了创建并激活一个动态定时器，内核必须：

1. 创建一个新的struct timer_list对象，比如说设为t。这可以通过几种方式来做：
   - 在代码中定义一个静态全局变量
   - 在一个函数内定义一个局部变量：在这种情况下，这个对象存在内核堆栈中
   - 在一个动态分配的描述符中包含这个对象
2. 调用init_timer(&t)函数初始化这个对象。只需简单地把next和prev域设置为空。
3. 如果这个动态定时器还没有被插入到链表中，给expires域赋一个合适的值。否则，如果这个动态定时器已经被插入到链表中，调用mod_timer()函数更新expires域，它也会把这个对象移到合适的链表中（不久就会讨论）。
4. 向function域里填入定时器到期时需要激活的函数的地址。如果有必要，在data域里填上传递给这个函数的参数值。
5. 如果这个动态定时器还没有被插入到链表中，调用add_timer(&t)函数把t元素插入到合适的链表中。

一旦定时器到期，内核自动地把元素t从它的链表中删除。不过，有时进程应该用del_timer()函数明确地从定时器链表中删除一个定时器。事实上，在定时到期结束前，睡眠的进程可能被唤醒，在这种情况下，进程可以选择撤消这个定时器。对已经从链表删除的定时器调用del_timer()没什么害处，因此，可以认为从定时器函数中调用del_timer()是一种比较好的方式。

前面我们已经看到，run_old_timers()函数是如何通过在32个timer_table元素上执行一个单循环，来识别出那些活动的并且到期的静态定时器的。这种方法不再适用于动态定时器，因为在每个时钟节拍去扫描一个动态定时器的长链表太费时。另一方面，维护一个排序的链表效率也不高，因为插入和删除操作也非常费时。

解决的办法以一种聪明的数据结构为基础，这种数据结构把expires值划分成不同的大小，并允许动态定时器从大expires值的链表以一种十分高效的方式过滤到小expires值的链表。

其主要数据结构是一个叫tvecs的数组，数组的元素指向五组分别由tv1、tv2、tv3、tv4及tv5结构标识的链表（见图5-1）。

tv1结构的类型为struct timer_vec_root，描述如下（译注1）：

```
struct timer_vec_root {
        int index;
        struct timer_list *vec[TVR_SIZE];
    };
```

其中，TVR_SIZE为256。它包含一个index字段和一个数组，这个数组由256个指向timer_list（即指向动态定时器列表）的指针组成。这个结构包含了在未来的255个节拍内将要到期的所有动态定时器。

index域指定当前正在扫描的链表，其初值为0，并在每个节拍时增1（以256为模）。index所指向的链表包含了当前节拍期间已经到期的所有动态定时器；下一个链表包含了下一个节拍将要到期的所有动态定时器；第（index+k）个链表包含了正好在第k个节拍到期的所有动态定时器。当index返回到0时，这意味着在tv1中的所有定时器都被扫描了一遍，在这种情况下，由tv2.vec[tv2.index]指向的链表来补充tv1。

类型为struct timer_vec的tv2、tv3及tv4结构分别包含了在紧接着到来的2^{14}-1、2^{20}-1及2^{26}-1个节拍内将要到期的所有动态定时器。

tv5的结构与前面的几个结构相同，除了vec数组的最后一项包含的动态定时器expires域的值可以任意大，它再也不需要从另一个数组进行补充了。

timer_vec结构非常类似于timer_vec_root：它有一个index域和一个由64个指向动态定时器列表的指针构成的vec数组。index域指定当前扫描的链表，每256^{i-1}个节拍增1（以64为模），这里i的取值范围为2到5，是tv$i$的组号。如在tv1的情况下，index为0，tv$j$.vec[tv$j$.index]指向的链表用tv$i$补充（i的取值范围为2到4，j等于i+1）。

tv2的单独一项就足以补充tv1的整个数组；tv3的单独一项足以补充tv2整个的数组，等等。

译注1：为阅读方便，结构原型为译者所加。

图 5-1 显示了这些数据结构是如何连接在一起的。

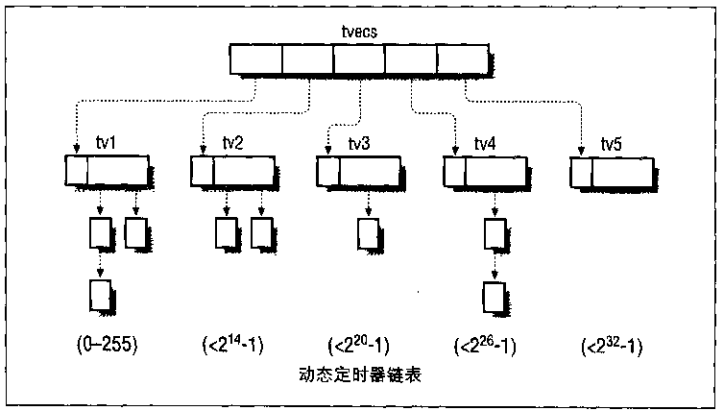


图 5-1 与动态定时器相关的链表组

与 TIMER_BH 下半部分相关的 timer_bh() 函数调用 run_timer_list() 这个辅助函数来检查到期的动态定时器。timer_bh() 函数依赖一个类似于 jiffies 的变量叫做 timer_jiffies。需要这个新的变量是因为，激活的 TIMER_BH 下半部分在有机会执行前还可能会有一些定时中断产生。典型的情况是，在一个短的时间间隔内发出几个不同类型的中断。

timer_jiffies 的值表示动态定时器链表仍需检查的到期时间。如果这个值与 jiffies 的值一样，说明下半部分函数没有积压；如果这个值小于 jiffies，说明指向前几个节拍的下半部分必须被处理。这个变量在系统启动时赋初值 0，然后每个节拍由 run_timer_list() 增加，它的值永远不会超过 jiffies。

run_timer_list() 函数的 C 代码如下（假定在单处理器系统上）：

```
cli();
while ((long)(jiffies - timer_jiffies) >= 0) {
    struct timer_list *timer;
    if (!tv1.index) {
        int n = 1;
```

```
            do {
                cascade_timers(tvecs[n]);
            } while (tvecs[n]->index == 1 && ++n < 5);
        }
        while ((timer = tv1.vec[tv1.index])) {
            detach_timer(timer);
            timer->next = timer->prev = NULL;
            sti();
            timer->function(timer->data);
            cli();
        }
        ++timer_jiffies;
        tv1.index = (tv1.index + 1) & 0xff;
    }
    sti();
```

当timer_jiffies变得大于jiffies时，最外层的while循环结束。因为jiffies和timer_jiffies的值经常是一样的，所以最外层的while循环常常只执行一次。一般情况下，最外层循环连续执行jiffies - timer_jiffies + 1次。此外，如果在run_timer_list()正执行时发生一个定时中断，也得考虑在这个节拍所出现的到期动态定时器，因为IRQ0中断处理程序是异步增加jiffies的（参见前面“PIT中断服务例程”一节）。

在最外层while循环的一次执行中，在tv1.vec[tv1.index]链表中的动态定时器函数被执行。在执行一个动态定时器函数前，这个循环调用detach_timer()函数从链表中删除动态定时器。一旦这个链表为空就把tv1.index的值（以256为模）和timer_jiffies的值增加1。

当tv1.index变为0时，tv1链表中的所有动态定时器都被检查过。在这种情况下，再填充tv1结构是必要的。这是由cascade_timers()函数完成的，这个函数把包含在tv2.vec[tv2.index]中的动态定时器传递到tv1.vec，因为它们将在紧接着到来的256个节拍必然地到期。如果tv2.index等于0，有必要用tv3.vec[tv3.index]的元素填充链表的tv2数组，等等。

请注意，就在进入最外层循环前，run_timer_list()要关中断。调用每个动态定时器函数前正好打开中断；函数执行结束又正好关闭中断。这就保证了动态定时器的数据结构不被交错执行的内核控制路径所破坏。

综上所述可知，这种相当复杂的算法确保了极好的性能。让我们来看看为什么，为了简单起见，假定TIMER_BH下半部分正好在相应的定时中断发生后执行。那么在256次定时中断中所发生的255次中，也就是在99.6%的情况中，如果必要，run_imter_list()仅仅运行到期定时器的函数。为了周期性地补充tv1.vec，在64次补充当中，63次足以把tv2.vec[tv2.index]指向的链表分成tv1.vec指向的256个链表。依次地，tv2.vec数组必须在0.02%的情况下得到补充，即每163秒一次。类似地，每2小时54分补充一次tv3，每7天18小时补充一次tv4，而tv5不需被补充。

动态定时器的应用

在一些情况下，例如，当内核不能提供一个给定的服务时，就可以把当前进程挂起一个固定的时间。这通常是通过执行一个进程延时完成的。

让我们假定内核决定把当前进程挂起两秒，可以通过执行下列代码来做到这一点：

```
timeout = 2 * HZ;
current->state = TASK_INTERRUPTIBLE;
timeout = schedule_timeout(timeout);
```

内核用动态定时器实现进程的延时。它们出现在schedule_timeout()函数中，该函数执行下列语句：

```
struct timer_list timer;
expire = timeout + jiffies;
init_timer(&timer);
timer.expires = expire;
timer.data = (unsigned long) current;
timer.function = process_timeout;
add_timer(&timer);
schedule();     /* 进程被挂起直到定时器到期 */
del_timer(&timer);
timeout = expire - jiffies;
return (timeout < 0 ? 0 : timeout);
```

当schedule()被调用时，就选择另一个进程执行；当前一个进程恢复执行时，该函数就删除这个动态定时器。在最后一句中，函数返回的值有两种可能，0表示延时到期、timeout表示如果进程因某些其他原因被唤醒，到延时到期时还剩余的节拍数。

当延时到期时，内核执行下列函数：

```
void process_timeout(unsigned long data)
{
    struct task_struct * p = (struct task_struct *) data;
    wake_up_process(p);
}
```

run_timer_list()函数调用process_timeout()，把存在定时器对象data域的进程描述符指针作为参数传递过去。结果，挂起的进程被唤醒。

与定时测量相关的系统调用

有几个系统调用允许用户态下的进程读取及修改时间和日期，以及创建定时器。让我们对它们进行一些简单的回顾，并讨论一下内核是如何处理它们的。

time()，ftime()及gettimeofday()系统调用

用户模式下的进程通过以下几个系统调用获得当前时间和日期：

time()

返回从1970年1月1日午夜开始所走过的秒数。

ftime()

返回一个类型为timeb的数据结构，该结构包含从1970年1月1日午夜开始所走过的秒数；在最后1秒内所走过的毫秒数；时区以及夏令时当前的状态。

gettimeofday()

返回的值存在两个数据结构timeval和timezone中，其包含的信息与ftime()中的相同。

前两个系统调用都被gettimeofday()所代替，但是，为了保持向后兼容，Linux中还包含它们。我们在这里不进一步讨论它们。

gettimeofday()系统调用由sys_gettimeofday()函数实现。为了计算当前一天的时间和日期，这个函数又调用do_gettimeofday()，它执行下列动作：

- 把xtime的内容拷贝到由系统调用参数tv指定的用户空间的缓冲区中：

```
*tv = xtime;
```

- 调用由do_gettimeoffset变量指向的函数来更新微秒数：

```
tv->tv_usec += do_gettimeoffset();
```

如果CPU有时间标记计数器，就执行do_fast_gettimeoffset()函数。这个函数用汇编语言指令rdtsc读TSC寄存器的值，然后减去保存在last_tsc_low中的值，就得到了从最后一个定时中断被处理以来CPU所花费的周期数。函数把这个数转换成微秒，并把它加到定时中断处理程序激活之前的延迟中，该延迟保存在前面“PIT中断服务例程”一节中所提到的delay_at_last_interrupt变量中。

如果CPU没有TSC寄存器，do_gettimeoffset指向do_slow_gettimeoffset()函数。这个函数读取8254芯片设备内部振荡器中的状态，然后计算自从最后一个定时中断以来所花费的时间。用这个值和jiffies的内容就可以推导出在最后一秒中已经过去的微秒数。

- 进一步地增加微秒数来考虑那些下半部分还没有被执行的所有定时中断：

```
if (lost_ticks)
    tv->tv_usec += lost_ticks * (1000000/HZ);
```

- 最后，检查微秒域的溢出，如果需要，调整微秒域和秒域：

```
while (tv->tv_usec >= 1000000) {
    tv->tv_usec -= 1000000;
    tv->tv_sec++;
}
```

拥有root权限的用户态下的进程可以用老式的stime()或settimeofday()系统调用来修改当前日期和时间。sys_settimeofday()函数调用do_settimeofday()，do_settimeofday()执行do_gettimeofday()操作的反操作。

请注意当这两个系统调用修改xtime的值时都没有修改RTC寄存器，因此当系统关机时新的时间会丢失，除非用户执行/sbin/clock这个程序来改变RTC的值。

adjtimex()系统调用

尽管时钟的走动确保了所有的系统最终都会从正确的时间离开，但是，突然改变时间既是一种管理的失误也是一种危险的行为。例如，设想程序员试图编译一个大规模的程序，并依靠文件时间标记来确保旧的文件对象被重新编译。系统时间大的改动可能搞乱make程序，并导致不正确的编译。当在计算机网络上执行一个分布式文件系统时，保持时钟的调整也是很重要的。在这种情况下，采用这种做法是明智的:调整互连PC的时钟以使所存取文件的inode中的时间标记值保持一致。因此，系统通常被配置成能运行一种建立在一个基准上的时间同步协议，在每一个节拍逐渐地调整时间,例如网络定时协议(NTP)。在Linux中，这个工作依赖于adjtimex()系统调用。

尽管这个系统调用不应该用在打算移植的程序中，但是它已经出现在几个Unix变种中。它接收指向timex结构的指针作为参数，从timex域中的值更新内核参数，并返回具有当前内核值的同一结构。update_wall_time_one_tick()使用这样的内核值对每一个节拍中加到xtime.tv_usec的微秒数进行细微地调整。

setitimer() 和 alarm()系统调用

Linux允许用户态的进程激活被称为间隔定时器的特殊定时器（注7)。这种定时器引起的Unix信号（参见第九章）被周期性地发送到进程。也可能激活一个间隔定时器以便在指定的延时后它仅发送一个信号，因此，间隔定时器由以下两个方面来刻画:

- 发送信号所必须的频率，或者如果只需要产生一个信号，则频率为空
- 在下一个信号被产生以前所剩余的时间。

在本章前面关于精确性的警告同样适用于这些定时器。在要求的时间已过去之后，可以确保这些定时器执行，但是却不可能预知恰好在什么时候会执行它们。

通过POSIX setitimer()系统调用可以激活间隔定时器。第一个参数指定应当采取下面的哪一个策略:

注7: 这些软件的构造与以前所描述的可编程间隔定时器没有什么共同之处。

ITIMER_REAL

真正过去的时间；进程接受SIGALRM信号

ITIMER_VIRTUAL

进程在用户态下花费的时间；进程接受SIGVTALRM信号

ITIMER_PROF

进程既在用户态下又在内核态下所花费的时间；进程接受SIGPROF信号

为了能分别实现前述每种策略的间隔定时器，进程描述符要包含三对域：

- it_real_incr和it_real_value
- it_virt_incr和it_virt_value
- it_prof_incr和it_prof_valu

每对中的第一个域存放着两个信号之间以节拍为单位的间隔；另一个域存放着定时器的当前值。

ITIMER_REAL间隔定时器是利用动态定时器实现的，因为即使进程不在CPU上运行时，内核也必须向进程发送信号。因此，每个进程描述符包含一个叫real_timer的动态定时器对象。setitimer()系统调用初始化real_timer域，然后调用add_timer()把动态定时器插入到合适的链表中。当定时器到期时，内核执行it_real_fn()定时函数。依次类推，it_real_fn()函数向进程发送一个SIGALRM信号。如果it_real_incr不为空，那么它会再次设置expires域，重新激活定时器。

ITIMER_VIRTUAL和ITIMER_PROF间隔定时器不需要动态定时器，因为只有当进程运行时，它们才能被更新：do_it_virt()和do_it_prof()由update_one_process()调用，当执行TIMER_BH下半部分时update_one_process()才运行。因此，每个节拍中，这两个间隔定时器通常都被更新一次，并且如果它们到期，就给当前进程发送一个合适的信号。

alarm()系统调用会在一个指定的时间间隔用完时向调用的进程发送一个SIGALRM信号。当以ITIMER_REAL为参数调用时，它非常类似于setitimer()，因为它利用了包含在进程描述符中的real_timer动态定时器。因此，不能同时使用以ITIMER_REAL为参数的alarm()和setitimer()。

对 Linux 2.4 的展望

Linux 2.4 对 2.2 版的时间处理函数没有引入什么重大的变化。

第六章
内存管理

在第二章中我们已看到，Linux如何有效地利用Intel分段和分页单元把逻辑地址转换为物理地址。同时我们还提到把RAM的某些部分永久地分配给内核用来存放内核代码以及静态内核数据结构。

RAM的其余部分称为动态内存（dynamic memory），这不仅是进程所需的宝贵资源，也是内核本身所需的宝贵资源。实际上，整个系统的性能取决于如何有效地管理动态内存。现在所有多任务操作系统都在尽力优化对动态内存的使用，也就是说，尽可能做到当需要时分配，不需要时释放。

本章主要通过三部分内容介绍内核如何给自己分配动态内存。“页框管理”和“内存区管理”两节分别介绍两种对连续物理内存区处理的不同技术。“非连续内存区管理”一节介绍了第三种技术：处理不连续的内存区。

页框管理

在第二章的“硬件的分页单元”一节中我们曾介绍过，Intel的奔腾处理器可以采用两种不同的页框（page frame）大小：4KB和4MB。Linux采用4KB页框大小作为标准的内存分配单元。基于以下两个原因这会使事情变得简单：

- 分页单元可以自动检测正在被访问的页是否包含在某个页框中。此外，通过使用指向页框的页表项中所包含的标志，每个页框是受硬件保护的。选用4KB作为分配单元，内核就可以直接确定发生缺页异常的页所在的内存分配单元。
- 4KB是大部分磁盘块大小的倍数，因此，在主存和磁盘之间传输数据可以更高效。管理4KB大小的数据比管理4MB的数据要更容易。

内核必须记录每个页框当前的状态。例如，内核必须能区分哪些页框包含的是属于进程的页，而哪些页框包含的是内核代码或内核数据。同理，内核还必须能够确定动态内存中页框是否空闲。这种状态信息被保存在一个描述符数组中，每个页框对应数组中的一个元素。这种类型为struct page的描述符具有如下形式：

```
typedef struct page {
    struct page *next;
    struct page *prev;
    struct inode *inode;
    unsigned long offset;
    struct page *next_hash;
    atomic_t count;
    unsigned long flags;
    struct wait_queue *wait;
    struct page **pprev_hash;
    struct buffer_head * buffers;
} mem_map_t;
```

我们只介绍其中的部分域（其余部分将在其他章节中介绍）：

count

如果相应页框为空闲，则该域设为0；如果页框被分配给一个或多个进程或用于某些内核数据结构，则该域设为一个大于0的值。

prev，next

用于把描述符插入到双向循环链表（doubly linked circular list）中。这两个域的含义取决于页框的当前使用情况。

flags

由32个标志组成的数组来描述页框状态（参见表6-1）。对于每个PG_*xyz*标志，都定义了相应的宏Page*Xyz*来读或设置它的值。

在表6-1中的某些标志我们将在后面的章节讲解。因为基于ISA总线的DMA(Direct Memory Access)处理器具有局限性(这样的DMA处理器只能访问RAM的前16MB空间),所以存在PG_DMA标志,也正因为这样,页框被分为两组,这取决于它们能否被DMA访问。[在第十三章的"直接内存访问(DMA)"一节中详细介绍DMA]。在本章中,术语DMA表示基于ISA总线的DMA。

表6-1 描述页框状态的标志

标志名	意义
PG_decr_after	参见第十六章中的"read_swap_cache_async()函数"一节
PG_dirty	没有使用
PG_error	在调页时发生I/O出错
PG_free_after	参见第十五章中的"从正规文件读取数据"一节
PG_DMA	由ISA DMA使用(参见正文)
PG_locked	页不能被换出
PG_referenced	通过页高速缓存的散列表来访问页框(参见第十四章的"页高速缓存"一节)
PG_reserved	页框留给内核代码使用或不能使用
PG_skip	用于SPARC/SPARC64结构以"跳过"一些地址空间
PG_Slab	包含在slab中:在本章后面的"内存区管理"一节中将有介绍
PG_swap_cache	包含在交换高速缓存中;参见第十六章的"交换高速缓存"一节
PG_swap_unlock_after	参见第十六章中的"read_swap_cache_async()函数"一节
PG_uptodate	在完成读操作后置位,除非发生磁盘I/O出错

系统中所有的页框描述符都存放在一个叫mem_map的数组中。由于每个描述符最多要占用64字节,因此RAM的每1MB的空间,mem_map占4个页框。宏MAP_NR

的参数是页框地址，该宏根据这个参数计算出相应页框的页框号，由此也计算出相应描述符在mem_map中的索引。

```
#define MAP_NR(addr)    (__pa(addr) >> PAGE_SHIFT)
```

MAP_NR宏利用__pa宏把逻辑地址转换为物理地址。

动态内存以及引用它的值如图6-1所示。页框描述符的初始化由free_area_init()函数完成，该函数有两个参数：start_mem表示动态内存的第一个线性地址，紧挨着内核所占内存；end_mem为动态内存的最后一个线性地址加1（参见第二章的“保留的页框”和“内核页表”两节），函数free_area_init()还要考虑变量i386_endbase，因为这个变量存放着保留页框的起始地址。free_area_init()还负责给mem_map分配合适大小的内存区，并通过把其中的所有域置0来初始化该内存区，不过，flags域除外（设置为PG_DMA和PG_reserved标志）：

```
mem_map = (mem_map_t *) start_mem;
p = mem_map + MAP_NR(end_mem);
start_mem = ((unsigned long) p + sizeof(long) - 1) &
                ~(sizeof(long)-1);
memset(mem_map, 0, start_mem - (unsigned long) mem_map);
do {
   --p;
   p->count = 0;
   p->flags = (1 << PG_DMA) | (1 << PG_reserved);
} while (p > mem_map);
```

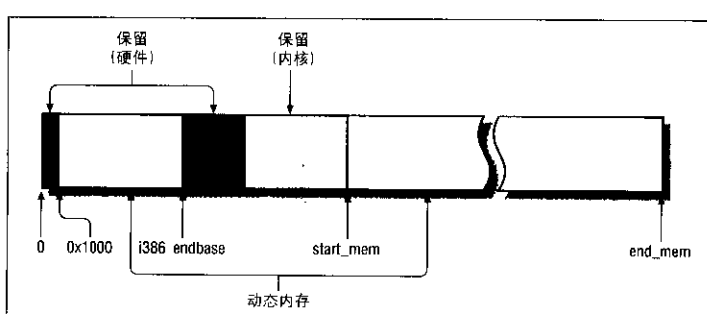


图 6-1 内存布局

随后，mem_init()函数把有关页框的PG_reserved标志清除，以使他们用作动态内存（参见第二章的“保留的页框”一节），并把物理地址大于或等于0x1000000的所有页框的PG_DMA标志清除。这些操作由下面的代码段完成：

```
start_low_mem = PAGE_SIZE + PAGE_OFFSET;
num_physpages = MAP_NR(end_mem);
while (start_low_mem < i386_endbase) {
    clear_bit(PG_reserved,
              &mem_map[MAP_NR(start_low_mem)].flags);
    start_low_mem += PAGE_SIZE;
}
while (start_mem < end_mem) {
    clear_bit(PG_reserved,
              &mem_map[MAP_NR(start_mem)].flags);
    start_mem += PAGE_SIZE;
}
for (tmp = PAGE_OFFSET ; tmp < end_mem ; tmp += PAGE_SIZE) {
    if (tmp >= PAGE_OFFSET+0x1000000)
        clear_bit(PG_DMA, &mem_map[MAP_NR(tmp)].flags);
    if (PageReserved(mem_map+MAP_NR(tmp))) {
        if (tmp >= (unsigned long) &_text
                && tmp < (unsigned long) &_edata)
            if (tmp < (unsigned long) &_etext)
                codepages++;
            else
                datapages++;
        else if (tmp >= (unsigned long) &__init_begin
                 && tmp < (unsigned long) &__init_end)
            initpages++;
        else if (tmp >= (unsigned long) &__bss_start
                 && tmp < (unsigned long) start_mem)
            datapages++;
        else
            reservedpages++;
        continue;
    }
    mem_map[MAP_NR(tmp)].count = 1;
    free_page(tmp);
}
```

首先，mem_init()函数确定num_physpages的值，也就是系统现有的页框总数。然后计算PG_reserved类型的页框数。编译内核时产生的一些符号（在第二章的

“保留的页框”一节中有它们的描述）能使这个函数计算出保留给硬件、内核代码、内核数据的页框数，还能计算出内核初始化期间使用的随后又被释放的页框数。

最后，mem_init()函数把与动态内存相关连的每个页框描述符count域的值置为1，并调用函数free_page()（参见本章后面的“伙伴系统算法”一节）。因为free_page()函数增加变量nr_free_pages的值，因此在循环结束时，这个变量将包含动态内存中页框的总数。

请求和释放页框

以上我们介绍了内核如何分配和初始化页框处理所涉及的数据结构，接下来我们看页框是如何被分配和释放的。通过下面四个稍有差别的函数和宏可以请求页框：

__get_free_pages(gfp_mask, order)

用这个函数请求2^{order}个连续的页框。

__get_dma_pages(gfp_mask, order)

用这个宏获得适合DMA的页框，它扩展为：

```
get_free_pages(gfp_mask | GFP_DMA, order)
```

__get_free_page(gfp_mask)

用这个宏获得一个单独的页框，它扩展为：

```
__get_free_pages(gfp_mask, 0)
```

get_free_page(gfp_mask)：

该函数调用：

```
__get_free_page(gfp_mask)
```

然后在所得到的页框中填0。

参数gfp_mask表示如何查找空闲的页框。它由以下几个标志组成：

__GFP_WAIT

如果在满足请求之前允许内核丢弃页框的内容以释放内存就设置该标志。直到2.2.14版本，进程可以阻塞直至有空闲页框可用。

__GFP_IO

如果允许内核把页的内容写到磁盘以释放相应页框就设置该标志。（因为在内

核态，交换可以阻塞进程。因此当处理中断或修改内核重要的数据结构时必须把该位清除。）

__GFP_DMA

如果请求的页框必须适用于DMA时设置该位。（引起这个标志的硬件局限性在前面的“页框管理”一节中已经介绍。）

__GFP_HIGH，__GFP_MED，__GFP_LOW

指定请求的优先级。__GFP_LOW通常与在用户态的进程发出的对动态内存的请求有关，而其他两个与内核态的请求有关。

实际上，Linux使用预定义标志值的组合，如表6-2所示，在源代码中会碰到这些组名。

表6-2 用于请求页框的标志值组

组名	__GFP_WAIT	__GFP_IO	优先级
GFP_ATOMIC	0	0	__GFP_HIGH
GFP_BUFFER	1	0	__GFP_LOW
GFP_KERNEL	1	1	__GFP_MED
GFP_NFS	1	1	__GFP_HIGH
GFP_USER	1	1	__GFP_LOW

通过下面三个函数和宏可以释放页框：

free_pages(addr, order)

该函数先检查物理地址为addr的页框的页描述符，如果该页框未被保留（PG_reserved标志为0），就把描述符的count域减1。如果count值为0，可以假定从地址addr开始的2^{order}个连续的页框不再使用。然后调用free_pages_ok()把第一个空闲页的页框描述符插入到空闲页框的适当链表中（下一节描述）。

__free_page(p)

与上一个函数很相似，不同之处就是它释放的是参数p所指向的页框（p为该页框的描述符）。

free_page(addr)

该宏释放物理地址为addr的页框。它扩展为free_pages(addr,0)。

伙伴（Buddy）系统算法

内核应该为分配一组连续的页框而建立一种稳定、高效的分配策略。为此，必须解决一个比较重要的内存管理问题，就是所谓的外碎片（external fragmentation）问题。频繁的请求和释放不同大小的一组连续页框，必然导致在已分配页框的块内分散许多小块的空闲页框。由此带来的问题是，即使有足够的空闲页框可以满足请求，但要分配一个大块的连续页框就可能无法满足。

从本质上说，避免外碎片的方法有两种：

- 利用分页单元把一组非连续的空闲页框映射到连续的线性地址区间。
- 开发一种适当的技术来记录现存的空闲连续页框块的情况，以尽量避免为满足对小块的请求而把大块的空闲块进行分割。

基于以下两个原因，内核首选第二种方法：

- 在某些情况下，由于连续的线性地址不足以满足请求，因此连续的页框确实是必要的。一个典型的例子就是请求内存给DMA处理器分配缓冲（参见第十三章）。因为DMA忽略分页单元而直接访问地址总线，因此，当在一次单独的I/O操作中传送几个磁盘扇区的数据时，所请求的缓冲就必须位于连续的页框。
- 即使连续页框的分配并不是很必要，但它在保持内核页表不变方面所起的作用也是不容忽视的。修改页表会怎样呢？从第二章我们知道，频繁的修改页表势必导致平均访问内存次数的增加，因为这会使CPU频繁地刷新转换后援缓冲器（TLB）的内容。

Linux采用著名的伙伴系统算法来解决外碎片问题。把所有的空闲页框分组为10个块链表，每个块链表分别包含大小为1，2，4，8，16，32，64，128，256和512个连续的页框。每个块的第一个页框的物理地址是该块大小的整数倍。例如：大小为16个页框的块，其起始地址是 16×2^{12} 的倍数。

我们通过一个简单的例子来说明该算法的工作原理。

假设要请求一个128个页框的块（即0.5M字节）。该算法先在128个页框的链表中检查是否有一个空闲块。如果没有这样的块，该算法会查找下一个更大的页块，也就是，在256个页框的链表中找一个空闲块。如果存在这样的块，内核就把256的页框分成两等份，一半用作满足请求，另一半插入到128个页框的链表中。如果在256个页框的块链表中也没找到空闲块，就继续找更大的块——512个页框的块。如果这样的块存在，内核把512个页框的块的128个页框用作请求，然后从剩余的384个页框中拿256个插入到256个页框的链表中，再把最后的128个插入到128个页框的链表中。如果512个页框的链表还是空的，该算法就放弃并发出错信号。

以上过程的逆过程就是页框块的释放过程，也是该算法名字的来由。内核试图把大小为b的一对空闲伙伴块合并为一个大小为$2b$的单独块。满足以下条件的两个块称为伙伴：

- 两个块具有相同的大小，记作b。
- 它们的物理地址是连续的。
- 第一块的第一个页框的物理地址是$2 \times b \times 2^{12}$的倍数。

该算法是迭代的，如果它成功合并所释放的块，它会试图合并$2b$的块来形成更大的块。

数据结构

Linux使用两种不同的伙伴系统：一种处理适合ISA DMA的页框，另一种处理其他页框。每个伙伴系统主要使用以下数据结构：

- 前面介绍过的mem_map数组。
- 类型为free_area_struct，有10个元素的一个数组，每个元素对应一组大小一致的块。变量free_area[0]指向的数组是伙伴系统为非ISA DMA所使用的页框，变量free_area[1]指向的数组是伙伴系统为ISA DMA所使用的页框。
- 有十个二进制的位图（bitmap）数组，每一种块大小对应一个数组。每个伙伴系统都有自己的位图集，用位图集来记录内存块的分配情况。

数组free_area[0]和free_area[1]的每个元素都是free_area_struct类型的一个结构，其定义如下：

```
struct free_area_struct {
    struct page *next;
    struct page *prev;
    unsigned int *map;
    unsigned long count;
};
```

注意，该结构的前两个域和页描述符中相应的域匹配，事实上，指向free_area_struct结构的指针有时用作指向页描述符的指针。

free_area[0]或free_area[1]的第k个元素是大小为2^k的块的双向循环链表，这是通过next和prev域来实现的。这种链表中每个成员是一个块的第一个页框描述符。数据结构free_area_struct的count域存放相应链表元素的个数。

map域指向一个位图，它的大小取决于现有的页框数。free_area[0]或free_area[1]的第k项位图的每一位描述大小为2^k的页框两个伙伴块的状态。如果位图的某位为0，表示一对兄弟块中或者两个都空闲或者两个都忙，如果为1，肯定有一块为忙。当兄弟块都空闲时，内核把他们当作一个大小为2^{k+1}的单独块来处理。

举例说明，让我们看一下与非DMA页框相关的128MB的RAM和相应的位图。128MB可以分成32768个单独的页，16384个大小为2个页的块，8192个大小为4个页的块，64个大小为512个页的块。因此，对应于free_area[0][0]的位图由16384位组成，32768个现有页框中的每对页框对应其中的一位；对应于free_area[0][1]的位图由8192位组成，每对含有2个页框的块对应其中一位；对应于free_area[0][9]的最后一个位图由32位组成，每对含有512个连续的页框的块对应其中的一位。

图6-2用一个简单例子说明伙伴系统算法所引入的数据结构的使用情况。数组mem_map在顶部包含9个空闲页框，上面一块由一个单独页框组成，下面二块分别由4个页框组成。双向箭头表示由next和prev域实现的双向循环链表。注意位图并不是按比例来画的。

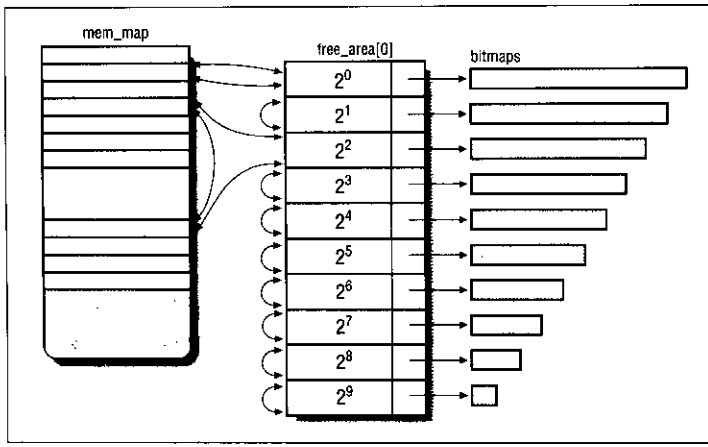


图 6-2 伙伴系统使用的数据结构

分配一个块

__get_free_pages()函数实现了分配页框的伙伴系统策略。这个函数首先检查是否有足够的空闲页，也就是说，nr_free_pages是否大于freepages.min。如果不是，它可以决定回收页框[参见第十六章中的“try_to_free_pages()函数”一节]。否则，通过执行RMQUEUE_TYPE宏中所包含的代码继续分配。

```
if (!(gfp_mask & __GFP_DMA))
    RMQUEUE_TYPE(order, 0);
RMQUEUE_TYPE(order, 1);
```

第一个参数order表示请求空闲页块大小的对数（0表示1页的块，1表示2页的块如此等等）。第二个参数是在free_area中的索引（0表示非DMA块，1表示DMA块）。代码还检查gfp_mask，看是否允许非DMA块，若是，便可从那个链表中获取块（索引值为0），因为最好把DMA块保存给真正需要它们的请求。如果页框分配成功，宏RMQUEUE_TYPE的代码就执行返回语句，由此结束__get_free_pages()函数。如果检测为DMA块，那么RMQUEUE_TYPE的第二个参数取1再执行，也就是，用适合DMA的页框来满足内存的分配请求。

宏RMQUEUE_TYPE产生的代码等价于如下程序片段。首先，声明并初始化几个局部变量：

```
struct free_area_struct * area = &free_area[type][order];
unsigned long new_order = order;
struct page *prev;
struct page *ret;
unsigned long map_nr;
struct page * next;
```

变量type表示宏的第二个参数：当宏处理非DMA页框时，该变量为0，否则为1。

然后执行一个循环，为找到一个可用的块而查遍每个链表。首先在链表中查找满足参数order的块，如果必要的话再继续查找更大的块。执行的循环部分等效于如下结构：

```
do {
    prev = (struct page *)area;
    ret = prev->next;
    if ((struct page *) area != ret)
        goto block_found;
    new_order++;
    area++;
} while (new_order < 10);
```

在循环结束时，如果仍没找到合适的空闲块，那么__get_free_pages()返回NULL，若找到合适块，那么该块的第一个页框的描述符从链表中移走，相应的位图被更新，nr_free_pages的值减小。

```
block_found:
    prev->next = ret->next;
    prev->next->prev = prev;
    map_nr = ret-mem_map;
    change_bit(map_nr>>(1+new_order), area->map);
    nr_free_pages -= 1 << order;
    area->count--;
```

如果所找到块是从比参数order大的new_order链表中找到的，那么就执行一个循环。下面这些代码行中所包含的基本原理如下：当请求2^h个页框时，就有必要用2^k个页框的块来满足这个请求（$h < k$），程序把最后2^h个页框分配给请求，然后把剩余的$2^k - 2^h$个页框反复再分配，加到索引在h和k之间的free_area链表中。

```
    size = 1 << new_order;
    while (new_order > order) {
        area--;
        new_order--;
        size >>= 1;
        /* 插入 *ret 作为链表中的第一个元素
           并更新位图 */
        next = area->next;
        ret->prev = (struct page *) area;
        ret->next = next;
        next->prev = ret;
        area->next = ret;
        area->count++;
        change_bit(map_nr >> (1+new_order), area->map);
        /* 现在注意从 *ret 开始的空闲块
           的后半部分 */
        map_nr += size;
        ret += size;
    }
```

最后，RMQUEUE_TYPE更新所选块的页描述符的count域，然后执行一个返回指令：

```
    ret->count = 1;
    return PAGE_OFFSET + (map_nr << PAGE_SHIFT);
```

因此，__get_free_pages()函数返回所找块的地址。

释放一个块

函数free_pages_ok()按照伙伴系统的策略释放页框。它使用3个输入参数：

map_nr
: 被释放块中所包含的页框的一个页号。

order
: 块的对数值。

type
: 等于1为适合DMA的页框，等于0则不是。

该函数在开始时声明和初始化一些局部变量：

```
struct page * next, * prev;
struct free_area_struct *area = &free_area[type][order];
unsigned long index = map_nr >> (1 + order);
unsigned long mask = (~0UL) << order;
unsigned long flags;
```

mask变量包含2^{order}的二进制补码。用它来把map_nr转化为被释放块的第一个页框的页号，然后增加nr_free_pages:

```
map_nr &= mask;
nr_free_pages -= mask;
```

现在函数开始执行循环，至多循环（9 - order）次，每次都尽量把一个块与其伙伴进行合并。函数以最小的块开始，然后向上移动直到顶部。控制while循环的条件是:

```
(mask + (1 << 9))
```

这里，每次循环都把mask中设置的一位左移。循环体检查块号为map_nr的伙伴块是否空闲:

```
if (!test_and_change_bit(index, area->map))
    break;
```

如果伙伴块不是空闲的，函数退出循环；如果空闲，函数把它从相应的空闲块链表中分离出来。伙伴的块号通过交换一位从map_nr中导出:

```
area->count--;
next = mem_map[map_nr ^ -mask].next;
prev = mem_map[map_nr ^ -mask].prev;
next->prev = prev;
prev->next = next;
```

在每次循环的最后，函数更新变量mask、area、index和map_nr:

```
mask <<= 1;
area++;
index >>= 1;
map_nr &= mask;
```

函数然后继续下一次循环，尽量把空闲块与上一次循环产生的相同大小的空闲块进

行合并。循环结束后，所获得的空闲块就不能再与其他的空闲块合并。然后把它插到合适的链表中：

```
next = area->next;
mem_map[map_nr].prev = (struct page *) area;
mem_map[map_nr].next = next;
next->prev = &mem_map[map_nr];
area->next = &mem_map[map_nr];
area->count++;
```

内存区管理

这部分涉及内存区（memory area），也就是说，具有连续的物理地址和任意长度的内存单元。

伙伴系统算法采用页框作为基本内存区，这适合于对大块内存的请求，但我们如何处理对小内存区的请求呢，比如说几十或几百个字节？

显然，如果为了存放很少的字节而给它分配一个整页框，这显然是一种浪费。取而代之的正确方法就是引入一种新的数据结构来描述在同一页框中如何分配小内存区。但这样也引出了一个新的问题，即所谓的内碎片（internal fragmentation）。内碎片的产生主要是由于请求内存的大小与分配给它的大小不匹配而造成的。

Linux 2.0所采用的一种典型的解决方法就是提供按几何分布的内存区大小，换句话说，内存区大小与2的幂有关而与所存放的数据大小无关。这样，不管请求内存的大小是多少，我们都可保证内碎片小于50%。为此，Linux 2.0建立了13个按几何分布的空闲内存链表，它们的大小从32到131056字节。伙伴系统的调用既为了获得额外所需的页框以存放新的内存区，也为了释放不再包含内存区的页框。用一个动态链表来记录每个页框所包含的空闲内存区。

slab 分配器

在伙伴算法上运行内存区分配算法没有显著的效率。Linux 2.2重新考察内存区分配算法并给出一些非常明智的改进。

新的算法源自于slab分配器模式，该模式早在1994年就被开发出来用于Sun Microsystem Solaris 2.4操作系统。新算法基于下列前提：

- 所存放数据的类型可以影响如何分配内存区。例如，当给用户态进程分配一个页框时，内核调用get_free_page()函数，并用0填充这个页框。

  slab分配器概念的扩充基于这种思想，并把内存区看作对象（object），这些对象由一组数据结构和几个叫做构造（constructor）或析构（destructor）的函数（或方法）组成：前者初始化内存区，而后者回收内存区。

  为了避免重复初始化对象，slab分配器并不丢弃已分配的对象，而是释放但把它们保存在内存中。当以后又要请求新的对象时，就可以从内存获取而不用重新初始化。

  实际上，Linux对内存区的处理并不需要进行初始化或回收。出于效率的考虑，Linux不依赖需要构造或析构方法的对象，引入slab分配器的主要目的是为了减少对伙伴系统分配算法的调用次数。因此，尽管内核完全支持构造和析构方法，但指向这两个方法的指针都为NULL。

- 内核函数倾向于反复请求同一类型的内存区。例如，只要内核创建一个新进程，它就要为一些固定大小的表（如进程描述符、打开文件对象等等）分配内存区。当进程结束时，包含这些表的内存区还可以被重新使用，因为进程的创建和撤消非常频繁，Linux内核的早期版本把时间浪费在反复分配和回收那些包含同一内存区的页框上；在Linux 2.2中更换为把那些页框保存在高速缓存中并重新使用。

- 对内存区的请求可以根据它们发生的频率来分类。对于预期频繁请求一个特定的大小内存区而言，可以通过创建一组具有适当大小的专用对象来高效地处理，由此以避免内碎片的产生。另一种情况，对于很少遇到的内存区大小，可以通过基于一系列几何分布大小（如Linux 2.0所使用的2的幂次方）的分配模式来处理，即使这种方法会导致内碎片的产生。

- 在引入的对象大小不是几何分布的情况下，也就是说，数据结构的起始地址不是物理地址值的2的幂次方，反倒有另外一种小小的奖赏，即通过处理器硬件高速缓存处理，这样必然导致较好的性能。

- 硬件高速缓存的使用又产生了另一个理由，以尽可能地限制对伙伴系统分配算

法的调用：因为对伙伴函数的每次调用都“弄脏”硬件高速缓存，因此增加了对内存的平均访问次数（注1）。

slab分配器把对象分组放进高速缓存。每个高速缓存都存放着同一种类型的对象。例如，当一个文件被打开时，存放相应“打开文件”对象所需的内存区是从一个叫做*filp*（“文件指针”）的slab分配器的高速缓存中得到的。可以在运行时读取*/proc/slabinfo*文件来查看Linux所使用的slab分配器高速缓存的情况。

包含高速缓存的主内存区被划分为多个slab，每个slab由一个或多个连续的页框组成，这些页框中既包含已分配的对象，也包含空闲的对象（如图6-3）。

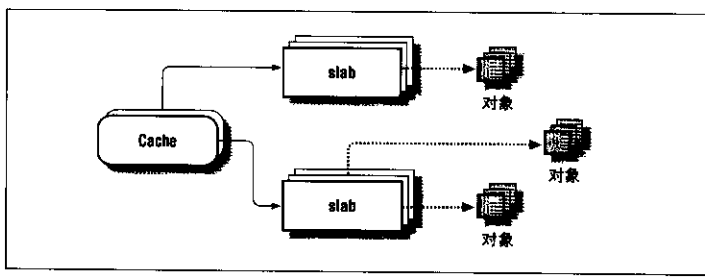


图6-3 slab分配器的组成

slab分配器从不靠它自己来释放空slab中的页框。它不知道什么时候需要空闲内存，当还有足够的空闲内存给新对象使用时，释放对象也没什么好处。因此，只有当内核正在寻找额外的空闲页框时，释放才发生（参见本章后面的“从高速缓存释放对象”一节和第十六章的“释放页框”一节）。

高速缓存描述符

每个高速缓存都是由`struct kmem_cache_s`类型（等价于`kmem_cache_t`类型）的一个表来描述的。这个表中最重要的域是：

注1：内核函数对硬件高速缓存的影响被称为函数覆盖面积(footprint)。它是指当函数操作结束时，缓存中已被函数重写的部分的百分比。显而易见，较大的覆盖面积会使恰好在内核函数后面执行的代码的执行速度下降，因为此时硬件高速缓存中填充了大量的无用信息。

c_name

指向高速缓存的名字。

c_firstp, c_lastp

分别指向高速缓存中的第一个和最后一个slab描述符。高速缓存中的slab描述符被链接成一个双向、循环、部分排序的链表：链表中的第一部分元素是没有空闲对象的slab，然后是包含已用对象但至少有一个空闲对象的slab，最后就是只包含空闲对象的slab。

c_freep

如果一组slab对象至少包含一个空闲对象，那么c_freep指向其中的第一个slab对象的s_nextp域。

c_num

集中在一个单独的slab中的对象个数。（高速缓存中的所有slab具有相同的大小。）

c_offset

高速缓存中所包含对象的大小。（如果这些对象的起始地址必须是在内存对齐的，就可以对这个大小取整。）

c_gfporder

一个单独的slab中所包含的连续页框数的对数。

c_ctor, c_dtor

分别指向与高速缓存对象相关的构造方法和析构方法。如前所述，通常把它们置为NULL。

c_nextp

指向下一个高速缓存描述符。这个域把所有的高速缓存描述符链接成一个简单的链表。

c_flags

描述高速缓存一些永久特性的一个标志数组。例如，有一个标志表示，为了在内存存放高速缓存对象描述符，应该选择两个选项中的哪一个（参见下一节）。

c_magic

从一组预先定义好的数值中选出的一个魔数（magic number），用于检查高速缓存的当前状态和它的一致性。

slab 描述符

高速缓存中的每个 slab 都有自己的类型为 struct kmem_slab_s（等价于 kmem_slab_t 类型）的描述符。

slab 描述符可以被存放在两个可能的地方，通常根据 slab 中对象的大小来选择存放的地方。如果对象的大小小于 512 字节，slab 描述符被存放在 slab 的末尾；否则，它被存放在 slab 的外面。对于大小是 slab 大小的约数的大对象，后一个选择是首选的。在某些情况下，内核可以通过不同程度地设置高速缓存描述符的 c_flags 域而违背这条原则。

slab 描述符最重要的域是：

s_inuse
: slab 中当前所被分配的对象的个数。

s_mem
: 指向 slab 内的第一个对象（或被分配或空闲）。

s_freep
: 指向 slab 中第一个空闲对象（如果有）。

s_nextp, s_prevp
: 分别指向下一个和前一个 slab 描述符。在链表中最后一个 slab 描述符的 s_nextp 域指向相应高速缓存描述符的 c_offset 域。

s_dma
: 如果在 slab 中所包含的对象是由 DMA 处理器使用就设置这个标志。

s_magic
: 类似于高速缓存描述符的 c_magic 域。它包含一个魔数，选自一个预定义值的集合并用来检查 slab 的当前状态和它的一致性。这个域的值不同于高速缓存描述符对应域 c_magic 的值。s_magic 在 slab 描述符内的偏移量等于关于 c_offset 的 c_magic 在高速缓存描述符内的偏移量；检查程序就依赖它们相同的值。

图 6-4 显示了高速缓存描述符与 slab 描述符之间的主要关系。全满的 slab 在部分满的 slab 之前，而部分满的 slab 又在空 slab 之前。

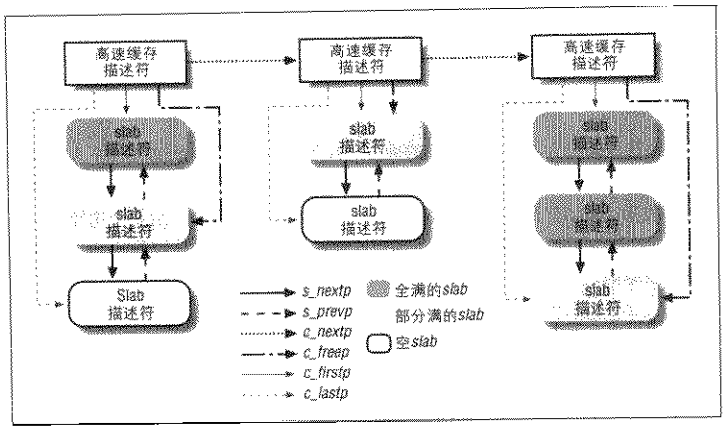


图 6-4 高速缓存描述符与 slab 描述符之间的关系

通用和专用高速缓存

高速缓存被分为两种类型：通用和专用。通用高速缓存只由slab分配器用于自己的目的，而专用高速缓存由内核的其余部分使用。

通用高速缓存是：

- 第一个高速缓存包含由内核使用的其余高速缓存的高速缓存描述符。cache_cache变量包含第一个高速缓存的描述符。
- 第二个高速缓存包含没有存放在slab内的slab描述符。cache_slabp变量指向第二个高速缓存的描述符。
- 另外的十三个高速缓存包含几何分布的内存区。一个叫做cache_sizes的表（其元素类型为cache_sizes_t）分别指向13个高速缓存描述符，与其相关的内存区大小为32，64，128，256，512，1024，2048，4096，8192，16384，32768，65536和131072字节。对于给定的大小，可以使用cache_sizes表来导出相应高速缓存的地址。

在系统初始化期间调用kmem_cache_init()和kmem_cache_sizes_init()来建立通用高速缓存。

专用高速缓存是由kmem_cache_create()函数创建的。这个函数首先根据参数确定处理新高速缓存的最佳方法（例如，是包含在slab描述符内部还是外面）；然后为新的高速缓存创建一个新的高速缓存描述符，并把这个描述符插入到cache_cache通用高速缓存中。应该注意，一旦一个高速缓存被创建，就不能撤消它。

所有通用和专用高速缓存的名字都可以在运行期间通过读取*/proc/slabinfo*文件得到。这个文件也说明每个高速缓存中空闲对象的个数和已分配对象的个数。

slab分配器与伙伴系统的接口

当slab分配器创建新的slab时，它依靠伙伴系统算法来获得一组空闲连续的页框。为了达到这个目的，它调用kmem_getpages()函数：

```
void * kmem_getpages(kmem_cache_t *cachep,
                     unsigned long flags, unsigned int *dma)
{
    void    *addr;
    *dma = flags & SLAB_DMA;
    addr = (void*) __get_free_pages(flags, cachep->c_gfporder);
    if (!*dma && addr) {
        struct page *page = mem_map + MAP_NR(addr);
        *dma = 1<<cachep->c_gfporder;
        while ((*dma--) {
            if (!PageDMA(page)) {
                *dma = 0;
                break;
            }
            page++;
        }
    }
    return addr;
}
```

其参数的含义如下：

cachep

如果高速缓存需要另外的页框，那么，cachep就指向这样的高速缓存的高速缓存描述符（请求页框的个数存放在cachep->c_gfporder域）。

Flags

说明如何请求页框（参见本章前面“请求和释放页框”一节）。

dma

指向一个变量，如果已分配的页框适合于ISA DMA，那么，由kmem_getpages()把这个变量置为1。

在相反的操作中，通过调用kmem_freepages()函数可以释放分配给slab分配器的页框（参见本章后面“从高速缓存中释放slab”一节）：

```
void kmem_freepages(kmem_cache_t *cachep, void *addr)
{
    unsigned long i = (1<<cachep->c_gfporder);
    struct page *page = &mem_map[MAP_NR(addr)];
    while (i--) {
        PageClearSlab(page);
        page++;
    }
    free_pages((unsigned long)addr, cachep->c_gfporder);
}
```

这个函数从地址addr开始释放页框，这些页框曾分配给由cachep标识的高速缓存中的slab。

给高速缓存分配slab

一个新创建的高速缓存没有包含任何slab，因此也没有空闲的对象。只有当以下两个条件都为真时，才给高速缓存分配slab：

- 已发出一个分配新对象的请求。
- 高速缓存不包含任何空闲对象。

当这种情况发生时，slab分配器通过调用kmem_cache_grow()函数给高速缓存分配一个新的slab。而这个函数调用kmem_getpages()从伙伴系统获得一组页框；然后又调用kmem_cache_slabmgmt()获得一个新的slab描述符。接下来，还要调用kmem_cache_init_objs()为新slab中所包含的所有对象申请构造方法（如果定义）。最后调用kmem_slab_link_end()把这个slab描述符插入到高速缓存slab链表的末尾：

```
void kmem_slab_link_end(kmem_cache_t *cachep,
                        kmem_slab_t *slabp)
{
    kmem_slab_t *lastp = cachep->c_lastp;
    slabp->s_nextp = kmem_slab_end(cachep);
    slabp->s_prevp = lastp;
    cachep->c_lastp = slabp;
    lastp->s_nextp = slabp;
}
```

kmem_slab_end宏产生相应高速缓存描述符中c_offset域的地址(如前所述,slab链表的最后一个元素指向那个域)。

把新的slab描述符插入到链表后，kmem_cache_grow()用这个高速缓存描述符的地址和新的slab描述符的地址，分别装载新slab中所包含的所有页框描述符的next和prev域。这项工作不会出错，因为只有当页框空闲时伙伴系统的函数才会使用next和prev域，而slab分配器函数所处理的页框不是空闲的。因此，页框描述符的这种专门用法不会搞乱伙伴系统。

从高速缓存中释放slab

如前所述，slab分配器不能靠自己释放空slab中的页框。事实上，只有在下列条件成立的情况下才能释放slab:

- 伙伴系统不能满足新请求的一组页框。
- slab为空，也就是说，slab中包含的所有对象都是空闲的。

当内核查找另外的空闲页框时，就调用try_to_free_pages()。这个函数又可以调用kmem_cache_reap()来选择一个至少包含一个空slab的高速缓存。然后kmem_slab_unlink()函数从slab的高速缓存链表中删除这个slab:

```
void kmem_slab_unlink(kmem_slab_t *slabp)
{
    kmem_slab_t *prevp = slabp->s_prevp;
    kmem_slab_t *nextp = slabp->s_nextp;
    prevp->s_nextp = nextp;
    nextp->s_prevp = prevp;
}
```

随后，调用kmem_slab_destroy()撤消slab（以及其中的对象）：

```
void kmem_slab_destroy(kmem_cache_t *cachep, kmem_slab_t *slabp)
{
    if (cachep->c_dtor) {
        unsigned long num = cachep->c_num;
        void *objp = slabp->s_mem;
        do {
            (cachep->c_dtor)(objp, cachep, 0);
            objp += cachep->c_offset;
            if (!slabp->s_index)
                objp += sizeof(kmem_bufctl_t);
        } while (--num);
    }
    slabp->s_magic = SLAB_MAGIC_DESTROYED;
    if (slabp->s_index)
        kmem_cache_free(cachep->c_index_cachep, slabp->s_index);
    kmem_freepages(cachep, slabp->s_mem-slabp->s_offset);
    if (SLAB_OFF_SLAB(cachep->c_flags))
        kmem_cache_free(cache_slabp, slabp);
}
```

这个函数检查高速缓存是否为它的对象提供了析构方法（c_dtor域不为NULL），如果是，就使用析构方法释放slab中的所有对象。objp局部变量记录当前已检查的对象。接下来，又调用kmem_freepages()，该函数返回由slab给伙伴系统使用的所有连续页框。最后，如果slab描述符被存放在slab的外面（在这种情况下，s_index和c_index_cachep域不为空，正如本章后面所介绍），那么，就从slab描述符的高速缓存释放这个slab描述符。

Linux的一些模块（参见附录二）可以创建高速缓存。为了避免浪费内存空间，内核必须在删除一个模块前撤消由该模块创建的所有高速缓存中的所有slab（注2）。kmem_cache_shrink()函数通过反复调用kmem_slab_destroy()来撤消高速缓存中的所有slab。当另一个内核控制路径试图为自己分配一个新的slab时，高速缓存描述中的c_growing域被用来避免kmem_cache_shrink()缩小高速缓存。

注2：我们在前面已经说过，Linux并不撤消高速缓存。这样，当链接一个新模块时，内核必须检查所需的新高速缓存描述符是否已在以前的该模块或其他模块安装中被创建。

对象描述符

每个对象都有类型为 struct kmem_bufctl_s（等价于 kmem_bufctl_t 类型）的一个描述符。与 slab 描述符本身类似，slab 的对象描述符也可以用两种可能的方式来存放，如图 6-5 所示。

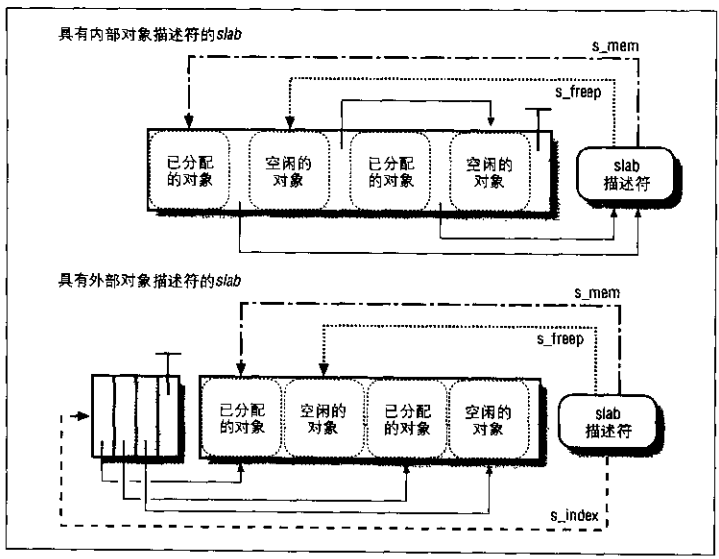


图 6-5 slab 描述符与对象描述符之间的关系

外部对象描述符

存放在 slab 的外面，由 cache_sizes 指向的一个通用高速缓存中。在这种情况下，内存区的第一个对象描述符描述的是 slab 中的第一个对象，依此类推。内存区的大小（尤其是存放对象描述符的通用高速缓存）取决于在 slab 中所存放的对象个数（高速缓存描述符的 c_num 域）。对象本身所在的高速缓存被连到它们的描述符所在的高速缓存，这是通过两个域链接的。首先，slab 所在的高速缓存的 c_index_cachep 域指向对象描述符所在的高速缓存的高速缓存描述符。其次，slab 描述符的 s_index 域指向对象描述符所在的内存区。

内部对象描述符

存放在slab内部，正好在描述符所描述的对象之后。在这种情况下，高速缓存描述符的c_index_cachep域和slab描述符的s_index域都为NULL。

当对象的大小是512、1024、2048或4096的倍数时，slab分配器选择第二种方案。在这种情况下，在slab内存放控制结构将导致大的内碎片。如果对象的大小小于512字节或不是512、1024、2048或4096的倍数，那么，slab分配器在slab内部存放对象描述符。

对象描述符是由一个单独的域组成的简单结构：

```
typedef struct kmem_bufctl_s {
    union {
        struct kmem_bufctl_s * buf_nextp;
        kmem_slab_t *          buf_slabp;
        void *                 buf_objp;
    } u;
} kmem_bufctl_t;
#define    buf_nextp    u.buf_nextp
#define    buf_slabp    u.buf_slabp
#define    buf_objp     u.buf_objp
```

这个域的含义与对象的状态及对象描述符的位置有关：

buf_nextp

如果对象空闲，buf_nextp就指向slab中的下一个对象，因此在slab内实现了一个简单的空闲对象链表。

buf_objp

如果对象被分配，且它的对象描述符存放在slab的外面，buf_objp就指向这个对象。

buf_slabp

如果对象被分配，且它的对象描述符存放在slab的内部，buf_slabp就指向对象所在的slab的slab描述符。这表示slab描述符是在slab的外部还是内部。

图6-5显示了slab，slab描述符，对象及对象描述符之间的关系。注意，尽管这个图说明slab描述符存放在slab的外面，但是，如果描述符存放在slab的内部，这个图仍然不变。

对齐内存中的对象

slab分配器所管理的对象可以在内存进行对齐，也就是说，存放它们的起始物理地址是一个给定常量的倍数，通常是2的倍数。这个常量就叫对齐因子（alignment factor），它的值存放在高速缓存描述符的c_align域。存放对象大小的c_offset域要考虑所增加的填充字节数以获得适当的对齐。如果c_align的值为0，说明这个对象不需要对齐。

slab分配器所允许的最大对齐因子是4096，即页框大小。这就意味着参考对象的物理地址或线性地址就可以对齐对象。在这两种情况下，只有最低的12位才可以通过对齐来改变。

通常情况下，如果内存单元的物理地址是字大小（即计算机的内部内存总线的宽度）对齐的，那么，微机对内存单元的存取会非常快。因此，kmem_cache_create()函数试图根据BYTES_PER_WORD宏所指定的字大小来对齐对象。对于Intel Pentium处理器，这个宏产生的值为4，因为字长是32位。然而，如果这会导致内存的浪费，这个函数就不对齐这些对象。

当创建一个新的高速缓存时，就可以让它所包含的对象在第一级高速缓存中对齐。为了做到这点，设置SLAB_HWCACHE_ALIGN高速缓存描述符标志。kmem_cache_create()函数按如下方式处理请求：

- 如果对象的大小大于高速缓存行（cache line）的一半，就在RAM中把这个对象的大小对齐到L1_CACHE_BYTES的倍数，也就是行的开始。
- 否则，这个对象的大小就是L1_CACHE_BYTES的因子取整。这可以保证一个对象不会横跨两个高速缓存行。

显然，slab分配器在这里所做的事情就是以内存空间换取访问时间，即通过人为地增加对象的大小来获得较好的高速缓存性能，由此也引起额外的内碎片。

slab着色

从第二章我们知道，同一硬件高速缓存行可以映射RAM中很多不同的块。在本章我们也已看到，相同大小的对象倾向于存放在高速缓存内相同的偏移量处。在不同的slab内具有相同偏移量的对象最终很可能映射在同一高速缓存行中。高速缓存的

硬件可能因此而花费内存周期在同一高速缓存行与RAM内存单元之间来来往往传送两个对象，而其他的高速缓存行并未充分使用。slab分配器通过一种叫做slab着色（slab coloring）的策略，尽量降低高速缓存的这种不愉快行为：把任意不同的称为颜色的值分配给slab。

在考察slab着色之前，我们先看一下高速缓存内对象的布局。让我们考虑某个高速缓存，它的对象在RAM中被对齐。因此，高速缓存的c_align域为正值，比如说*aln*。连对齐的约束考虑在内，在slab内放置对象就有很多种可能的方式。方式的选择取决于对下列变量所做的决定：

num

可以在slab中存放的对象个数（其值在高速缓存描述符的c_num域）。

osize

具有对齐字节的对象大小（其值在c_offset域）加上对象描述符的大小（如果slab内有描述符）。

dsize

slap描述符的大小。如果slab描述符被存放在slap的外部，它的值等于0。

free

在slab内未用字节的个数（没有分配给任一对象的字节）。

一个slab中的总字节长度可以表示为如下表达式：

slab的长度 = *(num* × *osize)+dsize +free*

*free*总是小于*osize*，因为否则的话，就有可能把另外的对象放在slab内。不过，*free*可以大于*aln*。

slab分配器利用空闲未用的字节来对slab着色。术语“颜色”只是用来再分slab，并允许内存分配器把对象展开在不同的线性地址之中。这样的话，内核从微处理器的硬件高速缓存中可能获得最好性能。具有不同颜色的slab把slab的第一个对象存放在不同的内存单元，同时满足对齐约束。可用颜色的个数是*free/aln*+1。第一个颜色表示为0，最后一个颜色（它的值在高速缓存的c_colour域）表示为*free/aln*。

如果用颜色*col*对一个slab着色，那么，第一个对象的偏移量（相对于slab的起始地址）就等于*col* × *aln*字节，这个值存放在slab描述符的s_offset域。图6-6显

示了slab内对象的布局对slab颜色的依赖情况。着色本质上导致把slab中的一些空闲区域从末尾移到开始。

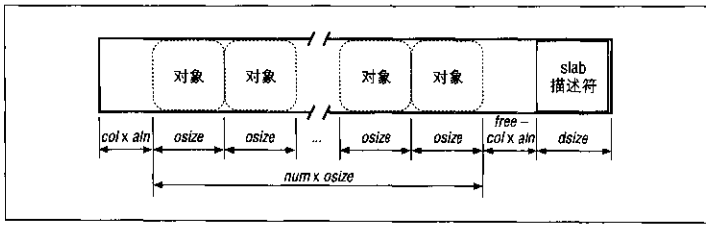


图6-6 具有颜色col与对齐aln的slab

只有当空闲足够大时，着色才起作用。显然，如果对象没有请求对齐，或者如果slab内的未用字节数小于所请求的对齐（*free* < *aln*），那么，唯一可能着色的slab就是具有颜色0的slab，也就是说，把这个slab的第一个对象的偏移量赋为0。

通过把当前颜色存放在高速缓存描述符的c_colour_next域，各种颜色在一个给定对象类型的slab之间公平地分布。kmem_cache_grow()函数把c_colour_next所表示的颜色赋给一个新的slab，并递减这个域的值。当c_colour_next的值变为0后，又封装为最大可用值。

```
if (!(offset = cachep->c_colour_next--))
    cachep->c_colour_next = cachep->c_colour;
offset *= cachep->c_align;
slabp->s_offset = offset;
```

因此，每个slab的创建所用的颜色与前一个都不同，一直到最大可用颜色。

给高速缓存分配对象

新的对象可以通过调用kmem_cache_alloc()函数而得到。参数cachep指向新空闲对象所在的高速缓存描述符。kmem_cache_alloc()首先检查是否存在高速缓存描述符，然后从c_freep域检索到第一个slab（至少要包含一个空闲对象）的s_nextp域的地址：

```
slabp = cachep->c_freep;
```

如果slabp没有指向一个slab，就跳转到alloc_new_slab并调用kmem_cache_grow()，给高速缓存增加一个新的slab：

```
if (slabp->s_magic != SLAB_MAGIC_ALLOC)
    goto alloc_new_slab;
```

s_magic域的SLAB_MAGIC_ALLOC值表示slab至少包含一个空闲对象。如果slab为满，slabp就指向cachep->c_offset域，因此slabp->s_magic与cachep->c_magic的值一致。但是，在这种情况下，高速缓存的这个域包含的魔数与SLAB_MAGIC_ALLOC不同。

在获得一个具有空闲对象的slab之后，kmem_cache_alloc()函数对slab中当前已分配对象的个数加1：

```
slabp->s_inuse++;
```

然后，用slab内第一个空闲对象的地址装载bufp，同时相应地更新slab描述符的slabp->s_freep域，让它指向下一个空闲对象：

```
bufp = slabp->s_freep;
slabp->s_freep = bufp->buf_nextp;
```

如果slabp->s_freep变为NULL，说明这个slab不再包含空闲对象，因此，必须更新高速缓存描述符的c_freep域：

```
if (!slabp->s_freep)
    cachep->c_freep = slabp->s_nextp;
```

注意，没有必要改变链表内slab描述符的位置，因为它仍然部分地排序。现在，这个函数必须导出空闲对象的地址并更新对象描述符。

如果slabp->s_index域为空，对象描述符就正好存放在slab内这个对象的后面。在这种情况下，把slab描述符的地址存放在对象描述符唯一的域中以表示这个对象不再空闲；然后，通过从对象描述符的地址减去这个对象的大小（包含在cachep->c_offset域中）就可以导出这个对象的地址：

```
    if (!slabp->s_index) {
        bufp->buf_slabp = slabp;
        objp = ((void*)bufp) - cachep->c_offset;
}
```

如果slabp->s_index域不为0，它就指向对象描述符所在的slap外部的一个内存区。在这种情况下，这个函数首先计算对象描述符在这个内存区外的相对位置；然后用对象的大小乘以这个数；最后，把乘的结果与slab中第一个对象的地址相加，因此得到要返回对象的地址。与前一种情况一样，对象描述符唯一的域被更新，现在指向新对象：

```
if (slabp->s_index) {
    objp = ((bufp-slabp->s_index)*cachep->c_offset) +
                slabp->s_mem;
    bufp->buf_objp = objp;
}
```

函数通过返回新对象的地址而结束：

```
return objp;
```

从高速缓存释放对象

kmem_cache_free()函数释放由slab分配器以前所获得的对象。它的参数为cachep和objp，前者是高速缓存描述符的地址，而后者是被释放对象的地址。这个函数首先检查参数，然后确定这个对象描述符的地址和对象所在的slab的地址。它利用cachep->c_flags标志（包含在高速缓存描述符中）来确定这个对象描述符是位于slab的内部还是外部。

在前一种情况下，通过把对象的大小与它的起始地址相加来确定这个对象描述符的地址。然后，从这个对象描述符适当的域中减去slab描述符的地址：

```
if (!SLAB_BUFCTL(cachep->c_flags)) {
    bufp = (kmem_bufctl_t *)(objp+cachep->c_offset);
    slabp = bufp->buf_slabp;
}
```

在后一种情况下，函数对slab描述符地址的确定，是通过对象所在页框的描述符中的prev域而得到的（这里prev的作用指的是前面“给高速缓存分配slab”中所指的那种prev的作用）。对象描述符的地址可以通过计算这个对象在slab内的第一个序列号而得到（对象的地址减去第一个对象的地址后再除以对象的长度）。然后，从slab描述符的slabp->s_index域所指的外部区域的起点开始，用这个数来确定

对象描述符的位置。为了安全起见，函数还要检查对象的地址，即将对象地址作为一个参数传递给函数，看这个对象描述符本该有的地址是否与这个参数相一致：

```
if (!SLAB_BUFCTL(cachep->c_flags)) {
    slabp = (kmem_slab_t *)((&mem_map[MAP_NR(objp)])->prev);
    bufp = &slabp->s_index[(objp - slabp->s_mem) /
                           cachep->c_offset];
    if (objp != bufp->buf_objp)
        goto bad_obj_addr;
}
```

现在，函数检查slab描述符的slabp->s_magic域是否包含正确的魔数，以及slabp->s_inuse域的值是否大于0。如果一切都正常，就递减slabp->s_inuse的值，并把这个对象插入到空闲对象的slab链表中：

```
slabp->s_inuse--;
bufp->buf_nextp = slabp->s_freep;
slabp->s_freep = bufp;
```

如果bufp->buf_nextp为NULL，说明空闲对象的链表只包含一个元素，即要释放的那个对象。在这种情况下，这个slab以前被填充到一定容量，因此，也许有必要把它的slab描述符重新插入到slab描述符链表的一个新位置。（回忆一下，在部分排序的链表中，完全填充的slab出现在具有一些空闲对象的slab之前。）这是通过kmem_cache_one_free()函数完成的：

```
if (!bufp->buf_nextp)
    kmem_cache_one_free(cachep, slabp);
```

如果除了正被释放的对象外，slab还包含其他空闲对象，就有必要检查所有的对象是否都空闲。与以前的情况一样，就有必要把这个slab重新插入到slab描述符链表的一个新位置。移动操作由kmem_cache_full_free()函数完成：

```
if (bufp->buf_nextp)
    if (!slabp->s_inuse)
        kmem_cache_full_free(cachep, slabp);
```

kmem_cache_free()函数到此结束。

通用对象

正如“伙伴系统算法”中所描述的一样，如果对内存区的请求不频繁，就用一组通用高速缓存来处理，通用高速缓存中的对象具有几何分布的大小，范围从32到131072字节。

调用kmalloc()函数就可以得到这种类型的对象：

```
void * kmalloc(size_t size, int flags)
{
    cache_sizes_t *csizep = cache_sizes;
    for (; csizep->cs_size; csizep++) {
        if (size > csizep->cs_size)
            continue;
        return __kmem_cache_alloc(csizep->cs_cachep, flags);
    }
    printk(KERN_ERR "kmalloc: Size (%lu) too large\n",
                    (unsigned long) size);
    return NULL;
}
```

函数使用cache_sizes表来确定适当大小的对象所在高速缓存的描述符的位置。然后调用kmem_cache_alloc()来确定对象（注3）的位置。

调用kmalloc()所获得的对象可以通过调用kfree()（注4）来释放：

```
void kfree(const void *objp)
{
    struct page *page;
    int    nr;
    if (!objp)
        goto null_ptr;
    nr = MAP_NR(objp);
    if (nr >= num_physpages)
        goto bad_ptr;
```

注3：实际上，为了效率起见，kmem_cache_alloc()的代码被拷贝到kmalloc()函数体内。实现kmem_cache_alloc()的__kmem_cache_alloc()函数被内嵌声明。

注4：有一个类似的函数叫kfree_s()，它请求另外一个参数，即被释放对象的大小。这个函数用在Linux以前的版本中。释放内存区前必须确定它的大小。文件系统的一些模块还在使用它。

```
        page = &mem_map[nr];
        if (PageSlab(page)) {
            kmem_cache_t    *cachep;
            cachep = (kmem_cache_t *)(page->next);
            if (cachep && (cachep->c_flags & SLAB_CFLGS_GENERAL)) {
                __kmem_cache_free(cachep, objp);
                return;
            }
        }
    bad_ptr:
        printk(KERN_ERR "kfree: Bad obj %p\n", objp);
        *(int *) 0 = 0; /* 强制产生核心转储 */
    null_ptr:
        return;
    }
```

要识别一个适当的高速缓存描述符，可以通过读取内存区所在的第一个页框描述符的next域来实现。如果这个域指向一个有效的描述符，那么，通过调用kmem_cache_free()来释放相应的内存区（注5）。

非连续内存区管理

从前面的讨论中我们已经知道，把内存区映射到一组连续的页框是最好的选择，这样会充分地利用高速缓存并达到较低的平均访问时间。不过，如果对内存区的请求不是很频繁，那么，考虑通过连续的线性地址来访问非连续的页框的一种分配模式就会很有意义。这种模式的主要优点是避免了外碎片，而缺点是必须打乱内核页表。显然，非连续内存区的大小必须是4096的倍数。Linux保守地使用非连续内存区，例如，为活动的交换区分配数据结构（参见第十六章中的“激活和使交换区无效”一节），为模块分配空间（参见附录二），或者给某些I/O驱动程序分配缓冲区。

非连续内存区的线性地址

要查找线性地址的一个空闲区，我们可以从PAGE_OFFSET开始查找（通常为0xc0000000，第4G字节的起始地址）。我们在第二章的“进程页表”一节中已了解到，内核保留整个上部内存区，用以映射可用的RAM供自己使用。但是，可用

注5：对于kmalloc()、kmem_cache_free()的代码在kfee()内部被复制。实现kmem_cache_free()的__kmem_cache_free()被内嵌声明。

RAM 只占从 PAGE_OFFSET 开始 1G 字节很少的部分。在那个保留区之上的所有线性地址都可以用来映射非连续内存区。与物理地址末尾对应的线性地址保存在 high_memory 变量中。

图 6-7 显示如何把线性地址分配给非连续的内存区。在物理地址的末尾与第一个内存区之间插入一个大小为 8MB（宏 VMALLOC_OFFSET）的安全区，目的是为了“捕获”对内存的非法访问。出于同样的理由，插入其他 4K 大小的安全区来隔离非连续的内存区。

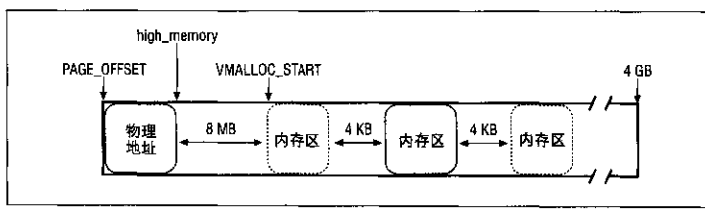


图 6-7 从 PAGE_OFFSET 开始的线性地址区间

为非连续内存区保留的线性地址空间的起始地址由 VMALLOC_START 宏定义，其定义如下：

```
#define VMALLOC_START (((unsigned long) high_memory + \
                        VMALLOC_OFFSET) & ~(VMALLOC_OFFSET-1))
```

非连续内存区的描述符

每个非连续区都有一个类型为 struct vm_struct 的描述符：

```
struct vm_struct {
    unsigned long flags;
    void * addr;
    unsigned long size;
    struct vm_struct * next;
};
```

通过 next 域，这些描述符被插入到一个简单的链表中，链表第一个元素的地址存放在 vmlist 变量中。addr 域包含内存区第一个内存单元的线性地址。size 域包含内存区的大小加 4096（即前面提到的安全区间的大小）。

get_vm_area()函数创建类型为struct vm_struct的新描述符，它的参数size指定新内存区的大小：

```
struct vm_struct * get_vm_area(unsigned long size)
{
    unsigned long addr;
    struct vm_struct **p, *tmp, *area;
    area = (struct vm_struct *) kmalloc(sizeof(*area),
                                        GFP_KERNEL);
    if (!area)
        return NULL;
    addr = VMALLOC_START;
    for (p = &vmlist; (tmp = *p) ; p = &tmp->next) {
        if (size + addr < (unsigned long) tmp->addr)
            break;
        addr = tmp->size + (unsigned long) tmp->addr;
        if (addr > 0xffffd000-size) {
            kfree(area);
            return NULL;
        }
    }
    area->addr = (void *)addr;
    area->size = size + PAGE_SIZE;
    area->next = *p;
    *p = area;
    return area;
}
```

函数首先调用kmalloc()为新描述符获得一个内存区。然后扫描类型为struct vm_struct的描述符链表，查找一个可用线性地址区间（至少包含size+4096个地址）。如果存在这样的一个区间，函数就初始化这个描述符的域，并通过返回这个非连续区的起始地址而结束。否则，当addr + size超过4GB的界限，get_vm_area()释放这个描述符并返回NULL。

分配非连续内存区

vmalloc()函数给内核分配一个非连续内存区。参数size表示所请求内存区的大小。如果这个函数能够满足请求，就返回新内存区的起始地址；否则，返回一个NULL指针。

```
void * vmalloc(unsigned long size)
{
    void * addr;
    struct vm_struct *area;
    size = (size+PAGE_SIZE-1)&PAGE_MASK;
    if (!size || size > (num_physpages << PAGE_SHIFT))
        return NULL;
    area = get_vm_area(size);
    if (!area)
        return NULL;
    addr = area->addr;
    if (vmalloc_area_pages((unsigned long) addr, size)) {
        vfree(addr);
        return NULL;
    }
    return addr;
}
```

函数首先把size参数取整为4096（页框大小）的一个倍数。再执行有效性检查，以确认size大于0且小于或等于现有页框数。如果大小适合可用内存，vmalloc()调用get_vm_area()，后者创建一个新的描述符并返回分配给这个内存区的线性地址。然后，vmalloc()调用vmalloc_area_pages()以请求非连续的内存区，并通过返回非连续区的起始地址而结束。

vmalloc_area_pages()函数使用两个参数：address，内存区的线性起始地址；size，内存区的大小。内存区的末尾线性地址被赋给end局部变量：

```
end = address + size;
```

然后函数使用pgd_offset_k宏来导出这个内存区的起始线性地址在页全局目录中的目录项：

```
dir = pgd_offset_k(address);
```

函数执行下列循环：

```
while (address < end) {
    pmd_t *pmd = pmd_alloc_kernel(dir, address);
    if (!pmd)
        return -ENOMEM;
    if (alloc_area_pmd(pmd, address, end - address))
        return -ENOMEM;
```

```
        set_pgdir(address, *dir);
        address = (address + PGDIR_SIZE) & PGDIR_MASK;
        dir++;
    }
```

每次循环，都首先调用pmd_alloc_kernel()来为新内存区创建一个页中间目录。然后调用alloc_area_pmd()为新的页中间目录分配所有相关的页表。接下来，调用set_pgdir()更新新的页中间目录在现有的页全局目录中对应的所有目录项(参见第二章的“进程性页表”一节)。把常量2^{22}与address的当前值相加(2^{22}就是一个页中间目录所横跨的线性地址范围的大小)，最后增加指向页全局目录的指针dir。

循环结束的条件是：指向非连续区的页表项全被建立。

alloc_area_pmd()函数执行一个类似的循环，不过，这是针对页中间目录所指向的所有页表的：

```
    while (address < end) {
        pte_t * pte = pte_alloc_kernel(pmd, address);
        if (!pte)
            return -ENOMEM;
        if (alloc_area_pte(pte, address, end - address))
            return -ENOMEM;
        address = (address + PMD_SIZE) & PMD_MASK;
        pmd++;
    }
```

pte_alloc_kernel()函数(参见第二章中的“页表的处理”一节)分配一个新的页表，并更新页中间目录中相应的目录项。地址值增加2^{22}(2^{22}就是一个页表所横跨的线性地址区间的大小)，并且循环被反复执行。

alloc_area_pte()的主循环为：

```
    while (address < end) {
        unsigned long page;
        if (!pte_none(*pte))
            printk("alloc_area_pte: page already exists\n");
        page = __get_free_page(GFP_KERNEL);
        if (!page)
            return -ENOMEM;
        set_pte(pte, mk_pte(page, PAGE_KERNEL));
```

```
        address += PAGE_SIZE;
        pte++;
    }
```

每个页框是通过__get_free_page()进行分配的。通过set_pte和mk_pte宏把新页框的物理地址写进页表。把常量4096（即一个页框的长度）加到address上之后，循环体又被反复执行。

释放非连续内存区

vfree()函数释放非连续的内存区。它的参数addr包含要释放内存区的起始地址。vfree()首先扫描由vmlist指向的链表以查找要释放内存区的内存区描述符的地址：

```
for (p = &vmlist ; (tmp = *p) ; p = &tmp->next) {
    if (tmp->addr == addr) {
        *p = tmp->next;
        vmfree_area_pages((unsigned long)(tmp->addr),
                          tmp->size);
        kfree(tmp);
        return;
    }
}
printk("Trying to vfree() nonexistent vm area (%p)\n", addr);
```

描述符的size域指定要释放内存区的大小。内存区本身的释放是通过调用vmfree_area_pages()完成的，而描述符的释放是通过调用kfree()完成的。

vmfree_area_pages()函数有两个参数：起始线性地址和内存区的大小。它通过执行下列的循环以反转vmalloc_area_pages()操作的结果：

```
while (address < end) {
    free_area_pmd(dir, address, end - address);
    address = (address + PGDIR_SIZE) & PGDIR_MASK;
    dir++;
}
```

free_area_pmd()依次在循环中执行alloc_area_pmd()的反操作：

```
while (address < end) {
    free_area_pte(pmd, address, end - address);
```

```
        address = (address + PMD_SIZE) & PMD_MASK;
        pmd++;
    }
```

free_area_pte()又在循环中执行alloc_area_pte()的反操作:

```
    while (address < end) {
        pte_t page = *pte;
        pte_clear(pte);
        address += PAGE_SIZE;
        pte++;
        if (pte_none(page))
            continue;
        if (pte_present(page)) {
            free_page(pte_page(page));
            continue;
        }
        printk("Whee... Swapped out page in kernel page table\n");
    }
```

分配给非连续内存区的每个页框是通过伙伴系统的free_page()函数来释放的。由pte_clear把页表中的相应表项置为0。

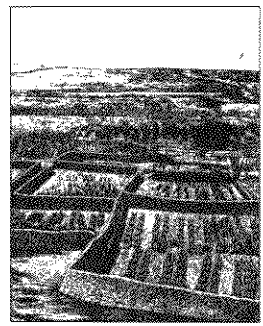
对 Linux 2.4 的展望

Linux 2.2有两个伙伴系统：第一个处理适合ISA DMA的页框，而第二个处理不适合ISA DMA的页框。Linux 2.4为高地址物理内存（即内核不永久映射的页框）增加了第三个伙伴系统。使用高内存页框意味着要在一个特殊的内核页表中改变一个页表项，以把页框的物理地址映射在4GB的线性地址空间中。

实际上，Linux 2.4把RAM的这三个部分看作不同的“地区（zone）”。每个“地区”有它自己的计数器和水印（watermark），以监控空闲页框的个数。当一个内存分配的请求发生时，内核首先试图从最适当的“地区”得到页框；如果失败，内核就求助于另一个“地区”。slab分配器基本上未变。然而，Linux 2.4允许撤消不再有用的slap分配器高速缓存。回想一下在Linux 2.2中，slab分配器能被动态创建但不能被撤消。模块在被装载时创建属于自己的slab高速缓存，而现在希望当模块被卸载时同时也撤消相应的高速缓存。

第七章

进程地址空间



在前一章我们已经看到，内核中的函数可以以相当直接了当的方式获得动态内存，就是通过调用几个函数中的一个：__get_free_pages()使用伙伴（buddy）系统算法获得页框，kmem_cache_alloc()或kmalloc()使用slab分配器为专用或通用对象分配块，以及vmalloc()获得一块非连续的内存区。如果所请求的内存区能被满足，这些函数都返回一个线性地址，这个线性地址就是所分配内存动态区的起始地址。

使用这些简单方法是基于以下两个原因：

- 内核是操作系统中优先级最高的组成部分。如果某个内核函数做出对动态内存的请求，那么，必定有某一正当的理由来发布这个请求，并且内核并不打算推迟这个请求。
- 内核信任自己。所有的内核函数都被假定是没有错误的，因此内核函数不必插入针对编程错误的任何保护措施。

当给用户态进程分配内存时，情况完全不一样：

- 进程对动态内存的请求被认为是不紧迫的。例如，当进程的可执行文件被装入时，进程并不一定马上对所有的代码页进行访问。同理，当进程调用malloc()以获得另外的动态内存时，也并不意味着进程很快就会访问所有所获得的内存。因此，一般来说，内核总是尽量推迟给用户态进程分配动态内存。

- 由于用户进程是不可信任的，因此，内核必须能随时捕获用户态进程所引起的所有寻址错误。

本章我们会看到，内核使用一种新的资源成功实现了对进程动态内存的推迟分配。当用户态进程请求动态内存时，并没有获得另外的页框，而仅仅获得对一个新的线性地址区间的使用权，而这一线性地址区间就成为进程地址空间的一部分。这一区间被称为一个线性区（memory region，译注1）。

首先，我们从“进程的地址空间”一节开始来讨论进程是怎样看待动态内存的。然后，在“线性区”一节中描述进程地址空间的基本组成。接下来，我们仔细分析缺页异常处理程序在推迟给进程分配页框中所起的作用。然后，我们阐述内核怎样创建和删除进程的整个地址空间。最后，我们讨论与进程的地址空间管理有关的API和系统调用。

进程的地址空间

进程的地址空间由允许进程使用的全部线性地址组成。每个进程所看到的线性地址集合是不同的，一个进程所使用的地址与另外一个进程所使用的地址之间没有什么关系。后面我们会看到，内核可以通过增加或删除某些线性地址区间而动态地修改进程的地址空间。

内核通过一种称为线性区的资源来表示线性地址区间，线性区是由起始线性地址、长度和一些存取权限来描述的。为了效率起见，起始地址和线性区的长度都必需是4096的倍数，以便用每个线性区所识别的数据完全填满分配给它的页框。我们简要提及进程获得新线性区的一些典型情况：

- 当用户在控制台输入一条命令时，shell进程创建一个新的进程去执行这个命令。结果是，一个全新的地址空间，也就是一组线性区，分配给了这个新的进程（参见本章后面的“创建和删除进程的地址空间”一节和第十九章）。

译注1：“memory region”字面含义为内存区，但实际含义为线性地址空间中的一个区域，在此把它译为线性区更易于理解。

- 一个正在运行的进程有可能决定装入一个完全不同的程序。在这种情况下，进程标识符仍然保持不变，可是在装入这个程序以前所使用的线性区却被释放，并有一组新的线性区被分配给这个进程（参见第十九章中的“exec类函数”一节）。
- 一个正在运行的进程可能对一个文件（或它的一部分）执行“内存映射”。在这种情况下，内核给这个进程分配一个新的线性区来映射这个文件（参见第十五章中的“内存映射”一节）。
- 进程可能持续向它的用户态堆栈增加数据，直到映射这个堆栈的线性区用完为止。在这种情况下，内核也许决定扩展这个线性区的大小（参见本章后面的“缺页异常处理程序”一节）。
- 进程可能创建一个IPC共享线性区来与其他合作进程共享数据。在这种情况下，内核给这个进程分配一个新的线性区以实现这个方案（参见第十八章中的“IPC共享内存”一节）。
- 进程可能通过调用类似malloc()这样的函数扩展自己的动态区（堆）。结果是，内核可能决定扩展给这个堆所分配的线性区（参见本章后面的“堆的管理”一节）。

表7-1显示了与前面提到的任务相关的一些系统调用。除brk()在本章的最后进行阐述外，其余的系统调用在其他章节进行阐述。

表7-1 与创建、删除线性区相关的系统调用

系统调用	描述
brk()	改变进程堆的大小
execve()	装入一个新的可执行文件，从而改变进程的地址空间
exit()	结束当前进程并撤销它的地址空间
fork()	创建一个新进程，并为它创建新的地址空间
mmap()	为一个文件创建一个内存映射，从而扩大进程的地址空间
munmap()	撤消对文件的内存映射，从而缩小进程的地址空间
shmat()	创建一个共享线性区
shmdt()	撤消一个共享线性区

我们会在“缺页异常处理程序”一节中看到，确定一个进程当前所拥有的线性区（即进程的地址空间）是内核的基本任务，因为这可以让缺页异常处理程序有效地区分引发这个异常处理程序的两种不同类型的无效线性地址。

- 由编程错误引发的无效线性地址。
- 由缺页引发的无效线性地址；即使这个线性地址属于进程的地址空间，对应于这个地址的页框仍然有待分配。

从进程的观点来看，后一种地址不是无效的。内核通过提供页框来处理这种缺页，并让进程继续执行。

内存描述符

与进程地址空间有关的全部信息都包含在由进程描述符的mm域所指向的一个表中。这个表是一个mm_struct类型的结构，如下所示：

```
struct mm_struct {
    struct vm_area_struct *mmap, *mmap_avl, *mmap_cache;
    pgd_t * pgd;
    atomic_t count;
    int map_count;
    struct semaphore mmap_sem;
    unsigned long context;
    unsigned long start_code, end_code, start_data, end_data;
    unsigned long start_brk, brk, start_stack;
    unsigned long arg_start, arg_end, env_start, env_end;
    unsigned long rss, total_vm, locked_vm;
    unsigned long def_flags;
    unsigned long cpu_vm_mask;
    unsigned long swap_cnt;
    unsigned long swap_address;
    void * segments;
};
```

为了便于讨论，对其中几个最重要的域解释如下：

pgd和segments

分别指向进程的页全局目录和局部描述符表。

rss

指定分配给进程的页框数。

total_vm

表示进程地址空间含有多少页。

locked_vm

统计“锁定”页的数目，也就是说，不能被交换出去的页数（参见第十六章）。

count

表示共享同一个struct mm_struct描述符的进程数。如果count大于1，说明几个进程是共享同一地址空间的轻量级进程，也就是说，使用同一内存描述符。

mm_alloc()函数用来获得一个新的内存描述符。由于这些描述符被保存在slab分配器高速缓存中，因此，mm_alloc()调用kmem_cache_alloc()来复制current内存描述符的内容以初始化新的内存描述符，并把count域置为1。

相反，mmput()函数递减内存描述符的count域。如果该域变为0，这个函数就释放局部描述符表、线性区描述符（参见本章后面的部分）、由内存描述符所指向的页表以及这个内存描述符本身。

其中的mmap、mmap_avl和mmap_cache域将在下节进行讨论。

线性区

Linux通过类型为vm_area_struct的描述符实现线性区：

```
struct vm_area_struct {
    struct mm_struct * vm_mm;
    unsigned long vm_start;
    unsigned long vm_end;
    struct vm_area_struct *vm_next;
    pgprot_t vm_page_prot;
    unsigned short vm_flags;
    short vm_avl_height;
    struct vm_area_struct *vm_avl_left, *vm_avl_right;
    struct vm_area_struct *vm_next_share, **vm_pprev_share;
```

```
    struct vm_operations_struct * vm_ops;
    unsigned long vm_offset;
    struct file * vm_file;
    unsigned long vm_pte;
};
```

每个线性区描述符表示一个线性地址区间。vm_start域表示区间的第一个线性地址，vm_end域表示区间之外的第一个线性地址。vm_end - vm_start表示线性区长度。vm_mm域指向拥有这个区间的进程的mm_struct内存描述符。我们稍后将描述vm_area_struct的其他域。

进程所拥有的线性区从来不重叠，并且内核尽力把新分配的线性区与紧邻的现有线性区进行合并。如果两个相邻区的存取权限相匹配，就能把它们合并在一起。

如图7-1所示，当一个新的线性地址区间被加入到进程的地址空间时，内核检查一个已经存在的线性区是否可以被扩大(情况*a*)。如果不能，创建一个新的线性区(情况*b*)。类似地，如果从进程的地址空间删除一个线性地址区间，内核就要调整受影响的线性区大小(情况*c*)。有些情况下，调整大小迫使一个线性区被分成两个更小的部分(情况*d*、注1)。

线性区数据结构

进程所拥有的所有线性区是通过一个简单的链表链接在一起的。出现在链表中的线性区是按内存地址的升序排列的；而每两个线性区可能是由未用的内存地址分开的。每个vm_area_struct元素的vm_next域指向链表的下一个元素。内核通过进程的内存描述符的mmap域来寻找特定的线性区，而mmap域指向链表中的第一个线性区描述符的vm_next域。

内存描述符的map_count域存放进程所拥有的线性区数目，一个进程可以最多拥有MAX_MAP_COUNT个不同的线性区(这个值通常设为65536)。

图7-2显示了进程的地址空间、它的内存描述符以及线性区链表三者之间的关系。

注1：从理论上说，删除一个线性地址区间可能会失败，因为一个新的内存描述符可能没有空闲的内存可用。

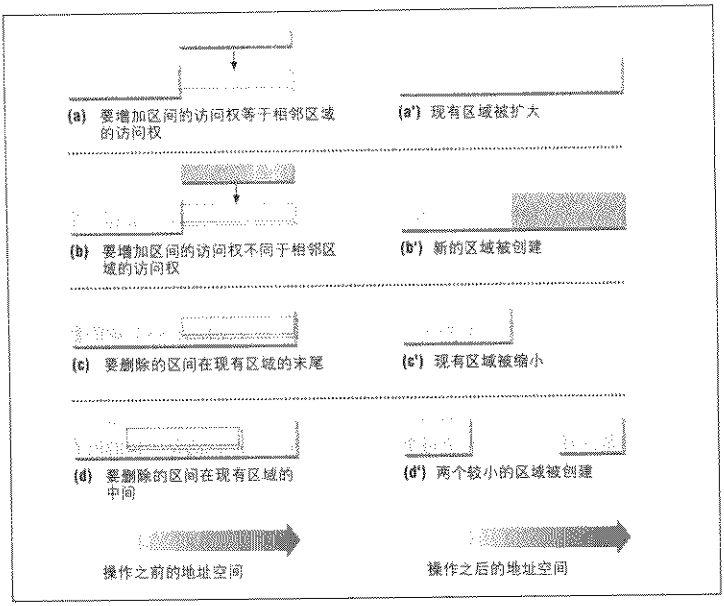


图7-1 增加或删除一个线性地址区间

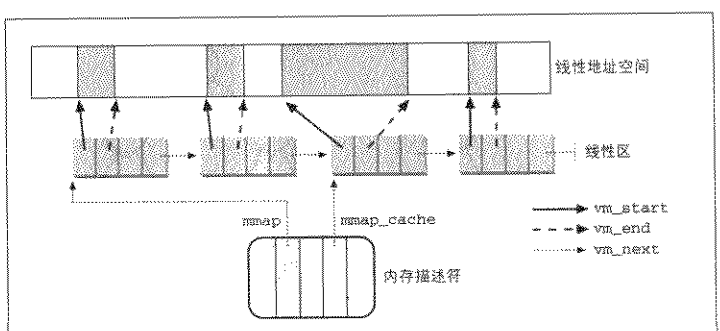


图7-2 与进程地址空间相关的描述符

内核频繁执行的一个操作就是查找指定线性地址所在的线性区。由于链表是经过排序的，因此，只要在指定线性地址之后找到一个线性区，搜索就可以结束。

然而，仅当进程有非常少的线性区时使用这种链表才是很方便的，比如说只有一二十个线性区。在链表中查找元素、插入元素、删除元素涉及许多操作，这些操作所花费的时间与链表的长度成线性比例。

尽管多数的Linux进程使用非常少的线性区，但是诸如面向对象的数据库那样过于庞大的大型应用程序可以有成百上千的线性区。在这种情况下，线性区链表的管理变得非常低效，因此，与内存相关的系统调用的性能就降低到令人无法忍受的程度。

当进程有大量的线性区时，Linux把它们的描述符存放在称为AVL树的数据结构中，AVL树是由Adelson-Velskii和Landis于1962年发明的。

在一棵AVL树中，每个元素（或者节点）通常有两个孩子：左孩子和右孩子。AVL树中的元素是这样排序的：对于每个结点N，以N为根结点的左子树的所有元素都小于N，相反，以N为根结点的右子树的所有元素都大于N［参见图7-3(a)；结点的关键字被写入到节点中］。

AVL树的每个结点N都有一个平衡因子（balanciing factor），它表明了这个结点下面分支的平衡程度。平衡因子是以节点N为根的左子树的深度减去它的右子树的深度。平衡AVL树每个节点的平衡因子只可能是-1，0或1［参见图7-3(a)：节点的平衡因子被写到节点的左边］。

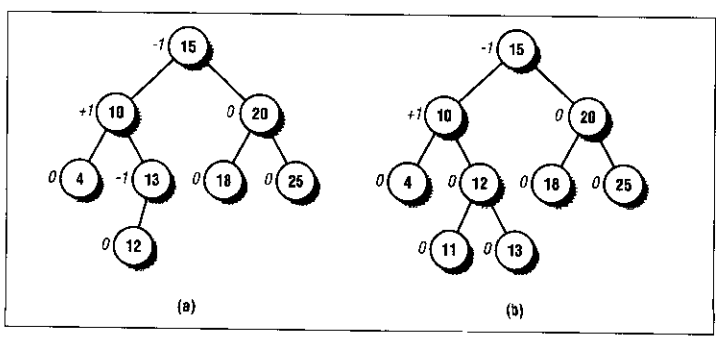


图7-3 AVL树的例子

在AVL树中查找一个元素是非常高效的，因为它需要的操作时间与树的大小的对数（以2为底）成线性比例。换句话说，线性区的数目增加一倍，查找的次数只增加一次。

在AVL树中插入或删除一个元素也是非常高效的，这是因为为了找到要插入元素的位置或者要删除元素的位置，所采用的算法能够快速遍历整个树。然而，这样的操作可能导致AVL树不平衡。例如，假设必须把一个关键字为11的元素插入到图7-3(a)所示的AVL树中，它的正确位置是关键字为12的节点的左孩子，但是，这个元素一旦被插入，关键字为13的节点的平衡因子就变为-2。为了重新平衡AVL树，在以关键字为13的节点为根的子树上，算法执行一次旋转操作，因此产生新的AVL树如图7-3 (b)所示。这看起来有点复杂，但是在一棵AVL树上插入或删除一个元素仅需要很少的操作——操作的次数与树的大小的对数成线性比例。

当然，AVL树也有它自己的缺点。它的处理函数要比线性链表的处理函数复杂得多。当元素的数目很少时，把它们放进线性链表中执行的效率远远高于放进AVL树中。

因此，为了存放一个进程的线性区，Linux通常利用内存描述符的mmap域所指向的链表。只有当进程的线性区数变得大于AVL_MIN_MAP_COUNT时（通常只有32个元素）才开始使用AVL树。因此，进程的内存描述符包含另外一个称之为mmap_avl的域指向这棵AVL树，在内核决定建立AVL树以前这个域的值为0。一旦AVL树被创建来管理进程的线性区，Linux就既维持链表的更新，也维持AVL树的更新。这两个数据结构含有指向同一个线性区描述符的指针。当插入和删除一个线性区描述符时，内核通过AVL树查找前一个和下一个元素，并利用所找到的元素快速更新链表而无需扫描链表。

AVL树每个节点的左孩子和右孩子的地址分别存放在vm_area_struct描述符的vm_avl_left和vm_avl_right域中。这个描述符还包含vm_avl_height域，用来存放以这个线性区描述符本身为根的子树的高度。这棵树根据vm_end域的值进行排序。

avl_rebalance()函数所接收的一个参数就是线性区AVL树的一个路径。如果必要，从这个路径的某一个节点开始，适当地旋转一个子树使它分叉以恢复平衡。该函数由avl_insert()和avl_remove()函数调用，而这两个函数分别在一棵树上

插入和删除一个线性区描述符。Linux 也使用 avl_insert_neighbours() 函数向 AVL 树插入一个元素，并返回新元素的左边和右边最近的节点的地址。

线性区存取权限

在讲述下一部分以前，先阐明页与线性区之间的关系。正如第二章中所提到的，我们使用“页”这个术语既表示一组线性地址又表示这组地址中所存放的数据。尤其是，我们把介于 0 到 4095 之间的线性地址区间称为第 0 页，介于 4096 到 8191 之间的线性地址区间称为第 1 页，依此类推。因此每个线性区都由一组号码连续的页所构成。

在前几章我们已经讨论了与页相关的两种标志：

- 在每个页表的表项中存放的几个标志，如：Read/Write、Present 或 User/Supervisor（参见第二章中的“常规分页”一节）。
- 存放在每个页描述符 flags 中的一组标志（参见第六章中的“页框管理”一节）。

第一种标志用于 Intel 80x86 硬件检查所请求的寻址类型是否可以被执行；第二种标志由 Linux 用于许多不同的目的（见表 6-1）。

现在介绍第三种标志，即与线性区的页相关的那些标志。它们存放在 vm_area_struct 描述符的 vm_flags 域中（见表 7-2）。一些标志给内核提供有关这个线性区全部页的信息，例如它们包含有什么内容，进程访问每个页的权限是什么。另外的标志描述线性区自身，例如它应该如何增长。

表 7-2 线性区标志

标志名	描述
VM_DENYWRITE	这个区映射一个打开后不能用来写的文件
VM_EXEC	页可以被执行
VM_EXECUTABLE	页含有可执行代码
VM_GROWSDOWN	这个区可以向低地址扩展
VM_GROWSUP	这个区可以向高地址扩展
VM_IO	这个区映射一个设备的 I/O 地址空间

表7-2 线性区标志（续）

标志名	描述
VM_LOCKED	页被锁住不能被交换出去
VM_MAYEXEC	VM_EXEC标志可以被设置
VM_MAYREAD	VM_READ标志可以被设置
VM_MAYSHARE	VM_SHARE标志可以被设置
VM_MAYWRITE	VM_WRITE标志可以被设置
VM_READ	页是可读的
VM_SHARED	页可以被多个进程共享
VM_SHM	页用于IPC共享内存
VM_WRITE	页是可写的

线性区描述符所包含的页存取权限可以任意组合。例如，存在这样一种可能性，允许一个线性区的页被执行但是不能被读取。为了有效地实现这种保护方案，与线性区的页相关的存取权限（读、写及执行）必须被复制到相应页表的所有表项中，以便由分页单元直接执行检查。换句话说，页存取权限表示什么样类型的存取应该产生一个“缺页”异常。稍后我们会看到，Linux委派缺页处理程序查找导致缺页的原因，因为缺页处理程序实现了许多页处理策略。

页表标志的初值（正如我们看到的，同一线性区所有页标志的初值必须一样）存放在vm_area_struct描述符的vm_page_prot域中。当增加一个页时，内核根据vm_page_prot域的值设置相应页表表项中的标志。

然而，并不能把线性区的存取权限直接转换成页保护位，这是因为：

- 在某些情况下，即使由相应线性区描述符的vm_flags域所指定的某个页的存取权限允许对该页进行访问，但是，对该页的访问还是应当产生一个“缺页”异常。例如，内核可能决定把属于两个不同进程的两个完全一样的可写的私有页（它的VM_SHARE标志被清除）存入同一个页框中。在这种情况下，无论哪一个进程试图改动这个页都应当产生一个异常（参见本章后面的“写时复制”一节）。
- Intel 80x86处理器的页表仅有两个保护位，称之为Read/Write和User/

Supervisor标志。此外，一个线性区所包含的任何一个页的User/Supervisor标志必须总置为1，因为用户态进程必须总能够访问其中的页。

为了克服Intel微处理器的硬件限制，Linux采取以下规则：

- 读存取权限总是隐含着执行存取权限。
- 写存取权限总是隐含着读存取权限。

此外，为了做到在写时复制技术中适当地推迟页框的分配（参见本章后面内容），只要相应的页不是由多个进程所共享，那么，这种页框都是写保护的。因此，由读、写、执行和共享存取权限所产生的十六种可能的组合被精简为以下三种：

- 如果页具有写和共享两种存取权限，那么，Read/Write位被设置为1。
- 如果页具有读或执行存取权限，但是既没有写也没有共享存取权限，那么，Read/Write位被清除为0。
- 如果页没有任何存取权限，那么，Present位被清除为0，以便每次访问都产生一个缺页异常。然而，为了把这种情况与真正的页框不存在的情况相区分，Linux还把Page size位置为1（注2）。

精简的保护位所对应的每种组合存取权限存放在protection_map数组中。

线性区的处理

对控制内存处理所用的数据结构和状态信息有了基本理解以后，我们来看一组对线性区描述符进行操作的低层函数。这些函数应当被看作简化了do_map()和ddo_unmap()实现的辅助函数。这两个函数将在本章后面的“分配线性地址区间”和“释放线性地址区间”两节中进行描述，它们分别扩大或者缩小进程的地址空间。这两个函数所处的层次比我们在这里所考虑函数的层次要高一些，它们并不接受线性区描述符作为参数，而是使用一个线性地址区间的起始地址、长度和存取权限作为参数。

注2：你可能认为Page size这种用法并不正当，因为这个位本来表示实际页的大小。但是，Linux可以侥幸逃脱这种骗局，因为Intel芯片在页目录项中检查Page size位，而不是在页表的表项中检查。

寻找给定地址的最近线性区

find_vma()函数有两个参数：进程内存描述符的地址mm和线性地址addr。它确定线性区的vm_end域大于addr的第一个线性区的位置，并返回这个线性区描述符的地址；如果没有这样的线性区存在，就返回一个NULL指针。注意由find_vma()函数所选择的线性区并不一定要包含addr。

每个内存描述符包含一个mmap_cache域，这个域保存进程最后一次所引用线性区的描述符地址。引进这个附加的域是为了减少查找一个给定线性地址所在线性区而花费的时间。程序中所引用地址的局部性使下面的情况出现的可能性很大：如果最后一个所检查的线性地址属于某一给定的线性区，那么，下一个要检查的线性地址也属于这一个线性区。

因此，该函数一开始就检查由mmap_cache所指定的线性区是否包含addr。如果是，就返回这个线性区描述符的指针：

```
vma = mm->mmap_cache;
if (vma && vma->vm_end > addr && vma->vm_start <= addr)
    return vma;
```

否则，必须扫描进程的线性区。如果进程没有使用AVL树，函数只简单地扫描链表：

```
if (!mm->mmap_avl) {
    vma = mm->mmap;
    while (vma && vma->vm_end <= addr)
        vma = vma->vm_next;
    if (vma)
        mm->mmap_cache = vma;
    return vma;
}
```

否则，函数在AVL树中查找线性区：

```
tree = mm->mmap_avl;
vma = NULL;
for (;;) {
    if (tree == NULL)
        break;
    if (tree->vm_end > addr) {
        vma = tree;
        if (tree->vm_start <= addr)
```

```
            break;
        tree = tree->vm_avl_left;
    } else
        tree = tree->vm_avl_right;
}
if (vma)
    mm->mmap_cache = vma;
return vma;
```

内核也使用find_vma_prev()函数，其参数之一是一个给定的线性地址，该函数返回两个值：位于该参数值之前的线性区的描述符地址（译注2）和位于该参数值之后的线性区的描述符地址。

寻找一个与给定的地址区间相重叠的线性区

find_vma_intersection()函数查找与给定的线性地址区间相重叠的第一个线性区。mm参数指向进程的内存描述符，而线性地址start_addr和end_addr指定这个区间。

```
vma = find_vma(mm,start_addr);
if (vma && end_addr <= vma->vm_start)
    vma = NULL;
return vma;
```

如果没有这样的线性区存在，函数就返回一个空指针NULL。准确地说，如果find_vma()函数返回一个有效的地址，但是所找到的线性区是从这个线性地址区间的末尾开始的，vma的值就被置为NULL。

寻找一个空闲的地址区间

get_unmapped_area()函数搜查进程的地址空间以找到一个可以使用的线性地址区间。len参数指定区间的长度，而addr参数可以指定从哪个地址开始进行查找。如果查找成功，函数返回这个新的区间的起始地址；否则返回0。

```
if (len > PAGE_OFFSET)
    return 0;
```

译注2：通过函数的另一个参数，即指向线性区描述符地址的指针传递出去。

```
if (!addr)
    addr = PAGE_OFFSET / 3;
addr = (addr + 0xfff) & 0xfffff000;
for (vmm = find_vma(current->mm, addr); ; vmm = vmm->vm_next) {
    if (addr + len > PAGE_OFFSET)
        return 0;
    if (!vmm || addr + len <= vmm->vm_start)
        return addr;
    addr = vmm->vm_end;
}
```

函数首先检查区间的长度是否小于用户态下线性地址区间的限长(通常为3GB)。如果addr为空，就从用户态线性地址空间的三分之一处开始查找。为了安全起见，函数把addr的值调整为4KB的倍数。然后，从addr开始，函数随着addr的增加反复调用find_vma()函数以找到所需要的空闲区间。在这个查找过程中，可能发生下面的情况：

- 如果所请求的区间大于正待扫描的部分线性地址空间(addr + len > PAGE_OFFSET)：由于没有足够的线性地址空间来满足这个请求，于是返回0。
- 刚刚扫描过的线性区后面的空闲区没有足够的大小(vmm != NULL && vmm->vm_start < addr + len)：继续考虑下一个线性区。
- 如果以上两种情况都没有发生，则找到一个足够大的空闲区：返回addr。

向内存描述符链表中插入一个线性区

insert_vm_struct()函数向内存描述符链表中插入一个vm_area_struct结构，如果必要，插入到AVL树中。这个函数使用两个参数：mm指定进程内存描述符的地址，vmp指定要插入的vm_area_struct描述符的地址：

```
if (!mm->mmap_avl) {
    pprev = &mm->mmap;
    while (*pprev && (*pprev)->vm_start <= vmp->vm_start)
        pprev = &(*pprev)->vm_next;
} else {
    struct vm_area_struct *prev, *next;
    avl_insert_neighbours(vmp, &mm->mmap_avl, &prev, &next);
    pprev = (prev ? &prev->vm_next : &mm->mmap);
}
```

```
vmp->vm_next = *pprev;
*pprev = vmp;
```

如果进程使用AVL树，则调用avl_insert_neighbours()函数，在适当的位置上插入这个线性区描述符。否则，insert_vm_struct()函数使用pprev局部变量向前扫描整个链表，直到找到应该位于vmp之前的描述符为止。在查找的最后，pprev指针指向在链表中应该位于vmp之前的线性区描述符的vm_next域，因此*pprev域产生应该位于vmp之后的线性区描述符的地址。这样就把一个线性区描述符插入到链表之中。

```
mm->map_count++;
if (mm->map_count >= AVL_MIN_MAP_COUNT && !mm->mmap_avl)
    build_mmap_avl(mm);
```

然后把进程内存描述符的map_count域增加1。此外，如果进程到目前为止还没有使用AVL树，但是线性区的数目已经变得大于或者等于AVL_MIN_MAP_COUNT，则调用build_mmap_avl()函数：

```
void build_mmap_avl(struct mm_struct * mm)
{
    struct vm_area_struct * vma;
    mm->mmap_avl = NULL;
    for (vma = mm->mmap; vma; vma = vma->vm_next)
        avl_insert(vma, &mm->mmap_avl);
}
```

从现在开始，进程开始使用AVL树。

如果这个线性区含有一个内存映射文件，该函数将执行另外的任务，这些任务将在第十六章中讲述。

不存在一个明确的函数用于从内存描述符链表中删除一个线性区（参见后面的“释放线性地址区间”一节）。

合并相邻的区

merge_segments()函数尽力把在一个给定的线性地址区间内的线性区合并在一起。如图7-1所示，仅当相邻的区有相同的存取权限时才能做到这一点。merge_segments()的参数是内存描述符指针mm和确定这个区间的两个线性地址start_addr

与end_addr。该函数找到在start_addr之前结束的最后一个线性区，并把它的描述符的地址放在prev局部变量中。然后重复执行下面的操作：

- 把prev->vm_next变量的值（即从start_addr之后开始的第一个线性区描述符的地址）装入局部变量mpnt。如果不存在这样的区，就没有合并的可能。
- 只要prev->vm_start小于end_addr，就循环遍历这个链表。检查有没有可能把prev与mpnt所指向的两个线性区合并在一起。下面是可能的情况：
  - 这两个线性区是连续的：prev->vm_end = mpnt->vm_start。
  - 这两个线性区有相同标志：prev->vm_flags = mpnt->vm_flags。
  - 当这两个线性区映射文件或者由多个进程所共享时，它们应当满足在后面章节所讨论的其他请求。

  如果合并是可能的，从链表中删除这个线性区描述符，如果必要，从AVL树中进行删除。
- 内存描述符的map_count域减1，设置prev变量，以便让prev指向合并的线性区描述符，并重新开始查找。

函数结束时把内存描述符的mmap_cache域设为NULL，这是因为mmap_cache现在指向一个并不存在的线性区。

分配线性地址区间

现在让我们讨论怎样分配一个新的线性地址区间。为了做到这点，do_mmap()函数为当前进程创建并初始化一个新的线性区。不过，分配成功之后，可以把这个新的线性区与进程已有的其它线性区进行合并。

do_mmap()函数使用下面的参数：

file和off

如果新的线性区将把一个文件映射到内存，则使用文件描述符指针file和文件偏移量off。这个主题将在第十五章进行讨论。在这一节中，我们假定不需要内存映射，并且file和off都为空。

addr

这个线性地址指定从何处开始查找一个空闲的区间（参见前面关于get_unmapped_area()函数的描述）。

len

线性地址区间的长度。

prot

这个参数指定这个线性区所包含页的存取权限。可能的标志有PROT_READ、PROT_WRITE、PROT_EXEC和PROT_NONE。前三个标志与标志VM_READ、VM_WRITE及VM_EXEC的意义一样。PROT_NONE表示进程没有以上三个存取权限中的任意一个。

flag

这个参数指定线性区的其它标志：

MAP_GROWSDOWN，MAP_LOCKED，MAP_DENYWRITE，和MAP_EXECUTABLE

它们的含义与表7-2中所列出标志的含义相同。

MAP_SHARED和MAP_PRIVATE

前一个标志指定线性区中的页可以被许多进程共享；后一个标志作用相反。这两个标志都涉及vm_area_struct中的VM_SHARED标志。

MAP_ANONYMOUS

没有文件与这个线性区相关联（参见第十五章）。

MAP_FIXED

这个区间的起始地址必须是由参数addr所指定的。

MAP_NORESERVE

函数不必预先检查空闲页框的数目。

do_mmap()函数首先检查参数的值是否正确，所提的请求是否能被满足。尤其是要检查以下不能满足请求的条件：

- 线性地址区间所包含的地址大于PAGE_OFFSET。
- 进程已经映射过多的线性区：进程的内存描述符mm的map_count域的值超过了MAX_MAP_COUNT的值。

- file参数等于NULL，并且flag参数指定新线性地址区间的页必须被共享。
- flag参数指定新线性地址区间的页必须被锁在内存，且进程加锁页的总数超过了保存在进程描述符的rlim[RLIMIT_MEMLOCK].rlim_cur域中的上限值。

如果发生上面情况中的任何一个，do_mmap()函数终止并返回一个负值。如果线性地址区间的长度为0，函数不执行任何操作就返回。

下面的步骤用于获得一个线性地址区间：如果MAP_FIXED标志被设置，则对addr值执行适当的边界检查；然后调用get_unmapped_area()函数获得addr。

```
if (flags & MAP_FIXED) {
    if (addr & 0xfffff000)
        return -EINVAL;
} else {
    addr = get_unmapped_area(addr, len);
    if (!addr)
        return -ENOMEM;
}
```

现在必须为新的区分配一个vm_area_struct描述符。这一步通过调用kmem_cache_alloc() slab分配器函数来实现。

```
vma = kmem_cache_alloc(vm_area_cachep, SLAB_KERNEL);
if (!vma)
    return -ENOMEM;
```

然后初始化这个线性区描述符。请注意vm_flags域的值是由参数prot、flags(通过vm_flags()结合在一起)与内存描述符的def_flags域共同决定的。def_flags域允许内核定义一组标志，而这组标志应该对进程的每个线性区进行设置(注3)。

```
vma->vm_mm = current->mm;
vma->vm_start = addr;
vma->vm_end = addr + len;
vma->vm_flags = vm_flags(prot,flags) | current->mm->def_flags;
vma->vm_flags |= VM_MAYREAD | VM_MAYWRITE | VM_MAYEXEC;
vma->vm_page_prot = protection_map[vma->vm_flags & 0x0f];
```

注3：实际上，这个域只能由mlockall()系统调用进行修改，而mlockall()可以用来设置VM_LOCKED标志，由此而锁定RAM中调用进程未来所有的页。

然后，do_mmap()函数检查以下这些错误条件是否存在：

- 进程在它的地址空间已经有一个线性区与从addr到addr+len的线性地址区间相重叠，而且do_munmap()函数不能释放该重叠区。
- 以页为单位的进程地址空间的大小超过了保存在进程描述符的rlim[RLIMIT_AS].rlim_cur域中的上限值。
- flags参数中没有设置MAP_NORESERVE标志，新的线性区必须含有私有的可写页，空闲页框的数目小于线性地址区间的大小（以页为单位）。这最后一步检查由vm_enough_memory()函数执行。

如果发生上面情况中的任何一个，do_mmap()函数释放所获得的vm_area_struct描述符，函数终止并返回-ENOMEM值。

一旦执行完所有的检查，do_mmap()增加current地址空间的大小（存放在内存描述符的total_vm域中）。然后调用insert_vm_struct()把新的区插入到current的线性区链表中（必要的话，通过AVL树进行插入），再调用merge_segments()函数检查线性区是否可以被合并。由于合并可能破坏掉新的线性区，因此把vm_flags和vm_start的值（以后可能用到）保存在flags和addr局部变量中。

```
current->mm->total_vm += len >> PAGE_SHIFT;
flags = vma->vm_flags;
addr = vma->vm_start;
insert_vm_struct(current->mm, vma);
merge_segments(current->mm, vma->vm_start, vma->vm_end);
```

仅当设置了MAP_LOCKED标志时才执行最后一步。首先，新线性区的页数被加到内存描述符的locked_vm域上。然后，调用函数make_pages_present()来连续地分配新线性区的所有页，并把它们锁定在RAM中。make_pages_present()函数的关键代码是：

```
vma = find_vma(current->mm, addr);
write = (vma->vm_flags & VM_WRITE) != 0;
while (addr < addr + len) {
    handle_mm_fault(current, vma, addr, write);
    addr += PAGE_SIZE;
}
```

我们将在“处理在地址空间内的一个错误地址”一节中看到，handle_mm_fault()函数分配一个页，并根据线性区描述符的vm_flags域设置其页表的表项。

最后，do_munmap()函数结束，并返回新的线性区的线性地址。

释放线性地址区间

do_munmap()函数从当前进程的地址空间中删除一个线性地址区间。参数为区间的起始地址addr和它的长度len。要删除的区间并不总是对应一个线性区：它或许是一个线性区的一部分，或许跨越两个或多个线性区。

该函数经过两个主要的阶段。第一阶段，扫描进程所拥有的线性区链表，并删除与指定线性地址区间相重叠的所有区。第二阶段，更新进程的页表，并重新调整线性区链表。

第一阶段：扫描线性区

首先对输入参数进行初步检查：如果这个线性地址区间所含的地址大于PAGE_OFFSET，如果addr不是4096的倍数，或者如果线性地址区间的长度为0，则函数返回一个负的错误代码。

下一步，函数确定与要删除的线性地址区间相重叠的第一个线性区的位置：

```
mpnt = find_vma_prev(current->mm, addr, &prev);
if (!mpnt || mpnt->vm_start >= addr + len)
    return 0;
```

如果这个线性地址区间位于一个线性区之内，删除这个区间将把这个区分为两个更小的部分。在这种情况下，do_munmap()函数检查是否允许当前进程获得一个额外的线性区：

```
if ((mpnt->vm_start < addr && mpnt->vm_end > addr + len) &&
    current->mm->map_count > MAX_MAP_COUNT)
    return -ENOMEM;
```

然后，do_munmap()尝试得到一个新的vm_area_struct描述符。这一步也许是没有必要的，但函数一定要这样做以便分配失败时可以立即终止。这种谨慎的方法简化了代码，因为它允许出现错误时很容易退出：

```
extra = kmem_cache_alloc(vm_area_cachep, SLAB_KERNEL);
if (!extra)
    return -ENOMEM;
```

现在函数建立一个链表，该链表含有与这个线性地址区间相重叠的所有线性区的描述符。这个链表是通过设置线性区描述符（临时的）的vm_next域创建的，因此vm_next指向链表中的前一项，vm_next域由此就起倒序链接的作用。由于每个区都加到这个倒序链表上，因而有一个局部变量free指向最后插入的元素。插入到这个链表中的区也要从进程的线性区链表中删除，如果必要，也从AVL树中删除。

```
npp = (prev ? &prev->vm_next : &current->mm->mmap);
free = NULL;
for ( ; mpnt && mpnt->vm_start < addr + len; mpnt = *npp) {
    *npp = mpnt->vm_next;
    mpnt->vm_next = free;
    free = mpnt;
    if (current->mm->mmap_avl)
        avl_remove(mpnt, &current->mm->mmap_avl);
}
```

第二阶段：更新页表

从free指针所指向的线性区描述符开始，用一个while循环扫描第一个阶段所创建的线性区链表。

在每一次循环中，mpnt局部变量指向链表中的一个线性区描述符。current->mm内存描述符的map_count域被减1（因为在第一阶段中这个区已经从进程所拥有的线性区链表中删除了），并检查（使用两个问号条件表达式）mpnt区是否必须被取消或者只使它变小。

```
current->mm->map_count--;
st = addr < mpnt->vm_start ? mpnt->vm_start : addr;
end = addr + len;
end = end > mpnt->vm_end ? mpnt->vm_end : end;
size = end - st;
```

st和end局部变量划定应当被删除的mpnt线性区内的线性地址区间；size局部变量指定这个区间的长度。

下一步，do_munmap()释放为位于st到end区间内的页所分配的页框：

```
zap_page_range(current->mm, st, size);
flush_tlb_range(current->mm, st, end);
```

zap_page_range()函数释放从st到end区间所包含的页框，并更新相应的页表表项。该函数以嵌套的方式调用zap_pmd_range()和zap_pte_range()函数来扫描页表，后一个函数使用pte_clear宏清除页表的表项，再使用free_pte()函数释放相应的页框。

然后调用flush_tlb_range()函数，使st到end区间所对应的TLB表项无效。在Intel 80x86体系结构中，该函数简单地调用__flush_tlb()函数，以使所有的TLB表项都无效。

do_munmap()的每次循环的最后一步是检查变小的mpnt线性区是否必须插入到当前进程的线性区链表中：

```
extra = unmap_fixup(mpnt, st, size, extra);
```

unmap_fixup()函数考虑了四种可能的情况：

- 这个线性区完全被取消。返回存放在extra局部变量中的地址，因此调用kmem_cache_free()发出释放extra线性区描述符的信号。
- 只有这个线性区的低地址部分被删除，也就是说：

  ```
  (mpnt->vm_start < st) && (mpnt->vm_end == end)
  ```

  在这种情况下，更新mnpt的vm_end域，调用insert_vm_struct()函数把变小的线性区插入到进程的线性区链表中，然后返回存放在extra中的地址。
- 只有线性区的高地址部分被删除，也就是说：

  ```
  (mpnt->vm_start == st) && (mpnt->vm_end > end)
  ```

  在这种情况下，更新mnpt的vm_start域，调用insert_vm_struct()函数把变小的线性区插入到进程的线性区链表中，然后返回存放在extra中的地址。
- 要删除的线性地址区间在这个线性区的中间，也就是说：

  ```
  (mpnt->vm_start < st) && (mpnt->vm_end > end)
  ```

  更新mnpt的vm_start与vm_end域，更新extra（以前分配的extra线性区描述符），以使它们分别指向两个线性地址区间：从mpnt->vm_start到st，

从 end 到 mpnt->vm_end。然后两次调用 insert_vm_struct()函数把这两个线性区插入到进程的线性区链表中（如果必要，插入到 AVL 树），并返回 NULL，这样就保留了以前所分配的 extra 线性区描述符。

通过对第二阶段一次循环过程的描述，也就完成了对整个第二阶段的描述。

处理完第二阶段所创建链表中的所有线性区描述符之后，do_munmap()检查是否还使用了另外的 extra 内存描述符。如果 extra 为 NULL，则描述符已经被使用过；否则，do_munmap()调用 kmem_cache_free()释放这个额外的内存描述符。最后，如果进程地址空间已经被修改过，则 do_munmap()把这个内存描述符的 mmap_cache 域置为 NULL，并返回 0。

缺页异常处理程序

如前所述，Linux 的“缺页”异常处理程序必须区分以下两种情况：由编程错误所引起的异常，及由引用进程地址空间的页但还尚未分配物理页框所引起的异常。

线性区描述符可以让缺页异常处理程序非常有效地完成它的工作。do_page_fault()函数是“缺页”中断服务程序，它把引起缺页的线性地址和当前进程的线性区相比较；从而能够根据图 7-4 所显示的方案选择适当的方法处理这个异常。

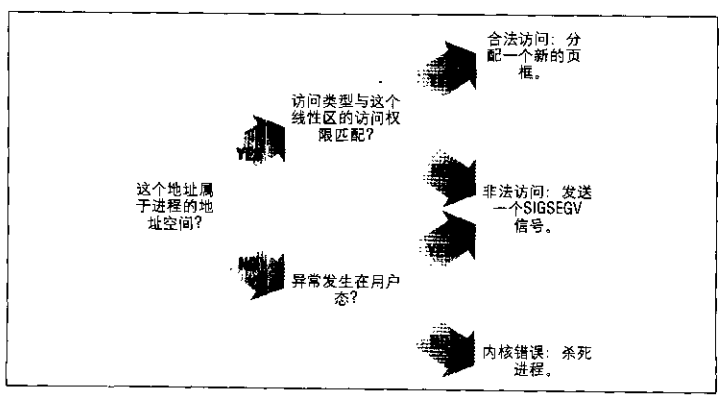


图 7-4 缺页异常处理程序的总体方案

实际上，情况更复杂一些，因为缺页处理程序必须处理多种分得更细的特殊情况，它们不宜在总体方案中列出来，还必须区分许多种合理的访问。处理程序的详细流程图如图7-5所示。

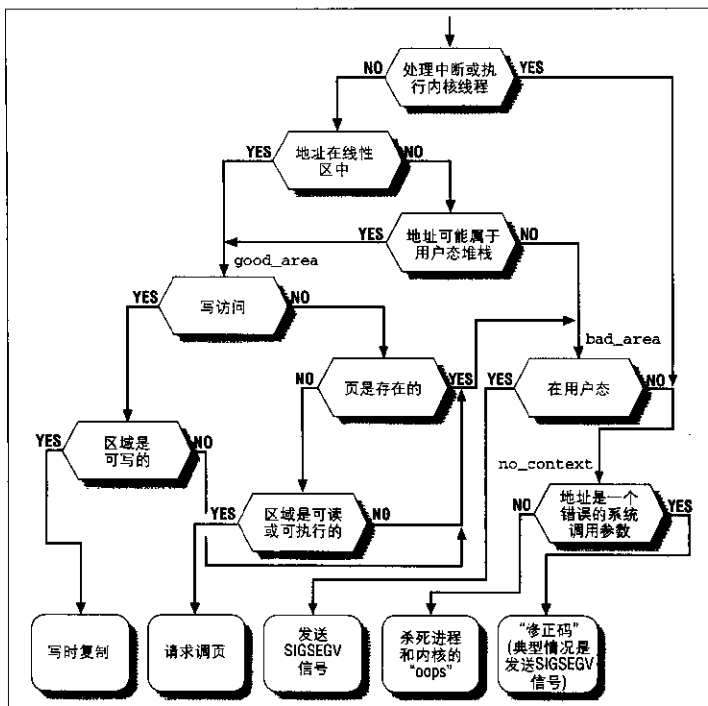


图7-5 缺页处理程序流程图

标识符good_area、bad_area和no_context是出现在do_page_fault()中的标号，它们有助于你理清流程图中的块与代码中特定行之间的关系。

do_page_fault()函数接收以下输入参数：

- 一个pt_regs结构的地址regs，该结构包含当异常发生时的微处理器寄存器的值。
- 一个3位的error_code，当异常发生时由控制单元压入栈中（参见第四章中的“中断和异常的硬件处理”一节）。这些位有以下含义：
  - 如果第0位被清0，则异常由访问一个不存在的页所引起（页表表项中的Present标志被清0）；否则，如果第0位被设置，则异常由无效的访问权所引起。
  - 如果第1位被清0，则异常由读访问或者执行访问所引起；如果被设置，则异常由写访问所引起。
  - 如果第2位被清0，则异常发生在处理器处于内核态时；否则，异常发生在处理器处于用户态时。

do_page_fault()的第一步操作是读取引起缺页的线性地址。当异常发生时，CPU控制单元把这个值存放在cr2控制寄存器中：

```
asm("movl %%cr2,%0":"=r" (address));
tsk = current;
mm = tsk->mm;
```

这个线性地址被保存在address局部变量中。该函数还把current的进程描述符指针和内存描述符指针分别保存在tsk和mm局部变量中。

正如图7-5中的顶部所示，do_page_fault()首先检查异常发生时CPU是否正在处理中断或者执行内核线程：

```
if (in_interrupt() || mm == &init_mm)
    goto no_context;
```

在这两种情况下，do_page_fault()并不打算把这个线性地址与current的线性区进行比较，因为这样做毫无意义：中断处理程序和内核线程永远不使用低于PAGE_OFFSET的线性地址，因此也就永远不依赖线性区。

我们假定缺页没有发生在中断处理程序或者内核线程中。于是函数必须检查进程的线性区以决定这个错误的线性地址是否包含在进程的地址空间中：

```
vma = find_vma(mm, address);
```

```
if (!vma)
    goto bad_area;
if (vma->vm_start <= address)
goto good_area;
```

现在函数已经确定address不在任何一个线性区中。可是它还必须执行进一步的检查，由于这个错误地址可能是由push或pusha指令在进程的用户态栈上的操作所引起的。

我们稍微离题一点，解释一下栈是如何映射到线性区上的。每个向低地址扩展的栈所在的区，它的VM_GROWSDOWN标志被设置，这样当vm_start域的值可能被减小的时候，而vm_end域的值保持不变。这个线性区确定的范围涵盖用户态栈当前的大小，但不正好是这个大小。这种细微的差别主要基于以下原因：

- 线性区的大小是4KB的倍数（必须包含完整的页），而栈的大小却是任意的。
- 分给一个线性区的页框在这个线性区被删除前永远不被释放。特别地，一个栈所在的线性区的vm_start域的值只能减小，永远也不能增加。甚至进程执行一系列的pop指令时，这个线性区的大小仍然保持不变。

现在这一点就很清楚了：当填满分配给进程栈的最后一个页框后，这个进程可能引起一个“缺页”异常，即push指向这个线性区以外的一个地址（即指向一个不存在的页框）。注意，这种异常不是由程序错误引起的，它必须由缺页处理程序单独处理。

我们现在回到对do_page_fault()的描述，它检查上面所描述的情况：

```
if (!(vma->vm_flags & VM_GROWSDOWN))
    goto bad_area;
if (error_code & 4      /* 用户态 */
    && address + 32 < regs->esp)
    goto bad_area;
if (expand_stack(vma, address))
    goto bad_area;
goto good_area;
```

如果这个线性区的VM_GROWSDOWN标志被设置，并且异常发生在用户态，函数就要检查address是否小于regs->esp栈指针（它应该只小于一点点）。因为只有当内存访问结束以后，几个与栈相关的汇编语言指令（如pusha）才执行减esp寄存器的操作，所以允许进程有32字节的后备区间。如果这个地址足够高（在允许的范围

内），则代码调用expand_stack()函数检查是否允许进程既扩展它的栈也扩展它的地址空间。如果一切都可以，就把vma的vm_start域设为address，并返回0；否则，返回1。

注意：只要线性区的VM_GROWSDOWN标志被设置，但异常不是发生在用户态，上述代码就跳过后备区的检查。这些条件意味着内核正在访问用户态的栈，也意味着这段代码总是应当运行expand_stack()。

处理地址空间以外的错误地址

如果address不属于进程的地址空间，do_page_fault()函数继续执行bad_area标号处的语句。如果错误发生在用户态，则发送一个SIGSEGV信号给当前进程current并结束函数（参见第九章的“发送信号”一节）：

```
bad_area:
if (error_code & 4) {    /* 用户态 */
    tsk->tss.cr2 = address;
    tsk->tss.error_code = error_code;
    tsk->tss.trap_no = 14;
    force_sig(SIGSEGV, tsk);
    return;
}
```

然而，如果异常发生在内核态（error_code的第2位被清0），仍然有两种情况：

- 进行系统调用时，把某个线性地址作为参数传递给内核，从而引起异常。
- 异常是因一个真正的内核缺陷所引起的。

函数这样区分这两种情况：

```
no_context:
if (!fixup = search_exception_table(regs->eip)) != 0) {
    regs->eip = fixup;
    return;
}
```

在第一种情况中，代码跳到一段“修正代码”处，这段代码特地给当前进程发送SIGSEGV信号，或用一个适当的错误码终止系统调用处理程序（参见第八章中的“动态地址检查：修正代码”一节）。

在第二种情况中，函数把CPU寄存器和内核态堆栈的全部转储打印到控制台，并输出到一个系统消息缓冲区，然后调用函数do_exit()杀死当前进程（参见第十九章）。这就是所谓的按消息显示命名的“内核oops”错误。这些输出值可由内核程序高手用于推测引发此错误的条件，进而发现并纠正这个错误。

处理地址空间内的错误地址

如果addr地址属于进程的地址空间，do_page_fault()转到good_area标号处的语句执行：

```
good_area:
write = 0;
if (error_code & 2) {  /* 写访问 */
    if (!(vma->vm_flags & VM_WRITE))
        goto bad_area;
    write++;
} else                 /* 读访问 */
    if (error_code & 1 ||
        !(vma->vm_flags & (VM_READ | VM_EXEC)))
        goto bad_area;
```

如果异常由写访问引起，函数检查这个线性区是否可写。如果不可写，跳到bad_area代码处；如果可写，把write局部变量置为1。

如果异常由读或执行访问引起，函数检查这一页是否已经存在于RAM中。在存在的情况下，异常发生是由于进程试图访问用户态下的一个有特权的页框（页框的User/Supervisor标志被清除），因此函数跳到bad_area代码处（注4）。在不存在的情况下，函数还将检查这个线性区是否可读或可执行。

如果这个线性区的访问权限与引起异常的访问类型相匹配，则handle_mm_fault()函数被调用：

```
if (!handle_mm_fault(tsk, vma, address, write)) {
    tsk->tss.cr2 = address;
    tsk->tss.error_code = error_code;
    tsk->tss.trap_no = 14;
```

注4：　然而，这种情况从不会发生，因为内核不会把具有特权的页框赋给进程。

```
        force_sig(SIGBUS, tsk);
        if (!(error_code & 4)) /* 内核态 */
            goto no_context;
    }
```

如果handle_mm_fault()函数成功地给进程分配一个页框，则返回1；否则返回一个适当的错误码，以便do_page_fault()函数可以给进程发送SIGBUS信号。这个函数有4个参数：

tsk

指向异常发生时正在CPU上运行的进程的描述符

vma

指向引起异常的线性地址所在线性区的描述符

address

引起异常的线性地址

write_access

如果tsk试图向address写，则置为1，如果tsk试图读或执行address，则置为0

这个函数首先检查用来映射address的页中间目录和页表是否存在。即使address属于进程的地址空间，相应的页表可能还没有被分配，因此，在做别的事情之前首先执行分配页目录和页表的任务：

```
    pgd = pgd_offset(vma->vm_mm, address);
    pmd = pmd_alloc(pgd, address);
    if (!pmd)
        return -1;
    pte = pte_alloc(pmd, address);
    if (!pte)
    return -1;
```

pgd局部变量所指向的页全局目录目录项涉及到address。如果需要的话，调用pmd_alloc()函数分配一个新的页中间目录（注5）。然后，如果需要的话，调用pte_alloc()函数分配一个新的页表。如果这两步都成功，pte局部变量所指向的

注5：在Intel 80x86微处理器中，这种分配永远不会发生，因为页中间目录就包含在页全局目录中。

页表表项就是引用address的表项。然后调用handle_pte_fault()函数检查address地址所对应的页表表项:

```
return handle_pte_fault(tsk, vma, address, write_access, pte);
```

handle_pte_fault()函数决定怎样给进程分配一个新的页框:

- 如果被访问的页不存在，也就是说，这个页还没有被存放在任何一个页框中，那么，内核分配一个新的页框并适当地初始化。这种技术称为请求调页(demand paging)。
- 如果被访问的页存在但是被标为只读，也就是说，它已经被存放在一个页框中，那么，内核分配一个新的页框，并把旧页框的数据拷贝到新页框来初始化它的内容。这种技术称为写时复制（Copy On Write，COW)。

请求调页

术语请求调页指的是一种动态内存分配技术，它把页框的分配推迟到不能再推迟为止，也就是说，一直推迟到进程要访问的页不在RAM中时为止，由此引起一个缺页异常。

请求调页技术背后的动机是：进程开始运行的时候并不访问其地址空间中的全部地址。事实上，有一部分地址也许进程永远不使用。此外，程序的局部性原理（参见第二章中的“硬件高速缓存”一节）保证了在程序执行的每个阶段，真正使用的进程页只有一小部分，因此临时用不着的页所在的页框可以由其它进程来使用。因此，对于全局分配（一开始就给进程分配所需要的全部页框，直到程序结束才释放这些页框）来说，请求调页是首选的，因为它增加了系统中的空闲页框的平均数，从而更好地利用空闲内存。从另一个观点来看，在内存总数保持不变的情况下，请求调页从总体上能使系统有更大的吞吐量。

为这一切优点付出的代价是系统额外的开销：由请求调页所引发的每个“缺页”异常必须由内核处理，这将浪费CPU的时钟周期。幸运的是，局部性原理保证了一旦进程开始在一组页上运行，在接下来相当长的一段时间内它会一直停留在这些页上而不去访问其他的页，这样我们就可以认为“缺页”异常是一种稀有事件。

基于以下原因，被寻址的页可不在主存中：

- 进程永远也没有访问到这个页。内核能够识别这种情况，这是因为页表相应的表项被填充为0，也就是说，pte_none宏返回1。
- 进程已经访问过这个页，但是这个页的内容被临时保存在磁盘上。内核能够识别这种情况，这是因为页表相应表项没被填充为0（然而，由于页框不在RAM中，Present标志被清除）。

handle_pte_fault()函数通过检查与address相关的页表表项来区分这两种情况：

```
entry = *pte;
if (!pte_present(entry)) {
    if (pte_none(entry))
        return do_no_page(tsk, vma, address, write_access,
                          pte);
    return do_swap_page(tsk, vma, address, pte, entry,
                        write_access);
}
```

我们将在第十六章中的“页换入”一节检查页被保存到磁盘上的这种情况（do_swap_page()函数）。

在其它情况下，当页从未被访问时则调用do_no_page()函数。有两种方法装入所缺的页，这取决于这个页是否被映射到磁盘文件。该函数通过检查vma线性区描述符的nopage域来确定这一点，如果页被映射到一个文件，nopage域就指向一个把所缺的页从磁盘装入到RAM的函数。因此，可能的情况是：

- vma->vm_ops->nopage域不为NULL。在这种情况下，某个线性区映射一个磁盘文件，nopage域指向装入这个页的函数。这种情况将在第十五章的“内存映射”一节和第十八章的“IPC共享内存”一节进行完整的阐述。
- 或者vm_ops域为NULL、或者vma->vm_ops->nopage域为NULL。在这种情况下，线性区没有映射磁盘文件，也就是说，它是一个匿名映射（anonymous mapping）。因此，do_no_page()调用do_anonymous_page()函数获得一个新的页框：

```
if (!vma->vm_ops || !vma->vm_ops->nopage)
    return do_anonymous_page(tsk, vma, page_table,
                             write_access);
```

do_anonymous_page()函数分别处理写请求和读请求：

```
if (write_access) {
    page = __get_free_page(GFP_USER);
    memset((void *)(page), 0, PAGE_SIZE)
    entry = pte_mkwrite(pte_mkdirty(mk_pte(page,
                vma->vm_page_prot)));
    vma->vm_mm->rss++;
    tsk->min_flt++;
    set_pte(pte, entry);
    return 1;
}
```

当处理写访问时，该函数调用__get_free_page()，并利用memset宏把新页框填为0。然后该函数增加tsk的min_flt域以跟踪由进程引起的次级缺页（minor page fault，这些缺页只需要一个新页框）的数目，再增加进程的内存描述符vma->vm_mm的rss域以跟踪分配给进程的页框数目（注6）。然后页表相应的表项被设为页框的物理地址，并把这个页框标记为可写和脏两个标志。

相反，当处理读访问时，页的内容是无关紧要的，因为进程正在对它进行第一次寻址。给进程一个填充为0的页要比给它一个由其他进程填充了信息的旧页更为安全。Linux在请求调页方面做得更深入一些。没有必要立即给进程分配一个填充为0的新页框，由于我们也可以给它一个现有的称为零页（zero page）的页，这样可以进一步推迟页框的分配。零页在内核初始化期间被静态分配，并存放在empty_zero_page变量中（一个有1024个长整数的数组，并用0填充）。它存放在第六个页框中（从物理地址0x00005000开始），并且可以通过ZERO_PAGE宏来引用。

因此页表表项被设为零页的物理地址：

```
entry = pte_wrprotect(mk_pte(ZERO_PAGE, vma->vm_page_prot));
set_pte(pte, entry);
return 1;
```

由于这个页被标记为不可写，如果进程试图写这个页，则写时复制机制被激活。当

注6：Linux为每个进程记录次级缺页数。可以用这个信息及其他几个统计信息来调整系统。内存描述符的rss域所存放的值也由内核用来选择可以盗取页框的线性区（参见第十六章中的“释放页框”一节）。

且仅当在这个时候，进程才获得一个属于自己的页并对它进行写。这种机制将在下一节中进行描述。

写时复制

第一代Unix系统，实现了一种傻瓜式的进程创建：当发出fork()系统调用时，内核原样复制父进程的整个地址空间并把复制的那一份分配给子进程。这种行为是非常耗时的，因为它需要：

- 为子进程的页表分配页框
- 为子进程的页分配页框
- 初始化子进程的页表
- 把父进程的页复制到子进程相应的页中

这种创建一个地址空间的方法涉及许多内存访问，消耗许多CPU周期，并且完全破坏了高速缓存中的内容。在大多数情况下，这样做常常是毫无意义的，因为许多子进程通过装入一个新的程序开始它们的执行，这样就完全丢弃了所继承的地址空间（参见第十九章）。

现在的Unix内核（包括Linux），采用一种更为有效的方法，称之为写时复制（或COW）。这种思想相当简单：父进程和子进程共享页框而不是复制页框。然而，只要页框被共享，它们就不能被修改，无论父进程和子进程何时试图写一个共享的页框，就产生一个异常，这时内核就把这个页复制到一个新的页框中并标记为可写。原来的页框仍然是写保护的，当其他进程试图写入时，内核检查写进程是否是这个页框的唯一属主，如果是，它把这个页框标记为对这个进程是可写的。

页描述符的count域用于跟踪共享相应页框的进程数目。只要进程释放一个页框或者在它上面执行写时复制，它的count域就减小；只有当count变为NULL时，这个页框才被释放。

现在我们讲述Linux怎样实现写时复制（COW）。当handle_pte_fault()确定“缺页”异常是由请求写一个页框所引起的时（这个页框存在于内存中且是写保护的），它执行以下指令：

```
    if (pte_present(pte)) {
        entry = pte_mkyoung(entry);
        set_pte(pte, entry);
        flush_tlb_page(vma, address);
        if (write_access) {
            if (!pte_write(entry))
                return do_wp_page(tsk, vma, address, pte);
            entry = pte_mkdirty(entry);
            set_pte(pte, entry);
            flush_tlb_page(vma, address);
            }
        return 1;
    }
```

首先，调用pte_mkyoung()和set_pte()函数来设置引起异常的页所对应页表项的访问位。这个设置使页“年轻”并使它被交换到磁盘上的机会减少（参见第十六章）。如果异常是由违背写保护而引起的，handle_pte_fault()返回由do_wp_page()函数产生的值；否则，则已检测到某一错误情况（例如，用户态地址空间中的页，其User/Supervisor标志为0），且函数返回1。

do_wp_page()函数首先把page_table参数所引用的页表表项装入局部变量pte，然后再获得一个新页框：

```
    pte = *page_table;
    new_page = __get_free_page(GFP_USER);
```

由于页框的分配可能阻塞进程，因此，一旦获得页框，这个函数就在页表表项上执行下面的一致性检查：

- 当进程等待一个空闲的页框时，这个页是否已经被交换出去（pte和*page_table的值不相同）。
- 这个页是否已不在RAM中（页表表项中页的Present标志为0）。
- 页现在是否可写（页表项中页的Read/Write标志为1）。

如果这些情况中的任意一个发生，do_wp_page()释放以前所获得的页框，并返回1。

现在，函数更新次级缺页的数目，并把引起异常的页的页描述符指针保存到page_map局部变量中。

```
tsk->min_flt++;
page_map = mem_map + MAP_NR(old_page);
```

接下来，函数必须确定是否必须真的把这个页复制一份。如果仅有一个进程使用这个页，就无须应用写时复制技术，而且进程应该能够自由地写这个页。因此，这个页框被标记为可写，这样当试图写入的时候就不会再次引起“缺页”异常，以前分配的新的页框也被释放，函数结束并返回1。这种检查是通过读取页描述符的count域而进行的（注7）：

```
if (page_map->count != 1) {
    set_pte(page_table, pte_mkdirty(pte_mkwrite(pte)));
    flush_tlb_page(vma, address);
    if (new_page)
        free_page(new_page);
    return 1;
}
```

相反，如果这个页框由两个或多个进程所共享，函数把旧页框（old_page）的内容复制到新分配的页框（new_page）中：

```
if (old_page == ZERO_PAGE)
    memset((void *) new_page, 0, PAGE_SIZE);
else
    memcpy((void *) new_page, (void *) old_page, PAGE_SIZE);
set_pte(page_table, pte_mkwrite(pte_mkdirty(
        mk_pte(new_page, vma->vm_page_prot))));
flush_tlb_page(vma, address);
__free_page(page_map);
return 1;
```

如果旧页框是零页框，就使用memset宏把新的页框填充为0。否则，使用memcpy宏复制页框的内容。不要求一定要对零页做特殊的处理，但是特殊处理确实能够提高系统的性能，因为它使用很少的地址而保护了微处理器的硬件高速缓存。

然后，用新页框的物理地址更新页表的表项，并把新页框标记为可写和脏。最后，函数调用__free_page()减小对旧页框的引用计数。

注7：实际上，这个检查还稍微复杂一些，因为当这个页被插入到交换高速缓存中时，count域也被增加（参见第十六章中的“交换高速缓存”一节）。

创建和删除进程的地址空间

除了“进程的地址空间”这一节所提到的进程获得一个新的线性区的六种典型的情况之外，首先要指出的是fork()系统调用要求为子进程创建一个完整的新地址空间。相反，当进程结束时，内核撤消它的地址空间。这一节我们讨论Linux如何执行这两种操作。

创建进程的地址空间

我们在第三章的“clone()，fork()及vfork()系统调用”一节中已经提到，当创建一个新的进程时内核调用copy_mm()函数。该函数通过建立新进程的所有页表和内存描述符来关注进程地址空间的创建过程。

通常，每个进程有自己的地址空间，但是轻量级进程可以通过调用__clone()函数（设置了CLONE_VM标志）来创建。这些轻量级进程共享同一地址空间，也就是说，允许它们对同一组页进行寻址。

按照前面讲述的写时复制方法（COW），传统的进程继承父进程的地址空间，只要页是只读的，就依然共享它们。当其中的一个进程试图写入某一个页时，这个页就被复制一份。一段时间之后，所创建的进程通常获得与父进程不一样的完全属于自己的地址空间。另一方面，轻量级的进程使用父进程的地址空间，Linux的实现方法很简单，即不复制这种地址空间。创建轻量级的进程比创建普通进程相应要快得多，而且只要父进程和子进程谨慎地调整它们的访问，就可以认为页的共享是有益的。

如果通过__clone()系统调用已经创建了新进程，并且flag参数的CLONE_VM标志被设置，则copy_mm()函数给出父进程地址空间的复制品：

```
if (clone_flags & CLONE_VM) {
    mmget(current->mm);
    copy_segments(nr, tsk, NULL);
    SET_PAGE_DIR(tsk, current->mm->pgd);
    return 0;
}
```

copy_segments()函数为克隆的进程建立LDT，因为即使轻量级进程也必须在

GDT中有一个独立的LDT表项。SET_PAGE_DIR宏设置新进程的页全局目录，并且把页全局目录的地址存放在新的内存描述符的mm->pgd域中。

如果CLONE_VM标志没有被设置，copy_mm()函数必须创建一个新的地址空间（即使在进程请求一个地址之前，并没有在地址空间内分配内存）。这个函数分配一个新的内存描述符并把它的地址存放在新进程描述符的mm域中。然后把新进程描述符的许多域初始化为零，并且，与前一种情况一样，调用copy_segments()函数建立LDT描述符：

```
mm = mm_alloc();
if (!mm)
    return -ENOMEM;
tsk->mm = mm;
copy_segments(nr, tsk, mm);
```

下一步，copy_mm()调用new_page_tables()分配页全局目录。这个表最后的一些表项（对应于高于PAGE_OFFSET的线性地址）是从swapper进程的页全局目录中复制而来的，而其余部分则设置为0（尤其是，Present和Read/Write标志被清0）。最后，new_page_tables()把页全局目录的地址存放在新内存描述符的mm->pgd域中。然后调用dup_mmap()函数既复制父进程的线性区，也复制父进程的页表：

```
new_page_tables(tsk);
dup_mmap(mm);
return 0;
```

从current->mm->mmap指向的地方开始，dup_mmap()函数扫描父进程线性区链表。它复制遇到的每个vm_area_struct线性区描述符，并把复制品插入到子进程的线性区链表中。

在插入一个新的线性区描述符之后，如果需要的话，dup_mmap()立即调用copy_page_range()创建必要的页表来映射这个线性区所在的一组页框，并且初始化新页表的表项。尤其是，与私有的、可写的页（VM_SHARE标志关闭，VM_MAYWRITE标志打开）所对应的任一页框都被标记为对父子进程都是只读的，以便这种页框能被写时复制机制进行处理。最后，如果线性区的数目大于或等于AVL_MIN_MAP_COUNT，则调用build_mmap_avl()函数创建子进程的线性区AVL树。

删除进程的地址空间

当进程结束时，内核调用 exit_mm() 函数释放进程的地址空间。由于进程正进入 TASK_ZOMBIE 状态、函数把 *swapper* 进程的地址空间分配给这个进程:

```
flush_tlb_mm(mm);
tsk->mm = &init_mm;
tsk->swappable = 0;
SET_PAGE_DIR(tsk, swapper_pg_dir);
mm_release();
mmput(mm);
```

然后该函数调用mm_release()和mmput()释放进程的地址空间。第一个函数清除fs、gs段寄存器，并把进程的LDT恢复为default_ldt；第二个函数减小mm->count域的值并释放LDT、线性区描述符和mm指向的页表。最后，内存描述符本身也被释放。

堆的管理

每个Unix进程都拥有一个特殊的线性区，这个线性区就是所谓的堆（heap），堆用于满足进程的动态内存请求。内存描述符的start_brk与brk域分别限定了这个区的开始地址和结束地址。

进程可以使用下面的C语言库函数来请求和释放动态内存:

malloc(size)

请求size个字节的动态内存。如果分配成功，返回所分配内存第一个字节的线性地址。

calloc(n,size)

请求含有n个元素、大小为size的一个数组。如果分配成功，把数组元素初始化为0，并返回第一个元素的线性地址。

free(addr)

释放由malloc()或calloc()分配的起始地址为addr的线性区。

brk(addr)

直接修改堆栈的大小。addr参数指定current->mm->brk的新值，返回值是线性区新的结束地址。(进程必须检查这个地址和所请求的地址值addr是否一致)。

brk()函数和以上列出的函数有所不同，因为它是唯一以系统调用的方式实现的函数，而其它所有的函数都是使用brk()和mmap()系统调用实现的C语言库函数。

当用户态的进程调用brk()系统调用时，内核执行sys_brk(addr)函数（参见第八章）。该函数首先验证addr参数是否位于进程代码所在的线性区。如果是，则立即返回：

```
mm = current->mm;
if (addr < mm->end_code)
    return mm->brk;
```

由于brk()系统调用作用于某一个线性区，它分配和释放完整的页。因此，该函数把addr的值调整为PAGE_SIZE的倍数，然后把这个结果与线性区描述符的brk域的值进行比较：

```
newbrk = (addr + 0xfff) & 0xfffff000;
oldbrk = (mm->brk + 0xfff) & 0xfffff000;
if (oldbrk == newbrk) {
    mm->brk = addr;
    return mm->brk;
}
```

如果进程请求缩小堆，sys_brk()调用do_munmap()函数完成这项任务，然后返回：

```
if (addr <= mm->brk) {
    if (!do_munmap(newbrk, oldbrk-newbrk))
        mm->brk = addr;
    return mm->brk;
}
```

如果进程请求扩大堆，sys_brk()首先检查是否允许进程这样做。如果进程企图分配在它限制范围之外的内存，该函数并不多分配内存，只简单地返回mm->brk的原有值：

```
rlim = current->rlim[RLIMIT_DATA].rlim_cur;
```

```
    if (rlim < RLIM_INFINITY && addr - mm->end_code > rlim)
        return mm->brk;
```

然后，该函数检查扩大后的堆是否和进程的其它线性区相重叠，如果是，不做任何事情就返回：

```
    if (find_vma_intersection(mm, oldbrk, newbrk+PAGE_SIZE))
        return mm->brk;
```

扩展之前所进行的最后一步检查是验证可用空闲虚拟内存的大小，看是否足以支持对堆的扩大（参见前一节“分配线性地址区间”）：

```
    if (!vm_enough_memory((newbrk-oldbrk) >> PAGE_SHIFT))
        return mm->brk;
```

如果一切都可以，则调用 do_mmap()函数并在其参数中设置 MAP_FIXED 标志。如果返回 oldbrk，则分配成功且 sys_brk()函数返回 addr 的值；否则，返回旧的 mm->brk 值：

```
        if (do_mmap(NULL, oldbrk, newbrk-oldbrk,
                PROT_READ|PROT_WRITE|PROT_EXEC,
                MAP_FIXED|MAP_PRIVATE, 0) == oldbrk)
            mm->brk = addr;
    return mm->brk;
```

对 Linux 2.4 的展望

除少量的优化和调整之外，Linux 2.4 以同样的方法处理进程的地址空间。

第八章
系统调用

操作系统为用户态运行的进程与硬件设备（如CPU、磁盘、打印机等等）进行交互提供了一组接口。在应用程序和硬件之间设置一个额外层具有很多优点。首先，这使得编程更加容易，把用户从学习硬件设备的低级编程特性中解放出来。其次，这极大地提高了系统的安全性，因为内核在试图满足某个请求之前在接口级就可以检查这种请求的正确性。最后，更重要的是这些接口使得程序更具有可移植性，因为只要任何内核所提供的一组接口相同，那么在任一内核之上就可以正确地编译和执行程序。

Unix系统利用内核所提供的系统调用（system call）实现了用户态进程和硬件设备之间的大部分接口。本章将详细讨论Linux内核是如何实现这些系统调用的。

POSIX API和系统调用

让我们先强调一下应用编程接口（API）与系统调用之不同。前者只是一个函数定义，说明了如何获得一个给定的服务；而后者是通过软中断向内核态发出一个明确的请求。

Unix系统给程序员提供了很多API的库函数。*libc*的标准C库所定义的一些API引用了封装例程（wrapper routine，其唯一目的就是发布系统调用）。通常情况下，每个系统调用对应一个封装例程，而封装例程定义了应用程序会引用的API。

反之则不然，顺便说一句，一个API没必要对应一个特定的系统调用。首先，API可能直接提供用户态的服务。（例如一些抽象的数学函数，根本没必要使用系统调用）。其次，一个单独的API函数可能调用几个系统调用。此外，几个API函数可能调用封装了不同功能的同一系统调用。例如，Linux的*libc*库函数实现了malloc()、calloc()和free()等POSIX API，这几个函数分配和释放所请求的内存，并都利用brk()系统调用来扩大或缩小进程的堆（heap）（参见第七章中的“堆的管理”一节）。

POSIX标准针对API而不针对系统调用。判断一个系统是否与POSIX兼容要看它是否提供了一组合适的应用程序接口，而不管对应的函数是如何实现的。事实上，一些非Uuix系统被认为是与POSIX兼容的，是因为它们在用户态的库函数中提供了传统Unix能提供的所有服务。

从编程者的观点看，API和系统调用之间的差别是没有关系的：唯一相关的事情就是函数名、参数类型及返回代码的含义。然而，从内核设计者的观点看，这种差别确实有关系，因为系统调用属于内核，而用户态的库函数不属于内核。

大部分封装例程返回一个整数，其值的含义依赖于相应的系统调用。返回值-1，在多数情况下（但不是经常），表示内核不能满足进程的请求。系统调用处理程序的失败可能是由无效参数引起的，也可能是因为缺乏可用资源，或硬件出了问题等等。在*libc*库中定义的errno变量包含特定的错误代码。

每个错误代码都与一个产生相应正整数值的宏相关。POSIX标准指定了很多错误代码的宏名。在基于Intel 80x86的Linux中，在一个叫做*include/asm-i386/errno.h*的头文件中定义了这些宏。为了使各种Unix系统上的C程序具有可移植性，在标准的C库头文件*/usr/include/errno.h*中也包含了*include/asm-i386/errno.h*头文件。其他的系统有它们自己专门的头文件子目录。

系统调用处理程序及服务例程

当用户态的进程调用一个系统调用时，CPU切换到内核态并开始执行一个内核函数。Linux对系统调用的调用必须通过执行int $0x80汇编指令，这条汇编指令产生向量为128的编程异常（参见第四章中的“中断、陷阱及系统门”及“中断和异常的硬件处理”两节）。

因为内核实现了很多不同的系统调用，因此进程必须传递一个叫做系统调用号的参数来识别所需的系统调用，eax寄存器就用作此目的。正如我们将在本章的“参数传递”一节看到的，当调用一个系统调用时通常还要传递另外的参数。

所有的系统调用都返回一个整数值。这些返回值与封装例程返回值的约定是不同的。在内核中，正数或0表示系统调用成功结束，而负数表示一个错误条件。在后一种情况下，这个值就是必须返回给应用程序错误码的负数。内核没有设置或使用errno变量。

与其他异常处理程序的结构类似，系统调用处理程序执行下列操作：

- 在内核栈保存大多数寄存器的内容（这个操作对所有的系统调用都是通用的，并用汇编语言编写）。
- 调用所谓系统调用服务例程的相应的C函数来处理系统调用。
- 通过ret_from_sys_call()函数从系统调用返回（这个函数用汇编语言编写）。

xyz()系统调用对应的服务例程的名字通常是sys_*xyz*()。不过，还有一些例外。

图8-1显示了调用系统调用的应用程序、相应的封装例程、系统调用处理程序及系统调用服务例程之间的关系。箭头表示函数之间的执行流。

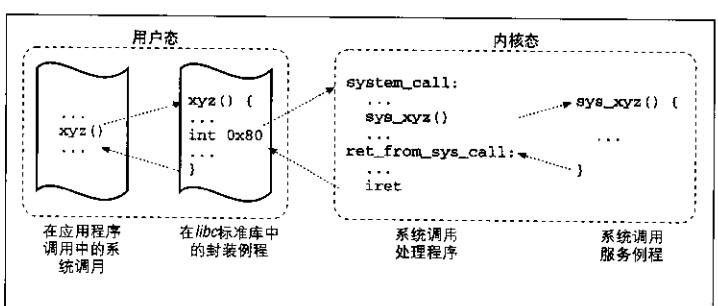


图8-1 调用一个系统调用

为了把系统调用号与相应的服务例程关联起来，内核利用了一个系统调用分派表(dispatch table)。这个表存放在sys_call_table数组中，有NR_syscalls个表项(通常是256个)：第n个表项包含系统调用号为n的服务例程的地址。

NR_syscalls宏只是对可实现的系统调用最大个数的静态限制，并不表示实际已实现的系统调用个数。实际上，分派表中的任一个表项也可以包含sys_ni_syscall()函数的地址，这个函数是“未实现”系统调用的服务例程，它仅仅返回错误码-ENOSYS。

初始化系统调用

内核初始化期间调用的trap_init()函数建立IDT表中向量128对应的表项，语句如下：

```
set_system_gate(0x80, &system_call);
```

该调用把下列值装入这个门描述符的相应域(参见第四章中的“中断、陷阱及系统门”一节)：

Segment Selector

内核代码段__KERNEL_CS的段选择符。

Offset

指向system_call()异常处理程序。

Type

置为15。表示这个异常是一个陷阱，相应的处理程序不禁止可屏蔽中断。

DPL(描述符特权级)

置为3。这就允许用户态进程调用这个异常处理程序(参见第四章中的“中断和异常的硬件处理”一节)。

system_call()函数

system_call()函数实现了系统调用处理程序。它首先把系统调用号和这个异常处理程序可以用到的所有CPU寄存器保存到相应的栈中，除了由控制单元已自动保存

的eflags、cs、eip、ss和esp寄存器（参见第四章中的“中断和异常的硬件处理”一节）。在第四章的“为中断处理程序保存寄存器的值”一节中已经讨论的SAVE_ALL宏，也在ds和es中装入内核数据段的段选择符：

```
system_call:
  pushl %eax
  SAVE_ALL
  movl %esp, %ebx
  andl $0xfffe000, %ebx
```

这个函数也在ebx中存放current进程描述符的地址，这是通过获得内核栈指针的值并把它取整到8KB的倍数而完成的（参见第三章中的“标识一个进程”一节）。

然后，对用户态进程传递来的系统调用号进行有效性检查。如果这个号大于或等于NR_syscalls，系统调用处理程序终止：

```
cmpl $(NR_syscalls), %eax
 jb nobadsys
 movl $(-ENOSYS), 24(%esp)
 jmp ret_from_sys_call
 nobadsys:
```

如果系统调用号无效，函数就把-ENOSYS值存放在栈中已保存eax寄存器的单元中（从当前栈顶开始偏移为24的单元）。然后跳到ret_from_sys_call()。当进程以这种方式恢复它在用户态的执行时，会在eax中发现一个负的返回码。

接下来，system_call()函数检查current的flags域所包含的PF_TRACESYS标志是否等于1，也就是检查是否有某一调试程序正在跟踪执行的程序对系统调用的调用。如果是这种情况，system_call()函数两次调用syscall_trace()函数，一次正好在这个系统调用服务例程执行之前，一次在其之后。这个函数停止current，并因此允许调试进程收集关于current的信息。

最后，调用与eax中所包含的系统调用号对应的特定服务例程：

```
call *sys_call_table(0, %eax, 4)
```

因为分派表中的每一表项占4个字节，因此首先把系统调用号乘以4，再加上sys_call_table分派表的起始地址，然后从这个地址单元获取指向服务例程的指针，内核就找到了要调用的服务例程。

当服务例程结束时，system_call()从eax获得它的返回值，并把这个返回值存放在曾保存用户态eax寄存器栈单元的那个位置上。然后跳转到ret_from_sys_call()，终止系统调用处理程序的执行（参见第四章中的“ret_from_sys_call()函数”一节）。

```
movl %eax, 24(%esp)
jmp ret_from_sys_call
```

当进程恢复它在用户态的执行时，就可以在eax中找到系统调用的返回码。

参数传递

与普通函数类似，系统调用通常也需要输入/输出参数，这些参数可能是实际的值（例如数值），也可能是函数的地址及用户态进程地址空间的变量。因为system_call()函数是Linux中所有系统调用唯一的入口点，因此每个系统调用至少有一个参数，即通过eax寄存器传递来的系统调用号。例如，如果一个应用程序调用fork()封装例程，在执行int $0x80汇编指令之前就把eax寄存器置为5。因为这个寄存器的设置是由*libc*库中的封装例程进行的，因此程序员通常并不用关心系统调用号。

fork()系统调用并不需要其他的参数。不过，很多系统调用确实需要由应用程序明确地传递另外的参数。例如，mmap()系统调用可能需要多达6个参数（除了系统调用号）。

普通函数的参数传递是通过把参数值写进活动的程序栈（或者用户态栈或者内核态堆栈）。但是系统调用的参数通常是传递给系统调用处理程序在CPU中的寄存器，然后再拷贝到内核态堆栈，这是因为系统调用服务例程是普通的C函数。

为什么内核不直接把参数从用户态的栈拷贝到内核态的栈呢？首先，同时操作两个栈是比较复杂的。此外，寄存器的使用使得系统调用处理程序的结构与其他异常处理程序的结构类似。

然而，为了用寄存器传递参数，必须满足两个条件：

- 每个参数的长度不能超过寄存器的长度，即32位（注1）。

注1：这是仍然指的是Intel 80x86处理器的32位体系结构。这部分的讨论并不适用于康柏的Alpha 64位处理器。

- 参数的个数不能超过6个（包括eax中传递的系统调用号），因为Intel Pentium寄存器的数量是有限的。

第一个条件总能成立，因为根据POSIX标准，不能存放在32位寄存器中的长参数必须通过指定它们的地址来传递。一个典型的例子就是settimeofday()系统调用，它必须读两个64位的结构。

然而，确实存在多于6个参数的系统调用。在这样的情况下，用一个单独的寄存器指向进程地址空间中这些参数值所在的一个内存区。当然，编程者不用关心这个工作区。正如任何C调用一样，当调用封装例程时，参数被自动地保存在栈中。封装例程将找到合适的方式把参数传递给内核。

存放系统调用参数所用的6个寄存器是（以字母递增的顺序）：eax（存放系统调用号）、ebx、ecx、edx、esi及edi。正如以前看到的一样，system_call()使用SAVE_ALL宏把这些寄存器的值保存在内核态堆栈中。因此，当系统调用服务例程转到内核态堆栈时，就会找到system_call()的返回地址，紧接着是存放在ebx中的参数（即系统调用的第一个参数）、存放在ecx中的参数等等（参见第四章中的“为中断处理程序保存寄存器的值”一节）。这种栈结构与普通函数调用的栈结构完全相同，因此，服务例程可以很容易地引用使用一般C语言构造的参数。

让我们来看一个例子。处理write()系统调用的sys_write()服务例程的声明如下：

```
int sys_write (unsigned int fd, const char * buf,
               unsigned int count)
```

C编译器产生一个汇编语言函数，该函数期望找到存放在栈顶部的fd、buf和count参数，而这些参数位于返回地址（就是用来分别存放ebx、ecx和edx寄存器的那些位置）的下面。

在几种情况下，即使系统调用不使用任何参数，相应的服务例程也需要知道在发出系统调用之前CPU寄存器的内容。例如，实现了fork()的do_fork()函数需要知道有关寄存器的值以便在子进程的TSS中复制它们。在这些情况下，类型为pt_regs的一个单独参数允许服务例程访问由SAVE_ALL宏保存在内核态堆栈中的值（参见第四章能中的“do_IRQ()函数”一节）：

```
int sys_fork (struct pt_regs regs)
```

服务例程的返回值必须写到eax寄存器中。这是在执行return n;指令时由C编译程序自动完成的。

验证参数

在内核打算满足用户的请求之前，必须仔细地检查所有的系统调用参数。检查的类型既依赖于系统调用，也依赖于特定的参数。让我们再回到前面引入的write()系统调用。fd参数应该是描述一个特定文件的文件描述符，因此，sys_write()必须检查fd是否确实是以前已打开文件的一个文件描述符，是否允许进程向这个文件中写数据（参见第一章中的"文件操作的系统调用"一节）。如果这些条件中有一个不成立，那么这个处理程序必须返回一个负数，在这种情况下的错误代码为-EBADF。

然而，有一种检查对所有的系统调用都是通用的：只要一个参数指定的是地址，那么内核必须检查它是否在这个进程的地址空间之内。有两种可能的方式来执行这种检查：

- 验证这个线性地址是否属于进程的地址空间，如果是，这个线性地址所在的线性区就具有正确的访问权。
- 仅仅验证这个线性地址小于PAGE_OFFSET（即没有落在留给内核的线性地址区间内）。

因此，Linux 2.2内核执行第二种检查。这是一种更高效的检查，因为不需要对进程的线性区描述符进行任何扫描。很显然，这是一种非常粗略的检查，验证线性地址小于PAGE_OFFSET是判断它的有效性的必要条件但不是充分条件。但是，因为其它的错误可以在随后捕获到，因此，内核使用这种有限的检查没有任何风险。

Linux 2.2接着采用的方法是将真正的检查尽可能向后推迟，也就是说，推迟到分页单元将线性地址转换为物理地址时。我们将在本章稍后的"动态地址检查：修正代码"一节中讨论，"缺页"异常处理程序如何成功地检测到由用户态进程以参数传递的这些地址在内核态是无效的。

在这里，你可能想知道究竟为什么要进行这种粗略检查。事实上，这种粗略的检查是至关重要的，它确保了进程地址空间和内核地址空间都不被非法访问。我们在第二章中已经看到，RAM的映射是从PAGE_OFFSET开始的。这就意味着内核例程能

对内存中现有的所有页进行寻址。因此，如果不进行这种粗略检查，用户态进程就可能把属于内核地址空间的一个地址作为参数来传递，然后还能对内存中现有的任何页进行读写而不引起“缺页”异常！

对系统调用所传递地址的检查是通过verify_area()函数实现的，它有两个参数（注2）分别为addr和size。该函数检查addr到addr + size - 1之间的地址区间，本质上等价于下面的C函数：

```
int verify_area(const void * addr, unsigned long size)
{
    unsigned long a = (unsigned long) addr;
    if (a + size < a || a + size > current->addr_limit.seg)
        return -EFAULT;
    return 0;
}
```

该函数首先验证addr+size（要检查的最高地址）是否大于2^{32}-1。这是因为GNU C编译器（gcc）用32位数表示无符号长整型数和指针，这就等价于对溢出条件进行检查。该函数还检查addr是否超过crurrent的addr_limit.seg域中存放的值。通常情况下，普通进程这个域的值是PAGE_OFFSET-1，内核线程是0xffffffff。可以通过get_fs和set_fs宏动态地改变addr_limit.seg域的值。这就允许内核直接调用系统调用的服务例程，并把内核数据段的地址传递给它们。

access_ok宏与verify_area()执行同样的检查。如果指定地址区间合法，则这个宏返回1，否则返回0。

访问进程地址空间

系统调用服务例程需要非常频繁地读写进程地址空间的数据。Linux包含的一组宏使这种访问更加容易。我们将描述其中的两个叫做get_user()和put_user()的宏。第一个宏用来从一个地址读取1、2或4个连续字节，而第二个宏用来把这几种大小的内容写到一个地址中。

注2：第三个参数为type，这个参数指定系统调用对所涉及的内存单元进行读还是进行写。它仅用在Intel 80486微处理器有bug的系统中，在这种微处理器中在内核态对一个写保护页进行写不产生缺页。我们不进一步讨论这种情况。

这两个函数都接受两个参数，一个要传送的值x和一个变量ptr。第二变量还决定有多少个字节要传送。因此，在get_user(x, btr)中，变量ptr所指的大小会使这个函数展开为__get_user_1()、__get_user_2()或__get_user_4()汇编语言函数。让我们看一下其中一个，比如，__get_user_2()：

```
__get_user_2:
    addl $1, %eax
    jc bad_get_user
    movl %esp, %edx
    andl $0xffffe000, %edx
    cmpl 12(%edx), %eax
    jae bad_get_user
2:  movzwl -1(%eax), %edx
    xorl %eax, %eax
    ret
bad_get_user:
    xorl %edx, %edx
    movl $-EFAULT, %eax
    ret
```

eax寄存器包含要读取的第一个字节的地址ptr。前六个指令所执行的检查事实上与verify_area()相同，即确保要读取的两个字节的地址小于4GB并小于current进程的addr_limit.seg域（这个域位于进程描述符偏移12处，出现在cmpl指令的第一个操作数中）。

如果这个地址有效，该函数就执行movzwl指令，把要读的数据存到edx寄存器的两个低字节而把两个高字节置为0，然后在eax中设置返回码0并终止。如果这个地址无效，该函数清edx，将eax置为-EFAULT并终止。

put_user(x, ptr)宏类似于前边讨论的get_user，但它把值x写到以地址ptr为起址的进程地址空间。根据x的大小(1、2或4字节)，它调用__put_user_1()、__put_user_2()或__put_user_4()函数。这次我们以__put_user_4()作为例子。该函数对存放在eax中的地址ptr执行常规检查，然后执行movl指令把4个字节写到edx中。如果成功返回0，否则返回-EFAULT。

在表8-1中列出了内核态下用来访问进程地址空间的另外几个函数或宏。注意许多函数或宏的名字前缀有两个下划线(__)。首部没有下划线的函数或宏要用额外的时间对所请求的线性地址区间进行有效性检查，而有下划线的则会跳过检查。当内

核必须重复访问进程地址空间的同一块线性区时，比较高效的办法是开始时只对该地址检查一次，以后就不用再对该进程区进行检查。

表 8-1 访问进程地址空间的函数和宏

函数	操作
get_user __get_user	从用户空间读一个整数（1、2 或 4 字节）
put_user __put_user	给用户空间写一个整数（1、2 或 4 字节）
get_user_ret __get_user_ret	类似于 get_user，但返回一个指定的错误值
p_user_ret __put_user_ret	类似于 put_user，但返回一个指定的错误值
copy_from_user __copy_from_user	从用户空间拷贝任意大小的块
copy_to_user __copy_to_user	把任意大小的块拷贝到用户空间
copy_from_user_ret	类似于 copy_from_user，但返回一个指定的错误值
copy_to_user_ret	类似于 copy_to_user，但返回一个指定的错误值
strncpy_from_user __strncpy_from_user	从用户空间复制一个以 null 结束的字符串
strlen_user strnlen_user	返回用户空间以 null 结束的字符串的长度
clear_user __clear_user	用 0 填充用户空间的一个内存区域

动态地址检查：修正代码（fixup code）

正如前面所看到的，verify_area()函数与access_ok宏对系统调用以参数传递来的线性地址的有效性只进行粗略检查。因为它们并不能确保这些地址被包含在用户的地址空间，进程可能因为传递一个错误的地址而引起“缺页”异常。

在描述内核如何检测这种错误之前，我们先说明一下在内核态引起“缺页”异常的三种情况：

- 内核试图访问属于进程地址空间的页，但是，或者是相应的页框不存在，或者是内核试图去写一个只读页。
- 某一内核函数包含程序设计错误，当这个函数运行时就引起异常；或者，可能由于瞬时的硬件错误引起异常。
- 本章所讨论的一种情况：一个系统调用服务例程试图读写一个内存区，而该内存区的地址是通过系统调用参数传递来的，但却不属于进程的地址空间。

必须由缺页处理程序对这些情况加以区分，因为对每种情况所采取的行动有很大的差别。在第一种情况下，缺页处理程序必须分配并初始化一个新的页框（参看第七章的“请求调页”和“写时复制”两节）；在第二种情况下，缺页处理程序必须执行一个内核 oops（参看第七章的“处理地址空间以外的错误地址”一节）；在第三种情况下，缺页处理程序必须通过返回适当的错误码而终止系统调用。

通过确定错误的线性地址是否属于进程所拥有的线性地址区间，缺页处理程序可以很容易地识别第一种情况。下面让我们解释一下缺页处理程序如何区分另外两种情况。

异常表

缺页发生的根源在于内核通过系统调用所访问的进程地址空间非常有限。只有前面描述的少数函数和宏可以用来访问进程的地址空间。因此，如果一个无效参数引起异常，那么，引起异常的指令一定是在这些函数或宏展开的代码中。如果你把所有这些函数和宏的代码加起来，会发现它们形成的地址范围很窄。

因此，我们把访问进程地址空间的任一条内核指令放到一个叫异常表（exceptfion table）的结构中并不用费太多功夫。如果我们成功地做到这点，其他的事情就很容易了。当在内核态发生页异常时，do_page_fault()处理程序检查异常表。如果表中包含产生异常的指令地址，那么这个错误就是由非法的系统调用参数引起的，否则，就是由某一更严重的 bug 引起的。

Linux 定义了几个异常表。主要的异常表在建立内核程序映像时由 C 编译器自动生

成。它存放在内核代码段的__ex_table部分，其起始与终止地址由C编译器产生的两个符号：__start__ex_table和__stop__ex_table标识。

此外，每个动态装载的内核模块（参看附录二）都包含有自己的局部异常表。这个表是在建立模块映像时由C编译器自动产生的，当把模块插入到运行中的内核时把这个表装入到内存。

每一个异常表的表项是一个exception_table_entry结构，它有两个域：

insn
: 访问进程地址空间的指令的线性地址

fixup
: 当存放在insn单元中的指令所触发的“缺页”异常发生时，fixup就是要调用的汇编语言代码的地址

修正代码由几条汇编指令组成，用以解决由缺页异常所引起的问题。在后面我们将会看到，修正通常包括插入一个指令序列，这个指令序列强制服务例程返回一个错误码给用户态进程。这些指令通常出现在访问进程地址空间的同一函数或宏中。有时由C编译器把它们放置在内核代码段的一个叫做.fixup的独立部分。

search_exception_table()用来在所有异常表中查找一个指定地址。若这个地址在某一个表中，则返回相应的fixup地址；否则，返回0。因此，缺页处理程序do_page_fault()执行下列语句：

```
if ((fixup = search_exception_table(regs->eip)) != 0) {
    regs->eip = fixup;
    return;
}
```

regs->eip域包含异常发生时保存到内核栈eip寄存器中的值。如果这个值在某个异常表中，do_page_fault()就把它替换为search_exception_table()的返回地址。然后缺页处理程序终止，被中断的程序以修正代码的执行而恢复。

生成异常表和修正代码

GNU汇编程序（Assembler）.section伪指令允许程序员指定可执行文件哪部分包

含紧接着要执行的代码。我们将在第十九章中看到，可执行文件包括一个代码段，这个代码段可能又依次被划分为一些小的片断。因此，下面的汇编指令在异常表中加入一个表项；"a"属性指定了必须把这一节（section）与内核映像的剩余部分一块加载到内存中。

```
.section __ex_table, "a"
    .long faulty_instruction_address, fixup_code_address
.previous
```

.previous伪指令强制汇编程序把紧接着的代码插入到遇到上一个.section伪指令时激活的节。

我们再看一下前面提到的__get_user_1()、__get_user_2()和__get_user_4()函数：

```
__get_user_1:
    [...]
1:  movzbl (%eax), %edx
    [...]
__get_user_2:
    [...]
2:  movzwl -1(%eax), %edx
    [...]
__get_user_4:
    [...]
3:  movl -3(%eax), %edx
    [...]
bad_get_user:
    xorl %edx, %edx
    movl $-EFAULT, %eax
    ret
.section __ex_table, "a"
    .long 1b, bad_get_user
    .long 2b, bad_get_user
    .long 3b, bad_get_user
.previous
```

访问进程地址空间的指令分别被标记为1、2和3。修正代码对这三个函数是公用的并被标记为bad_get_user。每一个异常表的表项都简单地由两个标号组成。第一个是以b为后缀的数字标号，表示这是一个“向后”的标号，换句话说，它出现在

程序的前一行中。在bad_get_user中的修正代码给发出系统调用的进程返回一个错误码。

再看下一个例子，strlen_user(string)宏。它返回进程地址空间中一个以null结束的字符串的长度，或错误时返回0。实质上，这个宏产生以下的汇编指令：

```
        movl $0, %eax
        movl $0x7fffffff, %ecx
        movl %ecx, %edx
        movl string, %edi
    0:  repne; scasb
        subl %ecx, %edx
        movl %edx, %eax
    1:
    .section .fixup,"ax"
    2:  movl $0, %eax
        jmp 1b
    .previous
    .section __ex_table,"a"
        .long 0b, 2b
    .previous
```

ecx和edx的初始值设为0x7fffffff，表示字符串的最大长度。repne; scasb汇编指令循环扫描由edi指向的字符串，在eax中查找值为0的字符（字符串的结束标志\0字符）。因为每一次循环ecx都减1，最后eax中存放的是在字符串中所扫描过的字节总数，也就是字符串的长度。

把这个宏的fixup代码插入.fixup节。"ax"属性指定这一节（section）必须被加载到内存中并包含可执行代码。如果缺页异常是由标号为0的指令引起，就执行fixup代码，它只简单地把eax置为0，因此强制该宏返回一个错误码0而不是字符串长度。然后跳转到标号1，即宏之后的相应指令。

封装例程

尽管系统调用主要由用户态进程使用，但也可以被内核线程调用，内核线程不能使用库函数。为了简化相应的封装例程的声明，Linux定义了六个从_syscall0到_syscall5的宏。

每个宏名字中的数字0到5对应着系统调用所用的参数号（系统调用号除外）。也可以用这些宏来简化*libc*标准库中封装例程的声明。然而，不能用这些宏来为超过5个参数（系统调用号除外）的系统调用或产生非标准返回值的系统调用定义封装例程。

每个宏严格地需要2+2 × *n*个参数，*n*是系统调用的参数个数。前两个参数指明系统调用的返回值类型和名字；每一对附加参数指明相应的系统调用参数的类型和名字。因此，以fork()系统调用为例，其封装例程可以通过如下语句产生：

```
_syscall0(int,fork)
```

而write()系统调用的封装例程可以通过如下语句生产：

```
_syscall3(int, write,int,fd,const char *,buf,unsigned int,count)
```

在后一种情况下，可以把这个宏展开成如下的代码：

```
int write(int fd, const char * buf,unsigned int count)
{
    long _ _res;
    asm("int $0x80"
        : "=a" (_ _res)
        : "0" (_ _NR_write), "b" ((long)fd),
          "c" ((long)buf), "d" ((long)count));
    if ((unsigned long)_ _res >= (unsigned long)-125) {
        errno = -_ _res;
        _ _res = -1;
    }
    return (int) _ _res;
}
```

__NR_write宏来自_syscall3的第二个参数。它可以展开成write()的系统调用号，当编译前面的函数时，生成下面的汇编代码：

```
write:
     pushl %ebx              ; 将ebx推入堆栈
     movl 8(%esp), %ebx      ; 将第一个参数放入ebx
     movl 12(%esp), %ecx     ; 将第二个参数放入ecx
     movl 16(%esp), %edx     ; 将第三个参数放入edx
     movl $4, %eax           ; 将_ _NR_write放入eax
     int $0x80               ; 进行系统调用
     cmpl $-126, %eax        ; 检测返回码
```

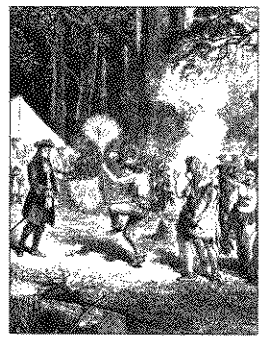
```
        jbe .L1                 ; 如无错跳转
        negl %eax               ; 求eax的补码
        movl %eax, errno        ; 将结果放入errno
        movl $-1, %eax          ; 将eax置为-1
.L1:    popl %ebx               ; 从堆栈弹出ebx
        ret                     ; 返回调用程序
```

注意write()函数的参数是如何在执行int $0x80指令前被装入到CPU寄存器中的。如果eax中的返回值在-1和-125之间，必须被解释为错误码(内核假定在*include/asm-i386/errno.h*中定义的最大错误码为125)。如果是这种情况，封装例程在errno中存放-eax的值并返回值-1；否则，返回eax中的值。

对Linux 2.4的展望

除了添加一些新的系统调用外，Linux 2.4对Linux 2.2的系统调用机制没有进行任何改变。

第九章
信号



Unix系统中首先引入的信号简化了进程间通信。内核也用信号通知进程系统所发生的事件。与中断和异常不同，大多数信号对用户态的进程是可见的。

信号已存在了30年，只有很小的变化。尽管我们将在第十八章中看到，出于相同的目的也引入了其他高级通信工具，但由于信号相对简单而有效，它们仍被广泛地使用着。

本章的第一部分详细考察Linux内核是如何处理信号的，然后，我们讨论几个允许进程交换信号的系统调用。

信号的作用

信号是很短的信息，可以被发送到一个进程或一组进程。发送给进程的这个唯一信息通常是标识信号的一个数。在标准信号中，没有给参数、消息、或者其他相随的信息留有空间。

名字前缀为SIG的一组宏用来标识信号。在前几章中，我们已经涉及到几个信号。例如，在第三章的“clone()，fork()及vfork()系统调用”一节中已提及到SIGCHLD宏。在Linux中，这个宏的值为17，当某一子进程停止或终止时，SIGCHLD宏产生发送给父进程的信号标识符。SIGSEGV宏的值为11，在第七章的“缺页异常处理

程序”一节中已提及到：当一个进程引用无效的内存时，SIGSEGV宏产生发送给这个进程的信号标识符。

使用信号的两个主要目的是：

- 让进程知道已经发生了一个特定的事件
- 强迫进程执行它自己代码中的信号处理程序

当然，两个目的不是互斥的，因为进程对某一事件的反映是通过执行一个特定的例程进行的。

表9-1列出基于Intel 80x86的Linux 2.2所处理的前31个信号（诸如SIGCHLD或SIGSTOP这样的一些信号是与体系结构相关的；此外，一些信号只为特定的体系结构而定义）。除了这个表中所描述的信号外，POSIX标准已经引入一种新的类型，叫“实时”信号。在本章后面的“实时信号”一节中将单独讨论这种信号。

表9-1 在Linux/i386中的前31个信号

编号	信号名称	缺省操作	解释	POSIX
1	SIGHUP	Abort	挂断控制终端或进程	是
2	SIGINT	Abort	来自键盘的中断	是
3	SIGQUIT	Dump	来自键盘的退出	是
4	SIGILL	Dump	非法指令	是
5	SIGTRAP	Dump	跟踪断点	否
6	SIGABRT	Dump	异常结束	是
6	SIGIOT	Dump	等价于SIGABRT	否
7	SIGBUS	Abort	总线错误	否
8	SIGFPE	Dump	浮点异常	是
9	SIGKILL	Abort	强迫进程终止	是
10	SIGUSR1	Abort	进程可以使用	是
11	SIGSEGV	Dump	无效的内存引用	是
12	SIGUSR2	Abort	进程可以使用	是
13	SIGPIPE	Abort	向无读者的管道写	是
14	SIGALRM	Abort	实时定时器时钟	是

表 9-1 在 Linux/i386 中的前 31 个信号（续）

编号	信号名称	缺省操作	解释	POSIX
15	SIGTERM	Abort	进程终止	是
16	SIGSTKFLT	Abort	协处理器栈错误	否
17	SIGCHLD	Ignore	子进程停止或结束	是
18	SIGCONT	Continue	如果已停止则恢复执行	是
19	SIGSTOP	Stop	停止进程执行	是
20	SIGTSTP	Stop	从 tty 发出停止进程	是
21	SIGTTIN	Stop	后台进程请求输入	是
22	SIGTTOU	Stop	后台进程请求输出	是
23	SIGURG	Ignore	套接字上的紧急条件	否
24	SIGXCPU	Abort	超过 CPU 时限	否
25	SIGXFSZ	Abort	超过文件大小的限制	否
26	SIGVTALRM	Abort	虚拟定时器时钟	否
27	SIGPROF	Abort	概况定时器时钟	否
28	SIGWINCH	Ignore	窗口调整大小	否
29	SIGIO	Abort	I/O 现在可能发生	否
29	SIGPOLL	Abort	等价于 SIGIO	否
30	SIGPWR	Abort	电源供给失效	否
31	SIGUNUSED	Abort	没有使用	否

许多系统调用允许程序开发者发送信号，并决定他们的进程如何使用所接收的信号。表 9-2 简洁地描述了这些系统调用，更详细的内容将在后面“与信号处理相关的系统调用”一节中描述。

表 9-2 与信号相关的系统调用

系统调用	描述
kill()	向进程发送一个信号
sigaction()	改变与信号相关的操作
signal()	类似于 sigaction()

表 9-2 与信号相关的系统调用（续）

系统调用	描述
sigpending()	检查是否有挂起的信号
sigprocmask()	修改阻塞信号屏蔽码
sigsuspend()	等待一个信号
rt_sigaction()	改变与实时信号相关的操作
rt_sigpending()	检查是否挂起实时信号
rt_sigprocmask()	修改实时信号的屏蔽码
rt_sigqueueinfo()	向进程发送一个实时信号
rt_sigsuspend()	等待一个实时信号
rt_sigtimedwait()	类似于 rt_sigsuspend()

信号的一个重要特点是它们可以被随时发送给状态经常不可预知的进程。发送给非运行进程的信号必须由内核保存，直到进程恢复执行。阻塞的信号（后面描述）要求信号排队，这使得刚产生的信号还没有被传递出去这一问题更加严重。

因此，内核区分与信号传递有关的两个不同阶段：

信号发送：

内核更新目标进程的描述符以表示一个新的信号已被发送。

信号的接收

内核强迫目标进程对这个信号做出反映，通过改变它的执行状态，或开始执行一个特定的信号处理程序，或两者都是。

每个被发送的信号只被接收一次。信号是可消费资源，一旦它们已被接收，进程描述符中有关这个信号的所有信息被取消。

已发送但还没被接收的信号被称为挂起信号（pending signal）。任何时候，一个进程仅存在给定类型的一个挂起信号，同一进程同样类型的其他信号不被排队，只被简单地丢弃。一般来说，可以把挂起的信号保留不可预知的时间。的确，必须考虑下列因素：

- 信号通常只被当前正运行的进程接收（即由 current 进程接收）。

- 给定类型的信号可以由进程选择性地阻塞（blocked）（参见“修改被阻塞信号的集合”一节）：这种情况下，在取消这个阻塞前进程将不接收这个信号。
- 当进程执行一个信号处理程序的函数时，通常“屏蔽”相应的信号，即自动阻塞这个信号直到这个处理程序结束。因此，已处理的信号的另一次出现不能中断信号处理程序，所以，这个函数不必是可重入的。一个被掩码（masked）的信号总是被阻塞，但反之则不然。

尽管信号的表示比较直观，但内核的实现相当复杂。内核必须：

- 记住每个进程阻塞哪些信号。
- 当从内核态切换到用户态时，对任何一个进程都要检查是否有一个信号已到达。这几乎在每个定时中断时都发生，即，大约每 10ms 发生一次。
- 确定是否可以忽略该信号。这发生在下列所有的条件都被满足时：
  - 目标进程没有被另一个进程跟踪（进程描述符中 flags 域的 PF_TRACED 标志等于 0，注 1）。
  - 这个信号没有被目标进程阻塞。
  - 这个信号被目标进程忽略（或者因为进程已显式地忽略了这个信号，或者因为进程没有改变信号的缺省操作，这个缺省操作就是“忽略”）。
- 处理信号，这个信号可能在进程运行期间的任一时刻请求把进程切换到一个信号处理函数，并在这个函数返回以后恢复原来执行的上下文。

此外，Linux 必须考虑 BSD 和 System V 所采用的不同的信号语义，而且，还必须与相当麻烦的 POSIX 要求相兼容。

接收信号之前所执行的操作

进程以三种方式对一个信号做出应答：

- 显式地忽略这个信号。

注 1：如果一个进程正在被跟踪时接收到一个信号，内核就停止这个进程，并向跟踪进程发送一个 SIGCHLD 信号以通知它一下。跟踪进程又可以使用 SIGCOUNT 信号重新恢复被跟踪进程的执行。

- 执行与这个信号相关的缺省操作（参见表9-1）。由内核预定义的这个操作依赖于信号的类型，缺省操作可以是下列类型之一：

  Abort

  进程被撤消（杀死）。

  Dump

  进程被撤消（杀死），并且，如果可能，创建包含进程执行上下文的core文件。这个文件可以用于调试。

  Ignore

  信号被忽略。

  Stop

  进程被停止，即把进程放进TASK_STOPPED状态（参见第三章的“进程状态”一节）。

  Continue

  如果进程被停止（TASK_STOPPED），把它放进TASK_RUNNING状态。

- 通过调用相应的信号处理函数捕获信号。

注意，对一个信号的阻塞和忽略是不同的：当信号被阻塞时，它再也不会被接收；而一个忽略的信号总是被接收，只是没有进一步的操作。

SIGKILL和SIGSTOP信号不可以被显式地忽略或捕获，因此，它们的缺省操作通常必须执行。所以，SIGKILL和SIGSTOP允许具有适当特权的用户分别撤消和停止任何进程（注2），不管程序执行时采取怎样的防御措施。

与信号相关的数据结构

用一个基本的数据结构sigset_t类型来存放发送给进程的信号，这个结构是一个位数组，每种信号类型对应一位：

注2：实际上，有两个例外：发送给进程0（*swapper*）的所有信号被丢弃，而发送给进程1（*init*）的信号在捕获到它们之前也总被丢弃。因此，进程0永不死亡，而进程1只有当*init*终止时才死亡。

```
typedef struct {
    unsigned long sig[2];
} sigset_t;
```

因为每个无符号长整数由32位组成，在Linux中可以声明的信号最大数是64(_NSIG宏表示这个值)。没有编号为0的信号，因此，sigset_t的第一个元素中的其他31位就是表9-1中所列出的标准信号。信号1映射到位0，信号2映射到位1，等等。第二个元素中的位就是实时信号。在进程描述符中所包含的下列域跟踪发送给进程的信号：

signal

类型为sigset_t的一个变量，表示发送给进程的信号。

blocked

类型为sigset_t的一个变量，表示被阻塞的信号。

sigpending

一个标志，如果一个或更多的非阻塞信号正在挂起，设置这个标志。

gsig

指向signal_struct数据结构的一个指针，signal_struct描述必须怎样处理每个信号。

signal_struct结构依次被定义如下：

```
struct signal_struct {
    atomic_t            count;
    struct k_sigaction  action[64];
    spinlock_t          siglock;
};
```

正如在第三章的“clone()，fork()及vfork()系统调用”一节中所提到的，通过调用设置了CLONE_SIGHAND标志（注3）的clone()系统调用，这个结构可以由几个进程共享。count域指定共享signal_struct结构的进程个数，而siglock域用来确保对signal_struct域的互斥访问（参见第十一章）。action域是64个元素的一个数组，其类型为k_sigaction结构，k_sigaction指定必须怎样处理每个信号。

注3：如果不这么做，仅仅为了对信号进行处理就得把大约1300个字节加到进程的数据结构。

一些体系结构把特性赋给仅对内核可见的信号。因此，一个信号的特性存放在k_sigaction结构中，k_sigaction结构既包含对用户态的进程所隐藏的特性，也包含大家熟悉的sigaction结构，而sigaction结构保存了用户态的进程能看见的所有特性。实际上，Intel平台信号的所有特性对用户态的进程都是可见的。因此，k_sigaction结构只不过简化为类型为sigaction的单个sa结构，sigaction结构包含下列域：

sa_handler

这个域指定要执行操作的类型。它的值可以是指向信号处理程序的一个指针，SIG_DFL（即值0）指定必须执行缺省操作，或者SIG_IGN（即值1）指定必须显式地忽略这个信号。

sa_flags

这是一个标志集，指定必须怎样处理信号。其中的一些标志在表9-3中列出。

sa_mask

这是类型为sigset_t的变量，指定当运行信号处理程序时要屏蔽的信号。

表9-3 指定如何处理信号的一组标志

标志名	描述
SA_NOCLDSTOP	当进程被停止时不给父进程发送SIGCHLD信号
SA_NODEFER, SA_NOMASK	当执行信号处理程序时不屏蔽这个信号
SA_RESETHAND, SA_ONESHOT	执行信号处理程序以后重置缺省操作
SA_ONSTACK	为信号处理程序的执行使用一个预备的栈（参见后面“系统调用的重新执行”一节）
SA_SIGINFO	为信号处理程序提供另外的信息（参见后面“改变信号的操作”一节）

在信号数据结构上的操作

内核使用几个函数和宏来处理信号。在下面的描述中，set是指向sigset_t变量的一个指针，nsig是信号的一个编号，mask是无符号长整数的位掩码。

sigaddset(set,nsig)和sigdelset(set,nsig)

把sigset_t变量的对应于信号nsig的位分别置为1或0。在实际中，sigaddset()简化为：

```
set->sig[(nsig - 1) / 32] |= 1UL << ((nsig - 1) % 32);
```

并且把sigdelset()简化为：

```
set->sig[(nsig - 1) / 32] &= ~(1UL << ((nsig - 1) % 32));
```

sigaddsetmask(set,mask)和sigdelsetmask(set,mask)

把sigset_t变量的对应于mask位的所有位分别设置为1或0。对应的函数简化为：

```
set->sig[0] |= mask;
```

和：

```
set->sig[0] &= ~mask;
```

sigismember(set,nsig)

返回sigset_t变量的对应于信号nsig的位值。在实际中，这个函数简化为：

```
1 & (set->sig[(nsig - 1) / 32] >> ((nsig - 1) % 32))
```

sigmask(nsig)

产生信号nsig的位索引。换句话说，如果内核需要设置、清除、或测试一个特殊的信号在sigset_t中对应的位，那么就能通过这个宏得出合适的位。

signal_pending(p)

如果*p进程描述符所表示的进程有非阻塞信号，就返回值1（真），否则返回0（假）。该函数只是检查这个进程描述符的sigpending域。

recalc_sigpending(t)

检查*t进程描述符所表示的进程是否有非阻塞的挂起信号，先查看进程的sig和blocked域，然后按如下代码适当地设置sigpending域：

```
ready  = t->signal.sig[1] &~ t->blocked.sig[1];
ready |= t->signal.sig[0] &~ t->blocked.sig[0];
t->sigpending = (ready != 0);
```

sigandsets(d,s1,s2)、sigorsets(d,s1,s2)和signandsets(d,s1,s2)

在sigset_t变量的s1和s2之间，分别执行逻辑“与”、逻辑“或”及逻辑“与非”。其结果保存在sigset_t变量的d中。

dequeue_signal(mask, info)

检查当前进程是否有非阻塞的挂起信号。如果有，返回最小编号的挂起信号并更新其数据结构以表示这个信号不再挂起。这个任务涉及清除current->signal中的相应位，更新current->sigpending的值，以及把出队信号的编号保存在*info表中。在mask参数中，被设置的每位表示一个阻塞的信号：

```
sig = 0;
if (((x = current->signal.sig[0]) & ~mask->sig[0]) != 0)
    sig = 1 + ffz(~x);
else if (((x = current->signal.sig[1]) &
          ~mask->sig[1]) != 0)
    sig = 33 + ffz(~x);
if (sig) {
    sigdelset(&current->signal, sig);
    recalc_sigpending(current);
}
return sig;
```

当前挂起的信号“与”非阻塞的信号（mask的补），如果不为0，表示有一个信号应当发送给进程。ffz()函数返回它的参数（~x）中第一个为0的位的索引，用这个值来计算要发送的最小编号的信号。

flush_signals(t)

删除发送给*t进程描述符所表示的进程的所有信号。这是通过清除t->sigpending和t->signal域，并腾空实时信号队列而完成的（参见“实时信号”一节）。

发送信号

当一个信号被发送到进程时，这个信号或来自内核或来自另一个进程。内核通过调用send_sig_info()、send_sig()、force_sig()或force_sig_info()函数发送信号。这些函数完成在前面“信号的作用”中所描述的信号处理第一阶段的任务，即更新所需的进程描述符。它们并不直接执行第二阶段接收信号的任务，这依赖于信号的类型和进程的状态，但它们可以唤醒进程并强迫进程接收信号。

send_sig_info()和send_sig()函数

send_sig_info()函数对三个参数起作用：

sig

信号编号。

info

或者是与实时信号相关的siginfo_t表的地址，或者是两个特殊的值之一：0意味着这个信号是由用户态的进程发送的，而1意味着是由内核发送的。siginfo_t数据结构具有必须传递给接收实时信号的进程的信息，如发送者进程的PID及其拥有者的UID。

t

指向目标进程描述符的指针。

send_sig_info()函数开始先检查这些参数是否正确：

```
if (sig < 0 || sig > 64)
    return -EINVAL;
```

然后这个函数检查信号是不是由用户态进程发送。当info为0或siginfo_t表的si_code域为负数或0时（正数表示由内核函数发送信号）表示信号由用户态进程发送：

```
if ((!info || ((unsigned long)info != 1 && (info->si_code <=0)))
    && ((sig != SIGCONT) || (current->session != t->session))
    && (current->euid [supscrsym] t->suid) && (current->euid [supscrsym] t->uid)
    && (current->uid [supscrsym] t->suid) && (current->uid [supscrsym] t->uid)
    && !capable(CAP_KILL))
       return -EPERM;
```

如果这个信号是由用户态的进程发送的，该函数决定是否允许这个操作。只有发送进程的拥有者具有适当的能力时，这个信号才能被传递（参见第十九章），这个信号就是SIGCONT，目标进程与发送进程处于同一个注册会话中，或两个进程属于同一个用户。

如果sig参数的值为0，该函数不发送任何信号就立即返回。因为0不是一个有效的信号编号，它只用来让发送进程检查自己是否具有向目标进程发送一个信号所必须的特权。如果目标进程处于TASK_ZOMBIE状态（通过检查它的siginfo_t表是否已被释放），该函数也返回。

```
If(!sig || !t->sig)
   Return 0;
```

对目标进程来说，一些信号类型可能使其他挂起的信号无效。因此，该函数还检查下列情况之一是否发生：

- sig是SIGKILL或SIGCONT信号。如果目标进程被停止，就把它放进TASK_RUNNING状态，以便它将能执行do_exit()函数。此外，如果目标进程有SIGSTOP，SIGTSTP，SIGTTOU，或SIGTTIN挂起信号，这些信号被删除：

```
if (t->state == TASK_STOPPED)
    wake_up_process(t);
t->exit_code = 0;
sigdelsetmask(&t->signal, (sigmask(SIGSTOP) |
        sigmask(SIGTSTP) | sigmask(SIGTTOU) |
        sigmask(SIGTTIN)));
recalc_sigpending(t);
```

- sig是SIGSTOP，SIGTSTP，SIGTTIN，或SIGTTOU信号。如果目标进程有挂起的SIGCONT信号，它被撤消：

```
sigdelset(&t->signal, SIGCONT);
recalc_sigpending(t);
```

接下来，send_sig_info()检查能否立即处理新的信号。在这种情况下，该函数也照顾到信号的接收阶段：

```
if (ignored_signal(sig, t)) {
  out:
    if (t->state == TASK_INTERRUPTIBLE && signal_pending(t))
        wake_up_process(t);
    return 0;
}
```

当在“信号的作用”一节中所提到的忽略一个信号的三个条件都被满足时，ignored_signal()函数返回1。然而，为了满足POSIX的一个要求，SIGCHLD信号被特殊地处理。POSIX区别对待对SIGCHLD信号显式地设置“忽略”操作与在适当的位置留下缺省操作(即使缺省操作是忽略信号)。为了让内核清理一个终止的子进程，并防止它变成一个僵死的进程(参见第三章的“删除进程”一节)，父进程必须显式地设置这个信号的“忽略”操作。因此，ignored_signal()的处理如下：如果这个信号被显式地忽略，ignored_signal()返回0，但是，如果缺省操作是“忽略”并且进程不改变这个缺省，那么，ignored_signal()返回1。

如果ignored_signal()返回1，那么，目标进程的siginfo_t表不必被更新。不过，如果这个进程处于TASK_INTERRUPTIELE状态，并且有其他非阻塞的挂起信号，send_sig_info()调用wake_up_process()函数唤醒这个进程。

如果ignored_signal()返回0，信号的接收阶段必须延迟，因此，send_sig_info()可能必须修改目标进程的数据结构，以让目标进程知道一个新的信号已发送给它。因为标准信号不被排队，send_sig_info()必须检查同一信号的另一个实例是否已经挂起，然后，把它的标记留在这个进程描述符适当的数据结构中：

```
if (sigismember(&t->signal, sig))
    goto out;
sigaddset(&t->signal, sig);
if (!sigismember(&t->blocked, sig))
    t->sigpending = 1;
goto out;
```

调用sigaddset()函数来设置t->signal中合适的位。除非目标进程已阻塞了sig信号，否则也设置t->sigpending标志。这个函数通过唤醒目标进程（如果必要）以普通的方式而结束。在“接收信号”一节中，我们将讨论进程所执行的操作。

send_sig()函数类似于send_sig_info()。不过，用一个priv标志代替info参数。如果信号是由内核发送的，priv标志为真，如果是由一个进程发送的，priv为假。send_sig()函数是作为send_sig_info()的一个特例而实现的：

```
return send_sig_info(sig, (void*)(priv != 0), t);
```

force_sig_info()和force_sig()函数

force_sig_info()函数由内核用来发送不能显式被忽略，或者不能被目标进程阻塞的信号。这个函数的参数与send_sig_info()的参数相同。force_sig_info()函数作用于signal_struct数据结构，而目标进程描述符t的sig域引用了signal_struct结构。

```
if (t->sig->action[sig-1].sa.sa_handler == SIG_IGN)
    t->sig->action[sig-1].sa.sa_handler = SIG_DFL;
sigdelset(&t->blocked, sig);
return send_sig_info(sig, info, t);
```

force_sig()类似于force_sig_info()。它仅用在由内核发送信号时，可以被作为force_sig_info()函数的一个特例而实现：

```
force_sig_info(sig, (void*)1L, t);
```

接收信号

我们假定内核已注意到一个信号的到来，并调用前面所介绍的函数为假定接收此信号的进程的进程描述符做准备。但万一这个进程在那一刻不在CPU上运行，内核就延迟唤醒这个进程的任务，如果必要，也延迟它接收这个信号的任务。我们现在转向内核为确保一个进程的挂起信号被处理所执行的一些操作。

正如我们在第四章的“ret_ from_intr()函数”一节中所提到的，内核在允许进程恢复用户态下的执行之前，检查是否存在非阻塞信号。每当一个中断或异常由内核例程处理完后，在ret_from_intr()中执行这个检查。

为了处理非阻塞的挂起信号，内核调用do_signal()函数，它接收两个参数：

regs

栈区的地址，当前进程用户态下寄存器的内容存放在这个栈中。

oldset

变量的地址，假设函数把阻塞信号的位掩码数组存放在这个变量中（实际上，当从ret_from_intr()调用时，这个参数为空）。

这个函数首先检查进程在用户态下运行时是否发生了中断，如果没有，就只是返回：

```
if ((regs->xcs & 3) != 3)
    return 1;
```

然而，我们将在“系统调用的重新执行”一节中看到，这并不意味着一个系统调用不能被一个信号所中断。

如果oldset参数为空，这个函数用current->blocked域的地址对它进行初始化：

```
if (!oldset)
    oldset = &current->blocked;
```

do_signal()函数的核心由重复调用dequeue_signal()的循环组成，当没有非阻塞的挂起信号时，循环才结束。dequeue_signal()的返回码存放在signr局部变量中，如果值为0，意味着所有挂起的信号已全部被处理，并且do_signal()可以结束。只要返回一个非0值，就意味着挂起的信号正等待被处理，并且do_signal()处理了当前信号后又调用了dequeue_signal()。

如果current接收进程正被其他一些进程监视，do_signal()函数调用notify_parent()和schedule()以使监视的进程意识到信号的处理。

do_signal()用要处理信号的k_sigaction数据结构的地址装载局部变量ka:

```
ka = &current->sig->action[signr-1];
```

可以执行三种操作，这取决于ka的内容，即忽略信号、执行一个缺省操作或执行一个信号处理程序。

忽略信号

当显式地忽略一个所接收的信号时，do_signal()函数只是以一个新循环的执行而正常地继续，并因此考虑另一个挂起的信号。正如前面所描述的，还存在一个例外:

```
if (ka->sa.sa_handler == SIG_IGN) {
    if (signr == SIGCHLD)
        while (sys_wait4(-1, NULL, WNOHANG, NULL) > 0)
            /* nothing */;
    continue;
}
```

如果接收的信号是SIGCHLD，就调用wait4()系统调用的sys_wait4()服务例程，以强迫这个进程读它子进程的信息、因此清理由终止的子进程所留下的内存(参见第三章的“撤消进程”一节)。

执行信号的缺省操作

如果ka->sa.sa_handler等于SIG_DFL，do_signal()必须执行信号的缺省操作。唯一的例外是当接收进程是*init*时，在这种情况下，正如“接收信号之前所执行的操作”一节中所描述的那样，这个信号被丢弃:

```
if (current->pid == 1)
    continue;
```

对其他进程，因为缺省操作依赖于信号的类型，这个函数执行一个基于 signr 的 switch 语句。

缺省操作是“忽略”的信号很容易进行处理：

```
case SIGCONT: case SIGCHLD: case SIGWINCH:
    continue;
```

缺省操作是“停止”的信号可以停止当前进程。为了做到这点，do_signal()把 current 状态设置为 TASK_STOPPED，然后调用 schedule()函数（参见第十章的“schedule()函数”一节）。do_signal()函数也把 SIGCHLD 信号发送给 current 的父进程，除非父进程已经设置了 SIGCHLD 的 SA_NOCLDSTOP 标志：

```
case SIGTSTP: case SIGTTIN: case SIGTTOU:
    if (is_orphaned_pgrp(current->pgrp))
        continue;
case SIGSTOP:
    current->state = TASK_STOPPED;
    current->exit_code = signr;
    if (!(SA_NOCLDSTOP &
        current->p_pptr->sig->action[SIGCHLD-1].sa.sa_flags))
        notify_parent(current, SIGCHLD);
    schedule();
    continue;
```

SIGSTOP 与其他信号的差异是微妙的：SIGSTOP 总是停止进程，而其他信号只停止不在“孤儿进程组”中的进程。POSIX 标准规定，只要进程组中存在一个进程，这个进程的父进程处于不同的进程组但在同一个会话中，这个进程组就不是孤儿。

缺省操作是“转储”的信号可以在这个进程的工作目录中创建一个“core”文件，这个文件列出该进程地址空间和 CPU 寄存器的全部内容。do_signal()创建了“core”文件后，就杀死这个进程。剩余 18 个信号的缺省操作是“abort”，它仅仅是杀死进程。

```
exit_code = sig_nr;
case SIGQUIT: case SIGILL: case SIGTRAP:
case SIGABRT: case SIGFPE: case SIGSEGV:
    if (current->binfmt
```

```
            && current->binfmt->core_dump
            && current->binfmt->core_dump(signr, regs))
            exit_code |= 0x80;
    default:
        sigaddset(&current->signal, signr);
        current->flags |= PF_SIGNALED;
        do_exit(exit_code);
```

当执行了一个“core”转储文件时，do_exit()函数把这个信号编号与所设置的一个标志“或”的结果作为它的输入参数。这个结果就用来决定进程的退出码。do_exit()终止当前进程，并因此从不返回（参见第十九章）。

捕获信号

如果这个信号有一个专门的处理程序，do_signal()函数必须强迫它执行。这是通过调用handle_signal()做到的：

```
handle_signal(signr, ka, &info, oldset, regs);
return 1;
```

处理了一个单独的信号以后，注意do_signal()怎样返回：直到下一次调用do_signal()时才考虑其他挂起的信号。这种方式确保了实时信号将以适当的顺序得到处理（参见后面的“实时信号”一节）。

执行一个信号处理程序是件相当复杂的任务，因为在用户态和内核态之间切换时需要谨慎地处理栈中的内容。我们将严格地解释这里所承担的任务。

信号处理程序是用户态进程所定义的函数，并包含在用户态的代码段中。handle_signal()函数运行在内核态而信号处理程序运行在用户态，这就意味着允许恢复当前进程“正常”执行之前，必须首先执行用户态的信号处理程序。此外，当内核打算恢复这个进程的正常执行时，内核态堆栈不再包含被中断程序的硬件上下文，因为每当从内核态向用户态转换时，内核态堆栈被清空。

而另外一个复杂性是因为信号处理程序可以调用系统调用，在这种情况下，执行了系统调用的服务例程以后，控制权必须返回到信号处理程序而不是到被中断程序的代码。

Linux所采用的解决方法是把保存在内核态堆栈中的硬件上下文拷贝到当前进程的

用户态堆栈中。用户态堆栈也以这样的方式被修改，当信号处理程序终止时，自动调用sigreturn()系统调用把这个硬件上下文拷贝回到内核态堆栈中，并恢复用户态堆栈中原来的内容。

图9-1说明了有关捕获一个信号的函数的执行流。一个非阻塞的信号被发送给一个进程。当中断或异常发生时，这个进程切换到内核态。正要返回到用户态前，内核执行do_signal()函数，这个函数又依次处理这个信号[通过调用handle_signal()]和建立用户态堆栈[通过调用setup_frame()]。当这个进程又切换到用户态时，因为信号处理程序的起始地址被强制放进程序计数器中，因此开始执行信号处理程序。当这个处理程序终止时，就执行setup_frame()函数放在用户态堆栈中的返回代码。这个代码调用sigreturn()系统调用，它的服务例程把正常程序的用户态堆栈硬件上下文拷贝到内核态堆栈，并把用户态堆栈恢复回到它原来的状态[通过调用restore_sigcontext()]。当这个系统调用结束时，正常进程因此能恢复自己的执行。

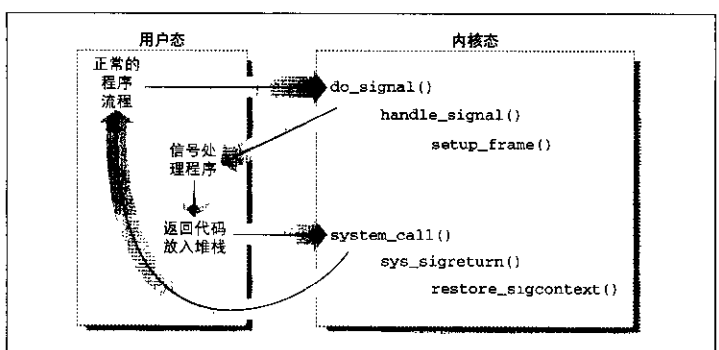


图9-1 捕获一个信号

让我们现在详细考察如何实施这种方案。

建立帧

为了适当地建立进程的用户态堆栈，handle_signal()函数或者调用setup_frame()(对没有siginfo_t表的信号)，或者调用setup_rt_frame()。

setup_frame()函数接收四个参数，它们具有下列含义：

sig
　信号编号

ka
　与这个信号相关的k_sigaction表的地址

oldset
　阻塞信号的一个位掩码数组的地址

regs
　用户态寄存器的内容已被保存在内核态堆栈区，regs就是这个内核态堆栈区的地址

setup_frame()函数把一个叫做帧（frame）的数据结构推进用户态堆栈中，这个帧含有处理信号和确保handle_signal()函数正确返回所需的信息。一个帧就是包含下列域的sigframe表（参见图9-2）：

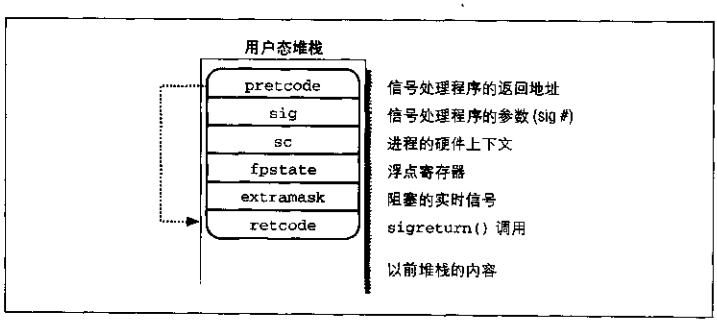


图9-2　用户态堆栈中的帧

pretcode
　信号处理函数的返回地址，它指向同一个表中的retcode域（稍后列出）。

sig
　信号编号，这是信号处理程序所需的参数。

sc

类型为sigcontext的结构，sigcontext包含正好切换到内核态前用户态进程的硬件上下文（这种信息是从current的内核态堆栈中拷贝过来的），还包含一个位数组，这个位数组指定了这个进程被阻塞的标准信号。

fpstate

类型为_fpstate的结构，_fpstate可以被用来存放用户态进程的浮点寄存器内容（参见第二章的“保存浮点寄存器”一节）。

extramask

被阻塞的实时信号的位数组。

retcode

发出sigreturn()系统调用的八字节代码，当从信号处理程序返回时，执行这段代码。

setup_frame()函数首先调用get_sigframe()计算这个帧的第一个内存单元，这个内存单元通常是在用户态堆栈中（注4），因此函数返回值：

```
(regs->esp - sizeof(struct sigframe)) & 0xfffffff8
```

因为栈朝低地址方向延伸，通过把当前栈顶的地址减去它的大小，使其结果与8的倍数对齐，就获得了这个帧的起始地址。

然后用access_ok宏对这个返回地址进行验证。如果这个地址有效，setup_frame()反复调用__put_user()填充这个帧的所有域。一旦完成了这个操作，就修改内核态堆栈的regs区，因此保证了当current恢复它在用户态的执行时，控制权将传通给信号处理程序。

```
regs->esp = (unsigned long) frame;
regs->eip = (unsigned long) ka->sa.sa_handler;
```

把内核态堆栈保存的段寄存器内容重新设置成它们的缺省值以后，setup_frame()函数结束。现在，信号处理程序所需的信息就在用户态堆栈的顶部。

注4：Linux允许进程通过调用sigaltstack()系统调用而为它们的信号处理程序指定一个预备的栈。这种特点也是X/Open标准所要求的。当一个预备的栈存在时，get_frame()函数就返回这个栈中的一个地址。我们在此不进一步讨论这种情况，因为它从概念上非常类似于标准信号的处理。

setup_rt_frame()函数与setup_frame()非常相似，但它的栈顶存放的是一个扩展的帧（保存在rt_sigframe数据结构中），这个帧也包含了与信号相关的siginfo_t表的内容。

检查信号标志

建立了用户态堆栈以后，handle_signal()函数检查与这个信号相关的标志值。

如果已接收的信号设置了SA_ONESHOT标志，它就必须被重设置到它的缺省操作，以便同一信号的进一步出现将不再触发这个信号处理函数的执行。

```
if (ka->sa.sa_flags & SA_ONESHOT)
    ka->sa.sa_handler = SIG_DFL;
```

此外，如果这个信号没有设置SA_NODEFER标志，在sigaction表中sa_mask域的这个信号必须在这个信号处理程序执行期间被阻塞：

```
if (!(ka->sa.sa_flags & SA_NODEFER)) {
    sigorsets(&current->blocked,
              &current->blocked,
              &ka->sa.sa_mask);
    sigaddset(&current->blocked,sig);
    recalc_sigpending(current);
}
```

handle_signal()然后返回到do_signal()，do_signal()也立即返回。

开始执行信号处理程序

do_signal()返回时，当前进程恢复它在用户态的执行。由于如前所述setup_frame()的准备，eip寄存器指向信号处理程序的第一条指令，而esp指向已推进用户态堆栈顶的帧的第一个内存单元。因此，信号处理程序被执行。

终止信号处理程序

当信号处理程序终止时，栈顶的返回地址指向这个帧retcode域中的代码。对于没有siginfo_t表的信号，这个代码等价于下列的汇编指令：

```
popl %eax
```

```
movl $__NR_sigreturn, %eax
int $0x80
```

因此，信号编号（即帧的sig域）被从栈中丢弃，然后调用sigreturn()系统调用。

sys_sigreturn()函数把数据结构为pt_regs的regs作为自己的参数，pt_regs包含用户态进程的硬件上下文（参见第八章的“参数传递”一节）。因此就导出这个帧在用户态堆栈内的地址：

```
frame = (struct sigframe *)(regs.esp - 8);
```

sys_sigreturn()从帧的sc域读调用信号处理函数前被阻塞的信号的位数组，并把这个位数组写到current的blocked域。结果，因为信号处理函数的执行而使被屏蔽的所有信号被解除阻塞。然后调用recalc_sigpending()函数。

此时，sys_sigreturn()函数必须把从帧的sc域获得的进程硬件上下文拷贝到内核态堆栈中，然后调用restore_sigcontext()函数从用户态堆栈中删除这个帧。

对于具有siginfo_t表的信号，其实现机制非常相似。扩展帧的retcode域中的返回代码调用rt_sigreturn()系统调用，其相应的sys_rt_sigreturn()服务例程把来自扩展帧的进程硬件上下文拷贝到内核态堆栈，并通过从用户态堆栈删除扩展帧以恢复用户态堆栈原来的内容。

系统调用的重新执行

在一些情况下，内核并不能立即满足系统调用发出的请求，在这种情况发生时，把发布系统调用的进程置为TASK_INTERRUPTIBLE或TASK_UNINTERRUPTIBLE状态。

如果进程处于TASK_INTERRUPTIBLE状态，并且另一个进程向这个进程发送了一个信号，那么，内核不完成系统调用而把该进程置成TASK_RUNNING状态（参看第四章的“从中断和异常返回”一节）。当这种情况发生时，系统调用服务例程没有执行完但返回EINTR、ERESTARTNOHAND、ERESTARTSYS或ERESTARTNOINTR错误码。当切换回用户态时进程就接收这个信号。

实际上，这种情况下用户态进程获得的唯一错误码是EINTR，这个错误码表示系统调用还没有执行完。（应用程序的编写者可以测试这个错误码并决定是否重新发布系

统调用。)内核内部使用剩余的错误码来指定信号处理程序结束以后是否重新调用这个系统调用。

表9-4列出与未完成的系统调用相关的错误码及这些错误码对信号三种可能的操作产生的影响。在表项中出现的几个术语的含义如下:

Terminate

不会自动地重新执行系统调用。在 `int $0x80` 指令紧接着的那条指令处,进程将恢复它在用户态的执行,这时 `eax` 寄存器包含的值为 `-EINTR`。

Reexecute

内核强迫用户态进程用系统调用号再装载 `eax` 寄存器,并重新执行 `int $0x80` 指令。进程意识不到这种重新执行,这个错误代码也不传递给进程。

Depends

只有接收到的信号设置了 `SA_RESTART` 标志,才重新执行系统调用;否则,系统调用以 `-EINTR` 错误码结束。

表 9-4 系统调用的重新执行

信号	错误码及其对系统调用执行的影响			
操作	EINTR	ERESTARTSYS	ERESTARTNOHAND	ERESTARTNOINTR
Default	Terminate	Reexecute	Reexecute	Reexecute
Ignore	Terminate	Reexecute	Reexecute	Reexecute
Catch	Terminate	Depends	Terminate	Reexecute

当内核接收到一个信号时,就必须在打算重新执行一个系统调用前确定进程确实发布了这个系统调用。这就是 `regs` 硬件上下文的 `orig_eax` 域起重要作用之处。让我们回顾一下中断或异常处理程序开始时是如何初始化这个域的。

中断

这个域包含与中断相关的IRQ号减去256(参看第四章的"为中断处理程序保存寄存器的值"一节)。

0x80异常

这个域包含系统调用号(参看第八章的"system_call()函数"一节)。

其他异常

这个域包含的值为-1（参看第四章的“为中断处理程序保存寄存器的值”一节）。

因此，orig_eax域中的非负数表示这个信号已经唤醒了在一个系统调用上睡眠的TASK_INTERRUPTIBLE进程。相应的服务例程只好承认这个系统调用曾被中断，并返回前面提到的错误码。

如果信号被显式地忽略，或者它的缺省操作已被执行，do_signal()分析系统调用的错误码并如表9-4中所指定的那样决定是否重新自动执行未完成的系统调用。如果必须重新开始执行系统调用，那么do_signal()修改regs硬件上下文，以便在进程返回到用户态时，eip指向int $0x80指令，并且eax包含这个系统调用号：

```
if (regs->orig_eax >= 0) {
    if (regs->eax == -ERESTARTNOHAND ||
        regs->eax == -ERESTARTSYS ||
        regs->eax == -ERESTARTNOINTR) {
        regs->eax = regs->orig_eax;
        regs->eip -= 2;
    }
}
```

regs->eax域已被（参见第八章的“system_call()函数”一节）系统调用服务例程的返回码填充。

如果信号已被捕获，那么handle_signal()分析错误码，也可能分析sigaction表的SA_RESTART标志来决定是否必须重新执行未完成的系统调用。

```
if (regs->orig_eax >= 0) {
    switch (regs->eax) {
        case -ERESTARTNOHAND:
            regs->eax = -EINTR;
            break;
        case -ERESTARTSYS:
            if (!(ka->sa.sa_flags & SA_RESTART)) {
                regs->eax = -EINTR;
                break;
            }
        /* fallthrough */
        case -ERESTARTNOINTR:
```

```
                regs->eax = regs->orig_eax;
                regs->eip -= 2;
        }
    }
```

如果系统调用必须被重新开始执行，handle_signal()与do_signal()完全一样继续执行；否则，它向用户态进程返回一个错误码-EINTR。

实时信号

POSIX标准引入一种新的信号类型来表示实时信号，相应的信号编号从32到63。实时信号与标准信号的主要差别在于同一种实时信号可以排队。这就确保了发送的多个信号将全部被接收。尽管Linux内核没有利用实时信号，但是它通过几个特殊的系统调用完全支持POSIX标准（参见后面的“实时信号的系统调用”一节）。

实时信号的队列是作为一个链表而实现的，链表的元素是signal_queue:

```
struct signal_queue {
    struct signal_queue *next;
    siginfo_t info;
};
```

类型为siginfo_t的info表已在“send_sig_info()和send_sig()函数”一节中给出解释。next域指向链表中的下一个元素。

每个进程描述符有两个特殊的域：sigqueue指向接收实时信号队列的第一个元素，而sigqueue_tail指向这个队列最后一个元素的next域。

当发送一个信号时，send_sig_info()函数检查它的这个信号的编号是否大于31，如果是，把这个信号插入目标进程的实时信号队列中。

与此相似，当接收一个信号时，dequeue_signal()检查挂起信号的信号编号是否大于31，如果是，从这个队列元素中提取相应的接收信号。如果队列不包含同一类型的其他信号，这个函数也清除current->signal的相应位。

与信号处理相关的系统调用

正如本章已提到的，用户态运行的进程可以发送和接收信号。这意味着必须定义一组系统调用来完成这些操作。遗憾的是，由于历史的原因，已经存在几个具有相同功能的非兼容性系统调用。为了确保与原来的Unix版本完全兼容，Linux既支持原来的系统调用也引入了符合POSIX标准的新系统调用。我们将描述其中一些最重要的POSIX系统调用。

kill()系统调用

一般用kill(pid,sig)系统调用发送信号，其相应的服务例程是sys_kill()函数。整数参数pid的几个含义取决于它的值：

pid > 0

把sig信号发送到进程的PID等于pid的进程。

pid = 0

把sig信号发送到与调用进程同组的所有进程。

pid = -1

把信号发送到所有进程，除了*swapper*（PID 0）、*init*（PID 1）和current。

pid < -1

把信号发送到进程组 *-pid* 中的所有进程。

sys_kill()函数调用kill_something_info()函数。后者还依次调用send_sig_info()向一个单独的进程发送信号，或者调用kill_pg_info()扫描所有的进程并为目标进程组中的每个进程又调用send_sig_info()。

System V和BSD Unix各种版本也有killpg()系统调用，这个系统调用能显式地向一组进程发送一个信号。在Linux中，这个函数是作为一个库函数，利用了kill()系统调用而实现的。

改变信号的操作

sigaction(sig,act,oact)系统调用允许用户为信号指定一个操作。当然，如果没有自定义的信号操作，那么内核执行与这个接收信号相关的缺省操作。

相应的sys_sigaction()服务例程作用于两个参数：sig信号编号和表示新操作的类型为sigaction的act表。第三个可选的输出参数oact可以用来获得与这个信号相关的以前的操作。

这个函数首先检查act地址的有效性。然后用*act相应的域填充类型为k_sigaction的在new_ka局部变量的sa_handler、sa_flags和sa_mask域：

```
__get_user(new_ka.sa.sa_handler, &act->sa_handler);
__get_user(new_ka.sa.sa_flags, &act->sa_flags);
__get_user(mask, &act->sa_mask);
new_ka.sa.sa_mask.sig[0] = mask;
new_ka.sa.sa_mask.sig[1] = 0
```

这个函数还调用do_sigaction()把新的new_ka表拷贝到current->sig->action的在sig-1位置的表项中：

```
k = &current->sig->action[sig-1];
if (act) {
    *k = *act;
    sigdelsetmask(&k->sa.sa_mask, sigmask(SIGKILL)
                                | sigmask(SIGSTOP));
    if (k->sa.sa_handler == SIG_IGN
        || (k->sa.sa_handler == SIG_DFL
        && (sig == SIGCONT ||
            sig == SIGCHLD ||
            sig == SIGWINCH))) {
        sigdelset(&current->signal, sig);
        recalc_sigpending(current);
    }
}
```

POSIX标准规定，当缺省操作是“ignore”时，把一个信号操作设置成SIG_IGN或SIG_DFL将引起同类型的的任一挂起信号被丢弃。此外还要注意，对信号处理程序来说，不论请求屏蔽的信号是什么，SIGKILL和SIGSTOP从不被屏蔽。

如果oact参数不为空，就把sigaction表以前的内容拷贝到由oact参数指定的进程地址空间：

```
if (oact) {
    __put_user(old_ka.sa.sa_handler, &oact->sa_handler);
    __put_user(old_ka.sa.sa_flags, &oact->sa_flags);
```

```
        __put_user(old_ka.sa.sa_mask.sig[0], &oact->sa_mask);
    }
```

为了与BSD Unix各种版本保持兼容，Linux提供了signal()系统调用，它仍由编程人员广泛地使用着。其相应的服务例程sys_signal()仅仅调用了do_sigaction()：

```
    new_sa.sa.sa_handler = handler;
    new_sa.sa.sa_flags = SA_ONESHOT | SA_NOMASK;
    ret = do_sigaction(sig, &new_sa, &old_sa);
    return ret ? ret : (unsigned long)old_sa.sa.sa_handler;
```

检查挂起的阻塞信号

sigpending()系统调用允许进程检查挂起的阻塞信号的设置情况，也就是说，一个信号被阻塞时已产生的那些信号。这个系统调用只取标准信号。

相应的服务例程sys_sigpending()只作用于一个参数set，即用户的一个变量地址，位数组必须拷贝到这个变量中：

```
    pending = current->blocked.sig[0] & current->signal.sig[0];
    if (copy_to_user(set, &pending, sizeof(*set)))
        return -EFAULT;
    return 0;
```

修改被阻塞信号的集合

sigprocmask()系统调用允许进程修改被阻塞信号的集合。与sigpending()类似，这个系统调用只应用于标准的信号。

相应的sys_sigprocmask()服务例程作用于三个参数：

oset

在进程地址空间的一个指针，指向存放以前位掩码的一个位数组。

set

在进程地址空间的一个指针，指向包含新位掩码的位数组。

how

一个标志，可以有下列之一值：

SIG_BLOCK

*set位掩码数组指定的信号是必须加到被阻塞信号的位掩码数组中的信号。

SIG_UNBLOCK

*set位掩码数组指定的信号是必须从被阻塞信号的位掩码数组中删除的信号。

SIG_SETMASK

*set位掩码数组指定被阻塞信号新的位掩码数组。

sys_sigprocmask()调用copy_from_user()把set参数所指向的值拷贝到局部变量new_set中，并把current的被阻塞的标准信号位掩码数组拷贝到old_set局部变量中。然后根据how标志来指定这两个变量的值：

```
if (copy_from_user(&new_set, set, sizeof(*set)))
    return -EFAULT;
new_set &= ~(sigmask(SIGKILL)|sigmask(SIGSTOP));
old_set = current->blocked.sig[0];
if (how == SIG_BLOCK)
    sigaddsetmask(&current->blocked, new_set);
else if (how == SIG_UNBLOCK)
    sigdelsetmask(&current->blocked, new_set);
else if (how == SIG_SETMASK)
    current->blocked.sig[0] = new_set;
else
    return -EINVAL;
recalc_sigpending(current);
if (oset) {
    if (copy_to_user(oset, &old_set, sizeof(*oset)))
        return -EFAULT;
}
return 0;
```

挂起进程

sigsuspend()系统调用阻塞mask参数指向的位掩码数组所指定的标准信号以后，把进程置成TASK_INTERRUPTIBLE状态。只有当一个非忽略、非阻塞的信号发送到这个进程以后，进程才被唤醒。

相应的sys_sigsuspend()服务例程执行下列这些语句:

```
mask &= ~(sigmask(SIGKILL) | sigmask(SIGSTOP));
saveset = current->blocked;
current->blocked.sig[0] = mask;
current->blocked.sig[1] = 0;
recalc_sigpending(current);
regs->eax = -EINTR;
while (1) {
    current->state = TASK_INTERRUPTIBLE;
    schedule();
    if (do_signal(regs, &saveset))
    return -EINTR;
}
```

schedule()函数选择另一个进程运行。当发布sigsuspend()系统调用的进程又开始执行时,sys_sigsuspend()调用do_signal()系统调用来接收唤醒这个进程的信号。如果do_signal()的返回值为1,则忽略这个信号,这个系统调用返回-EINTR错误码后终止。

sigsuspend()系统调用可能看起来多余,因为sigprocmask()和sleep()的组合执行显然能产生同样的效果。但这并不正确:进程的交错执行,你必须意识到,调用一个系统调用执行操作A,紧接着又调用另一个系统调用执行操作B,并不等于调用一个单独的系统调用执行操作A,然后执行操作B。

在实际情况中,sigprocmask()可以在调用sleep()之前解除对要接收信号的阻塞。如果这种情况发生,进程就可以一直停留在TASK_INTERRUPTIBLE状态,等待已被接收的信号。另一方面,在解除阻塞之后、schedule()调用之前,因为其他进程在这个时间间隔内无法抓到CPU,因此,sigsuspend()系统调用不允许信号被发送。

实时信号的系统调用

因为前面所提到的系统调用只应用到标准的信号,因此,必须引入另外的系统调用来允许用户态进程处理实时信号。

实时信号的几个系统调用(rt_sigaction()、rt_sigpending()、rt_sigprocmask()及rt_sigsuspend())与前面描述的类似,因此不再进一步讨论。

引入另外的两个系统调用来处理实时信号的队列:

`rt_sigqueueinfo()`

发送一个实时信号以便把它加入到目标进程的实时信号队列。

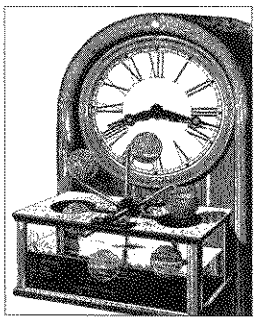
`rt_sigtimedwait()`

类似于 `rt_sigsuspend()`，但是进程仍被挂起一个固定的时间间隔。

因为这些系统调用与标准信号使用的系统调用非常相似，我们不再讨论它们。

对 Linux 2.4 的展望

信号在 Linux 2.2 和 Linux 2.4 中完全一样。



第十章

进程调度

Linux与任何分时系统一样，通过一个进程到另一个进程的快速切换，多个进程的并发执行达到了不可思议的效果。进程切换本身已在第三章中讨论过，本章讨论进程调度，主要关心什么时候进行进程切换及选择哪一个进程来运行。

本章由三部分组成："调度策略"一节从理论上介绍Linux进行进程调度所做的选择，"调度算法"一节讨论实现调度和相应的算法所采用的数据结构，最后，"与调度相关的系统调用"一节描述了影响进程调度的系统调用。

调度策略

传统Unix操作系统的调度算法必须实现几个互相冲突的目标：进程响应时间尽可能快，后台作业的吞吐量尽可能高，进程的饥饿现象尽可能避免，低优先级和高优先级进程的需要尽可能调和等等。决定什么时候以怎样的方式选择一个新进程运行的这组规则就是所谓的调度策略（scheduling policy）。

Linux的调度是基于第五章已介绍的分时技术（time-sharing）。允许多个进程"并发"运行就意味着CPU的时间被粗略地分成"片"，给每个可运行进程分配一片（注1）。当然，单处理器在任何给定的时刻只能运行一个进程。当一个并发执行的

注1：运行CPU时，调度算法不会选择调用已被停止和已被挂起的进程。

进程其时间片或时限（quantum）到期时还没有终止，进程切换就可以发生。分时依赖于定时中断，因此，对进程是透明的。为保证CPU分时，不需要在程序中插入额外的代码。

调度策略也是基于依照优先级排队的进程。有时用复杂的算法求出进程当前的优先级，但最后的结果是相同的：每个进程都与一个值相关联，这个值表示把进程如何适当地分配给CPU。

在Linux中，进程的优先级是动态的。调度程序跟踪进程做了些什么，并周期性地调整它们的优先级。在这种方式下，在较长的时间间隔内没有使用CPU的进程，通过动态地增加它们的优先级来提升它们。相应地，对于已经在CPU上运行了较长时间的进程，通过减少它们的优先级来处罚它们。

当谈及有关调度问题时，传统上把进程分类为“I/O范围（I/O-bound）”或“CPU范围（CPU-bound）”。前者频繁地使用I/O设备，并花费很多时间等待I/O操作的完成；而后者是需要大量CPU时间的数值计算应用程序。

另一种分类法把进程区分为三类：

交互式进程（Interactive process）

这些进程经常与用户进行交互，因此，要花很多时间等待击键和鼠标操作。当输入被接受时，必须很快地唤醒进程，否则，用户将发现系统反应迟钝。典型的情况是，平均延迟必须低于50到150ms。这样的延迟变化也必须进行限制，否则，用户将发现系统是不稳定的。典型的交互式程序是命令shell、文本编辑程序及图形应用程序。

批处理进程（Batch process）

这些进程不必与用户交互，因此，它们经常在后台运行。因为这样的进程不必被很快地响应，因此，它们常受到调度程序的处罚。典型的批处理进程是程序设计语言的编译程序、数据库搜索引擎及科学计算。

实时进程（Real-time process）

这些进程有很强的调度需要。这样的进程决不会被低优先级的进程阻塞。它们应该有一个短的响应时间，更重要的是，这个响应时间应该有很小的变化。典型的实时程序有视频和音频应用程序、机器人控制程序及从物理传感器上收集数据的程序。

我们刚刚提到的两种分类法有点相互独立。例如，一个批处理进程可能既是I/O范围（如数据库服务器）也是CPU范围（如图象着色程序）。在Linux中，调度算法可以明确地确认所有实时程序的身份，但没有办法区分交互式程序和批处理程序。为了为交互式应用程序提供好的响应时间，Linux（与所有的Unix内核类似）隐含地支持I/O范围的进程胜过CPU范围的进程。

程序员可以通过表10-1所列的系统调用改变调度参数。更详细的内容将在“与调度相关的系统调用”一节中给出。

表 10-1 与调度相关的系统调用

系统调用	描述
nice()	改变一个普通进程的优先级
getpriority()	获得一组普通进程的最大优先级
setpriority()	设置一组普通进程的优先级
sched_getscheduler()	获得一个进程的调度策略
sched_setscheduler()	设置一个进程的调度策略和优先级
sched_getparam()	获得一个进程的调度优先级
sched_setparam()	设置一个进程的优先级
sched_yield()	没有阻塞地自愿放弃处理器
sched_get_ priority_min()	为一种策略获得最小优先级
sched_get_ priority_max()	为一种策略获得最大优先级
sched_rr_get_interval()	为循环轮转策略获得的时间片值

在表中所显示的大部分系统调用适用于实时进程，因此，允许用户开发实时应用。不过，Linux不支持很多过分要求的实时应用，因为Linux的内核是非抢占式的（参见后面的“调度算法的性能”一节）。

进程的抢占

如第一章所述，Linux的进程是抢占式的。如果进程进入TASK_RUNNING状态，内核检查它的动态优先级是否大于当前正运行进程的优先级。如果是，current的执行被中断，并调用调度程序选择另一个进程运行（通常是刚刚变为可运行的进程）。

当然，进程在它的时间片到期时也可以被抢占。如第五章中的“CPU的分时”一节中所提到的，此时设置当前进程的need_resched域，以便定时中断处理程序终止时调用调度程序。

例如，让我们考虑一种情况，在这种情况中，只有一个正文编辑程序和一个编译程序正在被执行。正文编辑程序是一个交互式程序，因此，它的优先级高于编译程序。不过，因为编辑程序交替于暂停思考与数据输入之间，因此，它经常被挂起；此外，两次击键之间的平均延迟相对较长。然而，只要用户一按键，中断就发生，内核唤醒正文编辑进程。内核也确定编辑进程的动态优先级确实是高于current的优先级（当前正运行的进程，即编译进程），因此，设置编辑进程的need_resched域，如此来强迫内核处理完中断叫激活调度程序。调度程序选择编辑进程并执行任务切换；结果，编辑进程很快恢复执行，并把用户敲的字符回显在屏幕上。当处理完字符时，正文编辑进程自己挂起等待下一次击键，编译进程恢复执行。

注意被抢占的进程并没有被挂起，因为它还处于TASK_RUNNING状态，只不过不再使用CPU。

一些实时操作系统具有抢占式内核的特点，这就意味着正如在用户态一样，任何一条指令执行之后，在内核态运行的进程可能被中断。Linux内核不是抢占式的，这意味着进程只有在用户态运行时才能被抢占。非抢占式内核的设计非常简单，因为内核数据结构的大部分同步问题很容易被避免（参见第十一章中的“内核态进程的非抢占性”一节）。

一个时间片必须维持多长？

时间片的长短对系统性能是很关键的：它既不能太长也不能太短。

如果时间片太短，由任务切换引起的系统额外开销就变得非常高。例如，假定任务切换需要10ms，如果时间片也是10ms，那么，CPU至少把50%的时间花费在任务切换上（注2）。

如果时间片太长，进程看起来就不再是并发执行。例如，让我们假定把时间片设置

注2：实际上，事情比这里所介绍的可能更糟糕。例如，在进程的时间片中还要计算进程切换所需的时间，那么所有的CPU时间会花费在进程切换，就没有进程能够执行完。

为5秒，那么，每个可运行进程运行大约5秒，但是暂停的时间更长（典型的是5秒乘以可运行进程的个数）。

通常认为长的时间片降低交互式应用程序的响应时间，但这往往是错误的。正如本章前面“进程的抢占”一节中所描述的那样，交互式进程有相对较高的优先级，因此，不管时间片是多长，它们很快地抢占批处理进程。

在一些情况下，一个太长的时间片会降低系统的响应能力。例如，假定两个用户在各自的shell提示符下输入两条命令，一条是CPU范围的，而另一条是交互式应用。两个shell都创建一个新进程，并把用户命令的执行委托给新进程。此外，又假定这样的新进程最初有相同的优先级（Linux预先并不知道执行进程是批处理的还是交互式的）。现在，如果调度程序选择CPU范围的进程执行，那么，另一个进程开始执行前就可能要等待一个时间片。因此，如果这样的时间片较长，那么，看起来系统可能对用户的请求反应迟钝。

时间片大小的选择总是一种折衷。Linux采取单凭经验的方法，即选择尽可能长的一个时间片，同时能保持良好的响应时间。

调度算法

Linux调度算法把CPU的时间划分为时期（epoch）。在一个单独的时期内，每个进程有一个指定的时间片，时间片持续时间从这个时期的开始计算。一般情况下，不同的进程有不同大小的时间片。时间片的值是在一个时期内，分配给进程的最大CPU时间部分。当一个进程用完它的时间片时，这个进程被抢占，并用另一个可运行进程代替它。当然，在同一时期内，一个进程可以几次被调度程序选中（只要它的时间片还没用完），例如，如果进程挂起自己，等待I/O，那么，它还剩余一些时间片，并可以在同一时期内再度被选中。当所有的可运行进程都用完它们的时间片时，一个时期才结束；在这种情况下，调度程序的算法重新计算所有进程的时间片，然后，一个新的时期开始。

每个进程有一个基本的时间片（base time quantum）：如果进程在前一个时期内已用完它的时间片，那么这个时间片值就是调度程序赋给进程的基本时间片。用户可以通过`nice()`和`setpriority()`系统调用来改变进程的基本时间片（参见本章后面“与调度相关的系统调用”一节）。新进程总是继承父进程的基本时间片。

INIT_TASK宏把进程0（*swapper*)的基本时间片设置为DEF_PRIORITY。宏的定义如下：

```
#define DEF_PRIORITY (20*HZ/100)
```

因为表示定时中断频率的HZ在IBM PC中被设置为100（参见第五章中的“可编程间隔定时器”一节），因此，DEF_PRIORITY的值是20个节拍，即大约210ms。

用户很少改变他们进程的基本时间片，因此，DEF_PRIORITY也表示系统中大多数进程的基本时间片。

为了选择一个进程运行，Linux调度程序必须考虑每个进程的优先级。实际上，有两种优先级：

静态优先级（Static priority）

这种优先级由用户赋给实时进程，范围从1到99，调度程序从不改变它。

动态优先级（Dynamic prority）

这种优先级只应用于普通进程。实质上它是基本时间片[由此也叫进程的基本优先级（base priority)] 与当前时期内的剩余时间片之和。

当然，实时进程的静态优先级总是高于普通进程的动态优先级，只有当TASK_RUNNING状态没有实时进程时，调度程序才开始运行普通进程。

调度程序使用的数据结构

我们回忆一下第三章的“进程描述符”一节，进程链表把所有进程的描述符链接在一起，而运行队列链表把所有可运行状态（即TASK_RUNNING状态）的进程描述符链接在一起。在这两种情况下，init_task进程描述符起链表头的作用。

每个进程描述符包含与调度有关的几个域：

need_resched

由ret_from_intr()检查的一个标志，决定是否调用schedule()函数（参见第四章中的“ret_from_intr()函数”一节）。

policy

调度的类型，允许的取值是：

SCHED_FIFO

先入先出的实时进程。当调度程序把CPU分配给一个进程时，该进程的描述符还留在运行队列链表的当前位置。如果没有其他更高优先级的实时进程是可运行的，这个进程就可以随心所欲地使用CPU，即使具有相同优先级的其他实时进程是可运行的。

SCHED_RR

循环轮转的实时进程。当调度程序把CPU分配给一个进程时，把这个进程的描述符就放在运行队列的末尾。这种策略确保了把CPU时间公平地分配给具有相同优先级的所有SCHED_RR实时进程。

SCHED_OTHER

普通的分时进程。

policy域也对SCHED_YIELD二进制标志进行编码。当进程调用sched_yield()系统调用（不需要开始I/O操作或睡眠就自愿放弃处理器的一种方式）时，这个标志被设置。调度程序就把这个进程描述符放在运行队列链表的末尾（参见“与调度相关的系统调用”一节）。

每当内核在执行一个较长而非紧急的任务，但又希望给其他进程一个运行的机会时，内核也设置SCHED_YIELD标志，并调用schedule()函数。

rt_priority

实时进程的静态优先级。普通进程不用这个域。

priority

进程的基本时间片（或基本优先级）。

counter

进程的时间片用完之前剩余的CPU时间的节拍。当一个新的时期开始时，这个域包含进程的时间片。回想一下，update_process_times()函数在每个节拍把当前进程的counter域减1。

当一个新的进程被创建时，do_fork()以下列方式设置current（父）和p（子）进程的counter域。

```
current->counter >>= 1;
p->counter = current->counter;
```

换句话说，父进程剩余的节拍数被分成两部分，一部分给父进程，另一部分给子进程。这样做是为了防止用户使用下列方法无限制地使用CPU的时间：父进程创建了运行相同代码的子进程，然后杀死自己。通过适当地调节创建速度，子进程就可以在父进程的时间片用完之前总是获得一个新的时间片。这种编程技巧现在不起作用，因为内核不奖赏fork。类似地，用户不能通过在shell下启动很多后台进程或在图形化桌面上打开很多窗口而贪婪地不公平地共享处理器。更一般地说，一个进程不能通过创建很多后代而占有资源。

注意，priority和counter域在不同种类的进程中起的作用不同。对于普通进程，用它们既实现分时也计算进程的动态优先级。对于SCHED_RR实时进程，只用它们实现分时。最后，对于SCHED_FIFO实时进程，根本就不用它们，因为调度算法认为这种时间片是无限制的。

schedule()函数

schedule()实现调度程序。它的目的是在运行队列链表中找到一个进程，然后把CPU分配给它。几个内核例程以直接或松散的方式调用它。

直接调用（direct invocation）

因为current进程所需的资源无法得到满足而必须被立即阻塞时，就直接调用调度程序。在这种情况下，要阻塞current的内核例程按如下方式进行：

1. 把current插入到合适的等待队列中
2. 把current当前的状态改变为TASK_INTERRUPTIBLE或TASK_UNINTERRUPTIBLE
3. 调用schedule()
4. 检查那个资源是否可用；如果不，转到第2步
5. 一旦那个资源成为可用的，把current从等待队列中删除

正如所看到的，内核例程反复地检查进程所需的资源是否可用，如果不可用，就通过调用schedule()把CPU让给某一其他进程。随后，当调度程序又把CPU返还给这个进程时，又得检查资源的可用性。

你可能已注意到，这些步骤类似于在第三章的“等待队列”一节中所描述的

sleep_on()和interruptible_sleep_on()函数所实现的那些步骤。不过，我们在这里讨论的这个函数，只要进程一被唤醒就立即从等待队列中删除它。

执行长的、重复的任务的很多设备驱动程序也直接调用调度程序。在每次反复循环中，驱动程序都检查need_resched域的值，如果必要，调用schedule()主动放弃CPU。

松散调用（lazy invocation）

也可以通过把current的need_resched域设置为1，以松散的方式调用调度程序。因为对这个域值的检查总是在用户态的进程恢复执行之前进行（参见第四章中的"从中断和异常返回"一节），因此，在将来某一特定的时间，schedule()被明确地调用。

调度程序的松散调用在下列情况下被执行：

- 当current用完它的CPU时间片，这是通过update_process_times()函数进行的。
- 当一个进程被唤醒，并且它的优先级高于current进程的优先级。这个任务由reschedule_idle()函数执行，而它是由wake_up_process()函数调用的（参见第三章中的"标识一个进程"一节）：

```
if (goodness(current, p) > goodness(current, current))
    current->need_resched = 1;
```

（在后面的"如何衡量一个可运行进程？"一节中将描述goodness()函数）。

- 当发出一个sched_setscheduler()或sched_ yield()系统调用时（参见本章后面"与调度相关的系统调用"一节）。

schedule()所执行的操作

在实际调度一个进程之前，先运行其他内核控制路径在各种队列中剩余的函数，然后schedule()函数才开始运行。schedule()调用run_task_queue()执行tq_scheduler任务队列上的函数。当一个函数必须延迟执行直到下一次schedule()调用时，Linux就把这个函数放到tq_scheduler任务队列中：

```
run_task_queue(&tq_scheduler);
```

schedule()还执行所有活动的、未屏蔽的下半部分。这些下半部分通常执行由设备驱动程序请求的任务（参见第四章中的“下半部分”一节）：

```
if (bh_active & bh_mask)
    do_bottom_half();
```

现在，到了实际的调度，由此可能发生一个进程切换。

current的值被保存在prev局部变量中，并把prev的need_resched域置0。schedule()函数的关键操作是设置另一个局部变量next，以使next代替prev而指向被选中的进程的描述符。

首先，进行一个检查以确定prev是不是用完时间片的循环轮转实时进程（policy域被设置为SCHED_RR）。如果是，schedule()给prev分配一个新的时间片，并把它放到运行队列链表的末尾：

```
if (!prev->counter && prev->policy == SCHED_RR) {
    prev->counter = prev->priority;
    move_last_runqueue(prev);
}
```

现在，schedule()检查prev的状态。如果prev有非阻塞的挂起信号且它的状态为TASK_INTERRUPTIBLE，就以如下方式唤醒这个进程。这种操作与把处理器分配给prev不同，这仅仅是给prev一个被选择执行的机会。

```
if (prev->state == TASK_INTERRUPTIBLE &&
    signal_pending(prev))
    prev->state = TASK_RUNNING;
```

如果prev不是在TASK_RUNNING状态，prev进程本身就直接调用schedule()，因为prev必须等待某一外部资源。因此，必须从运行队列链表中删除prev：

```
if (prev->state != TASK_RUNNING)
    del_from_runqueue(prev);
```

接下来，schedule()必须选择在下一个时间片内要执行的进程。于是，函数扫描运行队列链表。这是从init_task［进程0（*swapper*）的描述符］的next_run域所指向的进程开始的，目的是把最高优先级进程的描述符指针保存在next中。为了做到这点，把next初始化为要检查的第一个可运行进程，把c初始化为它的“goodness”（参见后面“如何衡量一个可运行进程？”一节）。

```
if (prev->state == TASK_RUNNING) {
    next = prev;
    if (prev->policy & SCHED_YIELD) {
        prev->policy &= ~SCHED_YIELD;
        c = 0;
    } else
        c = goodness(prev, prev);
} else {
    c = -1000;
    next = &init_task;
}
```

如果prev->policy的SCHED_YIELD标志被设置，prev就发出sched_yield()系统调用自愿放弃CPU。在这种情况下，这个函数把0赋给它的goodness。

现在，schedule()在可运行进程队列上重复调用goodness()函数以确定最佳候选者：

```
p = init_task.next_run;
while (p != &init_task) {
    weight = goodness(prev, p);
    if (weight > c) {
        c = weight;
        next = p;
    }
    p = p->next_run;
}
```

通过while循环在运行队列上选择第一个具有最大权值的进程。如果前一个进程是可运行的，那么，对具有相同优先级的其他可运行进程而言，这个进程就是首选的。

注意，如果运行队列链表为空（除了*swapper*，没有可运行进程存在），就不进入循环，并且next指向init_task。此外，如果运行队列链表中所有进程的优先级都小于或等于prev的优先级，那么，不发生进程切换，原来的进程将继续执行。

在退出循环时还必须做进一步的检查，以确定c是否为0。这只发生在运行队列中的所有进程都用完了它们的时间片，即它们所有的counter域为0时。当这种情况发生时，一个新的时期开始，因此，schedule()给所有现有的进程（不仅仅指TASK_RUNNING进程）分配一个新的时间片，其大小为priority的值加counter值的一半：

```
    if (!c) {
        for_each_task(p)
            p->counter = (p->counter >> 1) + p->priority;
    }
```

在这种方式下，挂起和停止的进程让它们的动态优先级周期性地增加。如前所述，对挂起和停止进程counter值增加的基本原理是对I/O范围的进程给予优先选择。然而，即使增加无数次后，counter的值永远也不会变得大于priority值的两倍(注3)。

现在到了schedule()的结束部分。如果已选择了一个进程(除了prev)，进程切换必须发生。不过，执行切换前，kstat的context_swtch域增加1以更新由内核维护的统计数据：

```
    if (prev != next) {
        kstat.context_swtch++;
        switch_to(prev,next);
    }
    return;
```

注意，从schedule()退出的return语句并不是由next进程立即执行，而是稍后一点在调度程序又选择prev执行时由prev进程执行。

如何衡量一个可运行进程？

调度算法的核心是在运行队列链表的所有进程中确定最佳候选者。这就是goodness()函数所做的事情。它接受prev(前一个运行进程的描述符指针)和p(要评估的进程描述符指针)作为输入参数。由goodness()返回的c整数值反映了p的“值得运行的程度(goodness)”，并有如下含义：

c = -1000

永远不必选中p。当运行队列链表只包含init_task时返回这个值。

注3：假设priority和counter的值都等于P；那么几何级数 $p \times (1+\frac{1}{2}+\frac{1}{4}+\frac{1}{8}+\ldots)$ 收敛至 $2 \times p$。

c = 0

p用完了它的时间片。除非p是运行队列链表中的第一个进程，并且所有的可运行进程也用完了它们的时间片，否则，不会选中p运行。

0 < c < 1000

p是没有用完时间片的普通进程。c较大时表示goodness的更高级。

c >= 1000

p是实时进程。c较大时表示goodness的更高级。

goodness()函数等价于：

```
if (p->policy != SCHED_OTHER)
        return 1000 + p->rt_priority;
if (p->counter == 0)
        return 0;
if (p->mm == prev->mm)
        return p->counter + p->priority + 1;
return p->counter + p->priority;
```

如果是实时进程，把它的goodness置为至少1000。如果是用完时间片的普通进程，把它的goodness置为0；否则，把它的goodness置为p->counter + p->priority。

如果p与prev共享地址空间（即它们进程描述符的mm域指向同一内存描述符），给p一个小小的奖励。这种奖励的基本原理是，如果p正好在prev之后运行，它们将使用同一页表，也即同一内存，这样一些有价值的数据可以一直放在硬件的高速缓存中。

Linux/SMP 调度程序

Linux为了支持对称多处理器（SMP）体系结构，必须对Linux的调度程序稍做修改。实际上，每个处理器运行它自己的schedule()函数，但是，处理器必须交换信息以提高系统性能。

当调度程序计算一个可运行进程的goodness时，还应该考虑这个进程是运行在以前运行的CPU上还是运行在另一个CPU上。一直运行在同一CPU上的进程总是首选的，因为CPU的硬件高速缓存可能依然包含有用的数据。这条规则有助于减少高速缓存的不命中数。

然而，让我们假定，CPU1正在运行一个进程时，第二个高优先级的曾在CPU2上运行的进程变为可运行的。现在内核面临一种有趣的两难选择：是应该在CPU1上立即执行高优先级的进程呢，还是应该延迟那个进程的执行直到CPU2变为可用？在前一种情况下，硬件高速缓存的内容被抛弃；在后一种情况下，CPU2正在运行idle进程（*swapper*）时，SMP的并行性并没有得到充分地挖掘。

为了达到较好的系统性能，Linux/SMP采取一种经验性的规则来解决这种两难情况。采取的选择总是一种折衷，并且这种平衡主要依赖于集成到每个CPU中的硬件高速缓存：高速缓存越大，保持一个进程在那个CPU上就越便利。

Linux/SMP 调度程序的数据结构

aligned_data表包含了每个进程的数据结构，这个表主要用来很快地获得当前进程的描述符。表中每个元素的填充是通过调用schedule()函数进行的，其结构如下：

```
struct schedule_data {
    struct task_struct * curr;
    unsigned long last_schedule;
};
```

curr域指向在相应CPU上正在运行进程的描述符，而last_schedule域指定什么时候schedule()选curr作为运行进程。

进程描述符中包含了几个与SMP相关的域。特别是，avg_slice域保持对进程平均时间片的跟踪，processor域存放执行进程的最后一个CPU的逻辑标识符。

cacheflush_time变量包含对硬件高速缓存内容全部重写所花费的CPU最小周期数的粗略估计。它由smp_tune_scheduling()函数初始化为：

$$\left\lfloor \frac{\text{以KB为大小的高速缓存}}{5000} \times \text{以kHz为单位的CPU频率} \right\rfloor$$

Intel Pentium处理器有8MB的硬件高速缓存，因此，它们的cacheflush_time被初始化为几百个CPU周期，即几微秒。最近Intel处理器有了较大的硬件高速缓存，因此，最小的高速缓存刷新时间范围从50到100微秒。

我们将在后面看到，如果cacheflush_time大于某一当前运行进程的平均时间片，就不会执行进程的抢占，因为在这种情况下，把进程绑定到最后执行的处理器上是很方便的。

schedule()函数

当schedule()函数在SMP系统上执行时，它进行下列操作：

1. 照样执行schedule()的初始化部分。
2. 在this_cpu局部变量中存放正在执行的处理器的逻辑标识符。这个值由prev的processor域来读取。
3. 初始化sched_data局部变量，以便它指向this_cpu CPU的schedule_data结构。
4. 重复调用goodness()以选择将要执行的新进程。这个函数也检查进程的processor域，并对最后在this_cpu CPU上执行的进程给予一定的奖赏(PROC_CHANGE_PENALTY，通常为15)。
5. 如果必要，照样重新计算进程的动态优先级。
6. 把sched_data->curr置为next。
7. 把next->has_cpu置为1，next->processor置为this_cpu。
8. 在t局部变量中存放current时间标记寄存器的值。
9. 在this_slice局部变量中存放prev的最后时间片，这个值是t与sched_data->last_schedule的差。
10. 把sched_data->last_schedule置为t。
11. 把prev的avg_slice域置为(prev->avg_slice+this_slice)/2，换句话说，更新平均值。
12. 执行上下文切换。
13. 当内核返回到这儿时，以前的进程又被调度程序选中，prev局部变量现在指向刚刚被代替的进程。如果prev还依然是可运行的，并且不是这个CPU的空任务，那么，对prev调用reschedule_idle()函数。

14. 把prev的has_cpu域置为0。

reschedule_idle()函数

当进程p变为可运行时，reschedule_idle()函数被调用[参见前面的“schedule()函数”一节]。在SMP系统上，这个函数决定进程是否应该抢占某一CPU上的当前进程。它执行下列操作：

1. 如果p是实时进程，总会试图执行抢占，转到第3步。
2. 如果有一个CPU上的当前进程满足下列两个条件（注4），则立即返回（不试图抢占）：
   - cacheflush_time大于当前进程的平均时间片。如果这个条件为真，则这个进程没有使高速缓存变得太“脏”。
   - 为了存取某一临界内核数据结构，p和当前进程都需要全局内核锁（参见第十一章中的“全局内核锁和局部内核锁”一节）。必须执行这种检查，因为替换一个进程（保持着另一个进程所需的锁）并不是富有成效的。
3. 如果p->processor CPU（p最后运行的CPU）为空，就选它。
4. 否则，对于在某个CPU上运行的每个任务tsk计算差：

   ```
   goodness(tsk, p) - goodness(tsk, tsk);
   ```

   假定这个差为正，就选择这个差值最大的CPU。
5. 如果已选择了CPU，设置相应正运行进程的need_resched域，并向那个处理器发送一条“reschedule”消息（参见第十一章中的“处理器间中断”一节）。

调度算法的性能

Linux的调度算法既是独立的，也是相对易理解的。因为这个原因，很多内核高手热衷于试图对之做出改进。然而，调度程序是内核中相当神秘的一部分。当你可以通过修改几个关键参数而极大地改善它的性能时，通常从理论上不能证明所得结果是有效的。此外，当用户提出的混合请求（实时、交互式、I/O范围、后台等等）变

注4： 这些条件看起来不可思议，但或许它们是能让SMP更好工作的经验性规则。

化较大时，你也不能确信得到的是正面结果还是负面结果。实际上，几乎对每个所提议的调度策略，都可能产生人为的混合请求，而它们又导致了较差的系统性能。

让我们来略述一下Linux调度程序的缺陷。正如将描述的那样，有些局限性对具有很多用户的大型系统影响较大。在一次运行几十个进程的单个工作站上，Linux调度程序的效率是相当高的。因为Linux诞生于Intel 80386，并继续在PC世界非常流行，我们认为当前的Linux调度程序还相当合适。

这种算法不适合进程数量很大的情况

如果现有的进程数量很庞大，重新计算所有动态优先级就是低效的。

在老式传统的Unix内核中，每秒都计算动态优先级，因此，问题甚至更糟糕。Linux试图更换一种方法以减少调度程序的额外开销。只有当所有的可运行进程都用完它们的时间片时，才重新计算优先级。因此，当进程数量较大时，重算过程相当费时，但计算次数大大减少。

这种简单的方式具有这样的缺点，即当可运行进程的数量很大时，I/O范围的进程很少被执行，因此，交互式应用有较长的响应时间。

对高负载的系统来说，预定义的时间片太长

用户所感受的系统响应主要依赖于系统的负载，负载指的是可运行进程的平均数，以及因此而等待CPU的时间（注5）。

如前所述，系统的响应也依赖于可运行进程的平均时间片。在Linux中，对于有很高期望的系统负载，预定义的时间片显得太长。

I/O范围进程的执行策略不是最佳的

优先选择I/O范围进程是确保交互式程序具有短响应时间的好策略，但并不是理想的策略。几乎没有用户交互的一些批处理程序甚至是I/O范围的。例如，考虑一下数据库搜索引擎，必须从硬盘或网络应用（必须从慢速连接的远程主机搜集数据）

注5：uptime程序每过1、5、15分钟返回系统负载。也可以通过读取*/proc/loadavg*文件来获得相同的信息。

有代表性地读很多数据。即使这些种类的进程不需要短的响应时间，但它们也是由调度算法执行的。

另一方面，也是CPU范围的交互式程序可能让用户看起来反映迟钝，因为由于I/O阻塞操作而引起的动态优先级的增加并不能补偿由于CPU的使用而引起的动态优先级的减少。

对实对应用的支持是微弱的

正如第一章中提到的，非抢占式内核并不是很适合实时应用，因为在内核态处理中断或异常可能要花几毫秒。在这期间，变为可运行的实时进程不能被恢复执行。这对于需要可预知的及要求低响应时间的实时应用是无法接受的（注6）。

Linux的未来版本很可能致力于解决这个问题，或者通过实现SRV4的“固定抢占点”来解决，或者通过使内核完全地可抢占而解决。

然而，内核抢占仅仅是实现有效的实时调度程序的几个必需的条件之一。还有几个其他的问题必需考虑。例如，实时进程通常必需使用的资源也是普通进程所需的，直到低优先级的进程释放了某一资源，实时进程才可能因此结束等待，这种现象叫做优先级倒置（priority inversion）。此外，实时进程可能代表另一个低优先级的进程（如内核线程）请求一个内核同意的服务，这种现象叫做隐含调度（hidden scheduling）。一个有效的实时调度程序应该致力于解决这些问题。

与调度相关的系统调用

已经介绍的几个系统调用允许进程改变它们的优先级及调度策略。作为一般规则，总是允许用户降低他们进程的优先级。然而，如果他们想修改属于其他某一用户进程的优先级，或者如果他们想增加他们自己进程的优先级，那么，他们必须拥有超级用户的特权。

注6：Linux内核已在几个方面进行了修改，因此，如果一些硬实时工作较短，内核就能处理它们。硬件中断基本上被捕获，并由一种“超级内核”来监控内核的执行。但这种改变并不能使Linux成为一个真正的实时系统。

注7：因为这个系统调用经常用来降低进程的优先级，因此用户在它们的进程中调用它对其他进程“nice”。

nice()系统调用

nice()(注7)系统调用允许进程改变它们的基本优先级。包含在increment参数中的整数值用来修改进程描述符的priority域。nice Unix命令(允许用户运行修改调度优先级的程序)就是基于这个系统调用的。

sys_nice()服务例程处理nice()系统调用。尽管increment参数可以有任何值,但是大于40的绝对值被截为40。从传统上来说,负值相当于请求优先级增加,并请求超级特权,而正值相当于请求优先级减少。

通过把increment的值拷贝到newprio局部变量,这个函数就开始了。在increment为负的情况下,调用capable()函数核实进程是否有CAP_SYS_NICE能力(capability)。我们将在第十九章中与能力的表示一起来讨论这个函数。如果用户趋于让需要的能力改变优先级,sys_nice()就改变newprio的符号并设置increase局部标志:

```
increase = 0
newprio = increment;
if (increment < 0) {
    if (!capable(CAP_SYS_NICE))
        return -EPERM;
    newprio = -increment;
    increase = 1;
}
```

如果newprio的值大于40,把它截为40。在这点,newprio局部变量可以是0~40之间的任何值,包括0和40。然后,根据调度算法所用的优先级数值范围改变这个值。因为所允许的最高基本优先级是2 × DEF_PRIORITY,新的值是:

$$\lfloor (newpri \times 2 \times DEF\_PRIORITY)/40+0.5 \rfloor$$

以适当的符号把这一结果拷贝到increment:

```
if (newprio > 40)
    newprio = 40;
newprio = (newprio * DEF_PRIORITY + 10) / 20;
increment = newprio;
if (increase)
    increment = -increment;
```

因为newprio是一个整数变量，代码中的表达式与前面所列出的公式是相等的。

然后，从优先级中减去increment来设置优先级最后的值。不过，进程最后的基本优先级不能小于1或大于2 × DEF_PRIORITY:

```
if (current->priority - increment < 1)
    current->priority = 1;
else if (current->priority > DEF_PRIORITY*2)
    current->priority = DEF_PRIORITY*2;
else
    current->priority -= increment;
return 0;
```

调整了优先级的进程如同其他进程一样随着时间的过去而改变，如果必要，获得额外的优先级，或者退后以服从其他的进程。

getpriority()和 setpriority()系统调用

nice()系统调用只影响调用它的进程，而另外两个系统调用getpriority()和setpriority()则作用于给定组中所有进程的基本优先级。getpriority()返回20加给定组中所有进程之中最高的基本优先级；setpriority()把给定组中所有进程的基本优先级都设置为一个给定的值。

内核对这两个系统调用的实现是通过sys_getpriority()和sys_setpriority()服务例程完成的。这两个服务例程本质上作用于一组相同的参数:

which

指定进程组。它采用下列值之一:

PRIO_PROCESS

根据进程的ID选择进程（进程描述符的pid域）

PRIO_PGRP

根据组ID选择进程（进程描述符的pgrp域）

PRIO_USER

根据用户ID选择进程（进程描述符的uid域）

Who

用来选择进程的pid、pgrp及uid域的值（依赖于用哪个值）。如果who是0，把它的值设置为当前进程相应域的值。

niceval

新的基本优先级值［仅仅由sys_setpriority()需要］。它的取值范围应该在-20（最高优先级）和+20（最小优先级）之间。

正如以前提到的，只有有CAP_SYS_NICE能力的进程才被允许增加它们自己的基本优先级或修改其他进程的优先级。

正如我们在第八章已看到的，只有当出现了某一错误时，系统调用才返回一个负值。因为这个原因，getpriority()不返回-20到+20之间正常的nice值，而是0~40之间的一个非负值。

与实时进程相关的系统调用

现在我们介绍一组系统调用，它们允许进程改变调度规则，尤其是可以变为实时进程。一个进程为了修改任何进程的进程描述符的rt_priority和policy域（包括自己的），它照样必须具有CAP_SYS_NICE能力。

sched_getscheduler()和sched_setscheduler()系统调用

sched_ getscheduler()查询由pid参数所标识的进程当前所用的调度策略。如果pid等于0，将检索调用进程的策略。如果成功，这个系统调用返回进程的策略：SCHED_FIFO、SCHED_RR或SCHED_OTHER。相应的sys_sched_getscheduler()服务例程调用find_task_by_pid()，后一个函数确定给定pid所对应的进程描述符，并返回policy域的值。

sched_setscheduler()系统调用既设置调度策略，也设置由参数pid所标识进程的相关参数。如果pid等于0，调用进程的调度程序参数将被设置。

相应的sys_sched_setscheduler()函数检查由policy参数指定的调度策略和由param->sched_priority参数指定的新静态优先级是否有效。还检查进程是否有CAP_SYS_NICE能力，或它自己是否有超级用户特权。如果每件事都可以，执行下列语句：

```
    p->policy = policy;
    p->rt_priority = param->sched_priority;
    if (p->next_run)
        move_first_runqueue(p);
    current->need_resched = 1;
```

sched_getparam()和sched_setparam()系统调用

sched_getparam()系统调用为pid所标识的进程检索调度参数。如果pid是0，当前进程的参数被检索。正如你所期望的，相应的sys_sched_getparam()服务例程找到与pid相关的进程描述符指针，把它的rt_priority域存放在类型为sched_param的局部变量中，并调用copy_to_user()把它拷贝到由param参数指定的地址的进程地址空间。

sched_setparam()系统调用类似于sched_setscheduler()，它与后者的不同在于不让调用者设置policy域的值（注8）。相应的sys_sched_setparam()服务例程几乎与sys_sched_setscheduler()相同，但是，受影响进程的policy从不被改变。

sched_yield()系统调用

sched_yield()系统调用允许进程在不被挂起的情况下自愿放弃CPU，进程仍然处于TASK_RUNNING状态，但调度程序把它放在运行队列链表的末尾。在这种方式下，具有相同动态优先级的其他进程将有机会运行。这个调用主要由SCHED_FIFO进程使用。

相应的sys_sched_yield()服务例程执行这些语句

```
    if (current->policy == SCHED_OTHER)
        current->policy |= SCHED_YIELD;
    current->need_resched = 1;
    move_last_runqueue(current);
```

注意，只有进程是普通的SCHED_OTHER进程时，才在进程描述符的policy域设置SCHED_YIELD域。结果，schedule()的下一次调用将把这个进程当作已用完时间片的进程来处理［参见schedule()如何处理SCHED_YIELD域］。

注8：这种不规则是由POSIX标准的一个指定要求所引起的。

sched_get_priority_min()和sched_get_priority_max()系统调用

sched_get_priority_min()和sched_get_priority_max()系统调用分别返回最小和最大实时静态优先级的值，这个值由policy参数所标识的调度策略来使用。

如果current是实时进程，sys_sched_get_priority_min()服务例程返回1，否则返回0。

如果current是实时进程，sys_sched_get_priority_max()服务例程返回99（最高优先级），否则返回0。

sched_rr_get_interval()系统调用

sched_rr_get_interval()系统调用应该为指定的实时进程获得循环轮转的时间片。

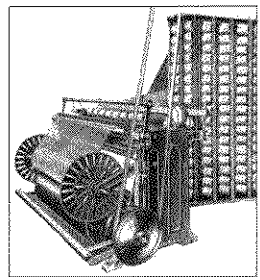
相应的sys_sched_rr_get_interval()服务例程并不像所期望的那样操作，因为在tp所指向的timespec结构中，它总是返回一个150ms的值。这个系统调用还没有在Linux中有效地实现。

对Linux 2.4的展望

对内核线程和僵死进程，Linux 2.4关于TLB的刷新引入一种巧妙的最优化策略。因此，由schedule()函数，而不是由switch_to宏设置活动的全局页目录。

Linux 2.4对SMP的调度算法已经进行了改进和简化。只要一个新的进程变为可运行的，内核检查进程的首选CPU（即最后运行的CPU）是否为空，如果是，内核把进程分配给那个CPU；否则，内核把进程分配给另一个空闲的CPU。如果所有的CPU都忙，内核检查进程是否有足够的优先级抢占运行在首选CPU上的进程。如果没有，只有在新的可运行进程是实时进程或与硬件cache相比有较短的平均时间片的情况下，内核才试图抢占某一其他的CPU。（粗略地说，抢占发生在新的可运行进程是交互式的且首选CPU不立即重新调度时。）

第十一章
内核同步



你可以把内核看作是一个不断对请求进行响应的服务器，这些请求可能来自正在CPU上执行的进程，也可能来自正在执行中断请求的外部设备。我们使用这个比喻来强调内核的各个部分并不是严格按照顺序依次执行的，而是采用交错执行的方式。因此，这些请求可能引起竞争条件，而我们必须采用适当的同步机制对这种情况进行控制。有关这些主题的简要介绍可以参看第一章中的“Unix内核概述”一节。

本章开始部分我们先回顾一下内核请求是何时以交错（interleave）的方式执行以及执行程度如何。然后我们将介绍内核中所实现的四种基本的同步机制，并通过例子说明如何应用这些机制。

接下来的两部分涉及Linux内核对多处理器体系结构的扩展。第一部分描述了对称多处理器（Symmetric Multiprocessor，SMP）体系结构的一些硬件特征，而第二部分讨论了Linux内核的SMP版本所采用的其他互斥机制。

内核控制路径

正如我们介绍的一样，内核函数是根据请求来执行的，请求的方式有如下两种可能：

- 正在用户态执行的进程产生一个异常，例如通过执行一条 `int 0x80` 汇编语言指令异常。
- 外部设备通过一条IRQ线路向可编程中断控制器发送一个信号，并打开相关的中断。

在内核态执行的用来处理内核请求的指令序列被表示为内核控制路径（kernel control path）。例如，当用户态的进程发出一个系统调用请求时，对应内核控制路径的第一条指令就包含在system_call()函数的初始化部分中，而最后一条指令则包含在ret_from_sys_call()函数中。

在第四章的“中断和异常处理程序的嵌套执行”一节中，内核控制路径被定义为内核用来处理系统调用、异常或中断所执行的指令序列。内核控制路径所充当的角色和进程的角色类似，二者之间的区别在于前者更原始：首先，没有任何描述符与内核控制路径相关；其次，内核控制路径不是通过单个函数进行调度的，而是通过把停止和恢复路径的指令序列插入到内核代码中进行调度的。

在最简单的情况中，CPU从第一条指令到最后一条指令顺序地执行内核控制路径。但是，在以下的事件之一发生时，CPU就会交错执行内核控制路径：

- 发生上下文切换。正如我们在第十章中已经所看到的一样，进程调度、上下文切换都只有在schedule()函数被调用时才会发生。
- 中断发生在当CPU正在执行开中断的内核控制路径时。在这种情况下，第一个内核控制路径还没有完成，CPU就开始执行另外一个内核控制路径来处理这个中断。

为了实现多处理技术，交错执行内核控制路径非常重要。另外，正如我们在第四章的“中断和异常处理程序的嵌套执行”一节中已经注意到的一样，交错执行提高了可编程中断控制器和设备控制器的吞吐量。

在交错执行内核控制路径时，要特别注意那些包含几个相关成员变量的数据结构，例如，缓冲区和说明缓冲区长度的整型变量。影响这种数据结构的所有语句都必须被放到一个单独的临界区中，否则，就有可能带来数据混乱的危险。

同步技术

第一章介绍了进程的竞争条件（race condition）和临界区（critical region）的概念。这些定义同样适用于内核控制路径。在本章中，当某个计算的结果依赖于交错的内核控制路径如何嵌套时，可能发生竞争条件。临界区是一段代码，在其他的内核控

制路径能够进入临界区前，进入临界区的每一内核控制路径都必须全部执行完这段代码。

现在我们检查一下在避免共享数据之间的竞争条件的同时，内核控制路径是如何被交错执行的。我们将区分四种主要类型的同步技术：

- 内核态进程的非抢占性
- 原子操作
- 关中断
- 加锁

内核态进程的非抢占性

正如前面已经指出的一样，Linux 内核是非抢占的，也就是说，当正在运行的进程还处于内核态时，不会被抢占（被更高优先级的进程替换）。特别是，以下断言在 Linux 中总是成立：

- 在内核态中运行的进程不会被其他进程取代，除非这个进程主动放弃 CPU 的控制权（注 1）。
- 中断或异常处理可以中断在内核态中运行的进程。但是，在中断处理程序结束时，该进程的内核控制路径被恢复。
- 执行中断或异常处理的内核控制路径只能被执行中断或异常处理的其他内核控制路径所中断。

由以上断言我们可以知道，对于系统调用所启用的其他控制路径来说，处理非阻塞的系统调用的内核控制路径是原子的。这简化了很多内核函数的实现：不是由中断或异常处理程序更新的内核数据结构都可以很安全地被访问。然而，如果处于内核态的进程主动放弃 CPU，那么它必须要保证剩下的所有数据结构都处于一致状态。此外，在进程恢复执行时，必须对以前访问过的可能被改变的所有数据结构进行重新检测。这种变化可能是由一个不同的内核控制路径所引起的，但可能代表一个独立的进程运行的是同一代码。

注 1：当然，所有的上下文切换都是在内核态中执行的。但是，在当前进程要返回用户态时才可能发生上下文切换。

原子操作

避免竞争条件的最简单的方法是在芯片级保证操作是原子的，即操作必须在一条单独的指令内执行。在用来创建临界区的其他的、更灵活的机制基础上，也可以找到这些非常小的原子操作。

因此，原子操作（atomic operation）就是以“原子”的方式（也就是说，操作过程中不会被中断）执行一条单独的汇编语言指令。

让我们根据那个分类来回顾一下Intel 80x86的指令：

- 访问零次或一次内存的汇编指令是原子的。
- 如果在读操作之后、写操作之前没有其他进程占用内存总线，那么从内存中读取数据、更新数据并把更新后的数据写回内存中的这些读/修改/写汇编语言指令（例如inc或dec）是原子的。当然，在单处理器的系统中，永远都不会发生内存总线窃用（stealing）的情况，因为所有的内存访问都是在同一个处理器上进行的。
- 操作码前缀是lock字节（0xf0）的读/修改/写汇编语言指令即使在多处理器系统中也是原子的。当控制单元检测到这个前缀时，就“锁定”内存总线，直到这条指令执行完成为止。因此，其他处理器在加锁的指令执行期间不能访问这个内存单元。
- 操作码前缀是一个rep字节（0xf2，0xf3）的汇编语言指令不是原子的，这条指令强行让控制单元多次重复执行相同的指令，控制单元在再次执行一条新指令之前要检查挂起（pending）的中断。

在编写C代码程序时，你不能保证编译器会为a=a+1或甚至像a++这样的操作都使用一个单独的、原子的指令。因此，Linux内核提供了一些特殊的函数（参见表11-1），用这些函数实现了单独的、原子的汇编语言。在多处理器系统中，每条这种指令都有一个lock字节的前缀。

表11-1 C语言中的原子操作

函数	说明
atomic_read(v)	返回*v。
atomic_set(v,i)	把*v设置成i

表11-1 C语言中的原子操作（续）

函数	说明
atomic_add(I,v)	给*v增加i
atomic_sub(I,v)	从*v中减去i
atomic_inc(v)	给*v加1
atomic_dec(v)	从*v中减去1
atomic_dec_and_test(v)	从*v中减去1，如果结果非空就返回1；否则返回0
atomic_inc_and_test_greater_zero(v)	给*v加1，如果结果为正就返回1；否则就返回0
atomic_clear_mask(mask,addr)	清除由mask所指定的addr中的所有位
atomic_set_mask(mask,addr)	设置由mask所指定的addr中的所有位

关中断

对于那些由于太长而不能定义成原子操作的代码段来说，就需要使用临界区所提供的更复杂的方法。为了保证竞争条件无机可趁（系统中不会给竞争条件留下缺口，即使一条指令大小的缺口也没有），这些临界区通常都是在此基础上执行原子操作。

关中断（interrupt disabling）是用来确保内核语句序列可以作为一个临界区进行操作的一种主要机制。即使是在硬件设备发出IRQ信号时，关中断也允许内核控制路径继续执行，这样就提供了一种有效的方法，来保护中断处理程序也会访问的数据结构。

然而，关中断本身并不总能防止控制路径的交错执行。实际上，内核控制路径可以产生一个“缺页（page fault）”异常，它可以依次把当前进程（以及相关的内核控制路径）挂起。内核控制路径也可以直接调用schedule()函数。这发生在大部分I/O磁盘操作的过程中，其原因是进程可能会阻塞，也就是说这可以强制进程睡眠，直到I/O操作完成为止。因此，当关中断时，内核从不执行阻塞操作，这是因为系统有可能冻结。

我们可以使用汇编语言指令cli来关中断，该指令产生于宏__cli()和宏cli()。可以使用汇编语言指令sti开中断，该指令产生于宏__sti()和宏sti()。在单处理器系统中，cli()等同于__cli()，sti()等同于__sti()。但是，正如我们在本章后面会看到的一样，这些宏在多处理器系统中则完全不同。

当内核进入一个临界区时，清除eflags寄存器的IF标志来关中断。然而内核不能在临界区的末尾简单地再次设置这个标志。因为中断可以以嵌套的方式执行，所以内核在执行到当前的控制路径之前并不知道IF标志的状态。因此所有的控制路径都要保存这个标志原来的设置，在结束时再恢复成这个设置。

内核使用__save_flags宏来保存eflags的内容。在单处理器系统中这个宏和save_flags宏是相同的。内核使用__restore_flags宏和（在单处理器系统中）restore_flags宏来恢复eflags的内容。通常这些宏可以以下面的方法使用：

```
__save_flags(old);
__cli();
 [...]
__restore_flags(old);
```

__save_flags宏把eflags的内容拷贝到old局部变量中，然后使用__cli()清除IF标志。在临界区的末尾使用__restore_flags宏把eflags恢复成原来的内容。这样，如果在内核控制路径执行__cli()宏之前是开中断的，那么现在也是开中断的。

Linux还提供了另外几个同步宏，这些宏在多处理器系统中非常重要（请参看本章后面的“自旋锁”一节），而在单处理器系统中就有些冗余了（参见表11-2）。注意有些函数不会执行任何可见的操作。这些函数只是用作gcc编译器的“屏障”，因为它们可以通过来回移动汇编指令来防止编译器对代码进行优化。参数lck通常都被忽略了。

表11-2 单处理器系统中的中断禁用/启用宏

宏	说明
spin_lock_init(lck)	无操作
spin_lock(lck)	无操作
spin_unlock(lck)	无操作

表 11-2 单处理器系统中的中断禁用/启用宏（续）

宏	说明
spin_unlock_wait(lck)	无操作
spin_trylock(lck)	总是返回 1
spin_lock_irq(lck)	__cli()
spin_unlock_irq(lck)	__sti()
spin_lock_irqsave(lck, flags)	__save_flags(flags); __cli()
spin_unlock_irqrestore(lck, flags)	__restore_flags(flags)
read_lock_irq(lck)	__cli()
read_unlock_irq(lck)	__sti()
read_lock_irqsave(lck, flags)	__save_flags(flags); __cli()
read_unlock_irqrestore(lck, flags)	__restore_flags(flags)
write_lock_irq(lck)	__cli()
write_unlock_irq(lck)	__sti()
write_lock_irqsave(lck, flags)	__save_flags(flags); __cli()
write_unlock_irqrestore(lck, flags)	__restore_flags(flags)

让我们回想一下前面章节中介绍过的这些宏是如何在函数中使用的例子：

- add_wait_queue()和remove_wait_queue()函数用write_lock_irqsave()和write_unlock_irqrestore()函数来保护等待队列。
- setup_x86_irq()为一个特定的IRQ加入一个新的中断处理程序，spin_lock_irqsave()和spin_unlock_irqrestore()函数用来保护处理程序的相应链表。
- run_timer_list()函数使用spin_lock_irq()和spin_unlock_irq()函数保护动态定时器的数据结构。
- handle_signal()使用spin_lock_irq()和spin_unlock_irq()函数来保护current的blocked域。

由于开中断的简单性，在实现临界区的内核函数中被广泛使用。显然，开中断所获得的临界区必须很短，这是因为I/O设备控制器和CPU之间任何种类的通信都是在内核进入临界区时被阻塞的。较长的临界区应该使用加锁的方式实现。

使用内核信号量加锁

一种广泛使用的同步技术是加锁（locking）。当内核控制路径必须存取一个共享的数据结构或进入临界区时，就必须先为它获得一个“锁”。使用锁机制保护的资源就像是一个人在房间里面把门锁上，这样房间内的资源就都是受保护的资源了。如果内核控制路径希望对这种资源进行存取，那么它就试图请求锁来“打开门”。只有在资源空闲时这个操作才能够成功。然后，只要它想使用这个资源，门一直都是锁上的。在内核控制路径释放这个锁时，门就打开了，其它内核控制路径就可以进入这个房间了。

Linux 提供了两种锁：内核信号量（kernel semaphore）和自旋锁（spin lock）。前者在单处理器系统和多处理器系统中都被广泛地使用，而后者只用于多处理器系统。我们在此处只讨论内核信号量，自旋锁将在本章“自旋锁”一节中进行讨论。当一个内核控制路径试图获得由内核信号量所保护的一个繁忙的资源时，相应的进程被挂起。直到该资源被释放时这个进程才变为可运行的。

内核信号量是 `struct semaphore` 类型的对象，有下面这些域：

`count`

存放一个整型值。如果该值大于0，那么资源就是空闲的，也就是说，该资源现在可以使用。相反，如果count小于或等于0，那么这个信号量就是繁忙的，也就是说，这个受保护的资源现在不能使用。在后一种情况下，count的绝对值表示了正在等待这个资源的内核控制路径的数量。该值为0表示有一个内核控制路径正在使用这个资源，但是没有其他内核控制路径正在等待这个资源。

`wait`

存放等待队列链表的地址，该链表中包含当前正在等待这个资源的所有的睡眠进程。当然，如果 count 大于或等于0，等待队列就为空。

`waking`

确保在该资源被释放并唤醒睡眠进程时，只有一个进程能够成功获得该资源。我们很快就会看到这个域的操作。

进程在获得锁时减少 `count` 域的值，在释放锁时增加 `count` 域的值。MUTEX 和 MUTEX_LOCKED 宏被用来初始化互斥访问（exclusive access）所需的信号量，这两个宏分别把 `count` 域设置成1（互斥访问的资源空闲）和0（对信号量进行初始化

的进程当前互斥访问的资源忙)。注意，信号量也可以被初始化为count的任意正数值n，在这种情况下，最多n个进程可以同时访问这个资源。

当进程希望获得内核信号量锁时，就调用down()函数。down()函数的实现非常麻烦，不过它实际上等同于下面的这个函数：

```
void down(struct semaphore * sem)
{
    /* 临界区开始 */
    --sem->count;
    if (sem->count < 0) {
    /* 临界区结束 */
        struct wait_queue wait = { current, NULL };
        current->state = TASK_UNINTERRUPTIBLE;
        add_wait_queue(&sem->wait, &wait);
        for (;;) {
            unsigned long flags;
            spin_lock_irqsave(&semaphore_wake_lock, flags);
            if (sem->waking > 0) {
                sem->waking--;
                break;
            }
            spin_unlock_irqrestore(&semaphore_wake_lock, flags);
            schedule();
            current->state = TASK_UNINTERRUPTIBLE;
        }
        spin_unlock_irqrestore(&semaphore_wake_lock, flags);
        current->state = TASK_RUNNING;
        remove_wait_queue(&sem->wait, &wait);
    }
}
```

该函数减少*sem信号量的count域的值，然后检查该值是否为负。该值的减少和检查过程都必须是原子的，否则其他内核控制路径就可以同时访问该域的值，这样会导致灾难性的后果(请参看第一章的“同步和临界区”一节)。因此，这两个操作都是使用下面的汇编语言指令实现的：

```
movl sem, %ecx
lock /* 仅用于多处理器系统 */
decl (%ecx)
js 2f
```

在多处理器系统中，decl指令使用了一个lock前缀来确保递减操作的原子性（请参看前面的“原子操作”一节）。

如果count的值大于或等于0，那么当前进程就可以获得资源并正常继续执行。否则，count就是负数，这样当前进程必须被挂起。在后一种情况中，通过调用shcedule()函数，当前进程被加入这个信号量的等待队列链表并直接转入睡眠状态。

当资源被释放时进程被唤醒。虽然如此，并不能假定这个资源现在就可用，因为在这个信号量等待队列中可能有几个进程都在等待这个资源。为了选择一个胜出的进程，就要使用waking域。在释放资源的进程要唤醒等待队列中的进程时，就要增加waking的值。每个被唤醒的进程都会紧接着进入一个down()函数的临界区并测试waking值是否为正数。如果被唤醒的进程发现这个域为正值，那么它就减waking的值并获得这个资源；否则就返回睡眠状态。临界区的保护是通过semaphore_wake_lock全局自旋锁和关中断实现的。

注意一个中断处理程序或下半部分一定不能调用down()函数，因为在信号量繁忙时，down()函数会挂起这个进程（注2）。由于这个原因，Linux提供了down_trylock()函数，前面所介绍的异步函数可以安全地使用down_trylock()函数。除了对资源繁忙情况的处理外，该函数和down()函数是相同的。当然，在资源繁忙时，该函数会立即返回，而不是把进程转换成睡眠状态。

系统中还定义了一个稍微不同的函数，即down_interruptible()。在设备驱动程序中会广泛地使用该函数，因为进程只要能接收在信号量上被阻塞的信号，就允许它放弃“down”操作。如果睡眠进程在获得需要的资源之前就被唤醒了，那么这个函数就会增加这个信号量的count域的值并返回-EINTR。另一方面，如果down_interruptible()正常结束并得到了需要的资源，就返回0。这样设备驱动程序在返回值是-EINTR时可能会放弃这次I/O操作。

当进程释放内核信号量锁时，它要调用up()函数，该函数实际上等同于下面的函数：

注2：异常处理程序可以在信号量上阻塞。Linux特别小心地避免产生两个嵌套的内核控制路径为相同的信号量而竞争特定种类的竞争条件，其中一个内核控制路径会永远等待，因为另外一个内核控制路径既不能运行，也不能释放这个信号量。

```
void up(struct semaphore * sem)
{
    /* 临界区开始 */
    ++sem >count;
    if (sem->count <= 0) {
    /* 临界区结束 */
       unsigned long flags;
        spin_lock_irqsave(&semaphore_wake_lock, flags);
        if (atomic_read(&sem->count) <= 0)
            sem->waking++;
        spin_unlock_irqrestore(&semaphore_wake_lock, flags);
        wake_up(&sem->wait);
    }
}
```

该函数增加*sem信号量的count域的值，然后检查该值是否为负数或0。该值的增加和检查过程都必须是原子的，因此这两个操作都是使用下面的汇编语言指令实现的：

```
movl sem, %ecx
lock
incl (%ecx)
jle 2f
```

如果count的新值是正数，而且没有进程正在等待这个资源，那么这个函数就正常结束了。否则，它就唤醒这个信号量等待队列中的进程。为了实现这个功能，它增加waking域（该域是使用semaphore_wake_lock自旋锁和关中断进行保护的）的值，然后对信号量等待队列调用wake_up()。

waking域的增加是在临界区中完成的，因为可能有几个进程在并发访问受保护的同一资源；因此，在等待进程已经被唤醒并且其中一个仍在访问waking域时，一个进程就可以开始执行up()函数。这也就解释了为什么up()要在增加waking值之前检查count是否为非正数，在第一次对count进行检查之后，进入临界区之前，另外一个进程可能已经执行了up()函数。

现在我们来研究一下信号量在Linux中是如何使用的。由于内核是非抢占的，所以只需要几个信号量。实际上，在单处理器系统中竞争条件通常或者发生在执行I/O磁盘操作的过程中进程被阻塞时，或者发生在中断处理程序访问全局内核数据结构

时。在多处理器系统中可能发生其他种类的竞争条件，而在这种情况下Linux更倾向于使用自旋锁（请参看本章后面的“自旋锁”一节）。

以下部分讨论了使用信号量的几个典型例子。

slab高速缓存链表的信号量

slab高速缓存描述符链表（参见第六章中的“高速缓存描述符”一节）是使用cache_chain_sem信号量进行保护的，这个信号量允许互斥地访问和修改该链表。

当kmem_cache_create()在链表中增加一个新的元素，而kmem_cache_shrink()和kmem_cache_reap()顺序地扫描整个链表时就可能产生竞争条件。然而，在处理中断时这些函数从不被调用，在访问链表时也它们也不会阻塞。由于内核是非抢占的，这个信号量只在多处理器系统中才真正起作用。

内存描述符的信号量

每个mm_struct类型的内存描述符在mmap_sem域中都包含了自己的信号量（请参看第七章的“内存描述符”一节）。由于一个内存描述符可以在几个轻量级进程之间进行共享，信号量避免了这个描述符可能产生的竞争条件。

例如，让我们假设内核必须为某个进程创建或扩展一个线性区。为了做到这一点，内核要调用do_mmap()函数，该函数分配一个新的vm_area_structure数据结构。在这样处理的过程中，如果没有空闲内存可用，而且另外一个共享同一内存描述符的进程可能运行，那么当前进程必须被挂起。如果没有信号量，第二个需要访问这个内存描述符的进程的任何操作（例如，由于写时复制而产生的缺页）都可能会导致严重的数据崩溃。

索引节点的信号量

本例涉及文件系统的处理，本书到现在为止还没有对此介绍。因此，我们不用了解太多细节，只需有个大概的印象。正如我们将在第十二章中所看到的一样，Linux是使用一种称为索引节点（inode）的内存对象存放一个磁盘文件的信息。相应的数据结构在i_sem域中包括自己的信号量。

在文件系统的处理过程中会出现很多竞争条件。实际上，磁盘上的每个文件是所有用户共有的一种资源，因为所有进程都会（可能）存取文件的内容、修改文件名或文件位置、删除或复制文件等等。

例如，让我们假设一个进程正在显示在某个目录中所包含的文件。由于每个磁盘操作都可能会阻塞，因此即使在单处理器系统中，当第一个进程正在执行显示操作的中间，其他进程也可能存取同一目录并修改了它的内容。或者，两个不同的进程可能同时修改同一目录。通过用索引节点信号量对目录文件进行保护，所有这些竞争条件都被避免。

避免信号量上的死锁

只要一个程序使用两个或多个信号量时，都存在发生死锁的可能，因为两个不同的路径可能因为彼此等待释放一个信号量而僵死。典型的死锁发生的条件是：当一个内核控制路径为信号量A获得了锁，并正在等待信号量B；而另外一个内核控制路径保持着信号量B的锁，并正在等待信号量A。Linux对于信号量的请求很少会出现死锁问题，因为每个内核控制路径通常每次只需获得一个信号量。

然而，在两种情况下内核必须获得两个信号量锁，这发生在rmdir()和rename()系统调用的服务例程中（注意在这两种情况下，在操作中涉及两个索引节点）。为了避免这种死锁情况，信号量请求的执行按照给定的地址顺序进行：位于低地址的信号量数据结构首先发出信号量请求。

SMP体系结构

对称多处理技术（symmetrical multiprocessing，SMP）是指多处理器的体系结构，其中并没有选定一个CPU作为主CPU，而是所有这些CPU在平等的基础上协同工作，因此而得名“对称”。和以前一样，我们将集中介绍Intel的SMP体系结构。

在一个多处理器系统中究竟包含多少独立的CPU才能获得最大性能是一个热点问题。问题主要在于在高速缓存系统领域的发展速度惊人。硬件高速缓存所带来的很多优点都浪费在对CPU芯片上的局部硬件高速缓存进行同步的总线周期上了。CPU的数量越多，浪费的问题就越严重。

然而，从内核的设计观点来看，我们完全可以忽略这个问题，不管有多少个CPU，SMP的内核都还是相同的。从一个CPU（单处理器系统）转换到两个CPU上，内核的复杂性出现了一个极大的跳跃。

在继续描述Linux为了成为一个真正的SMP内核而不得不做修改之前，我们先简单地回顾一下Pentium双处理器系统的硬件特性。这些特性体现在计算机体系结构的以下方面：

- 共享内存
- 硬件高速缓存同步
- 原子操作
- 分布式中断处理
- CPU同步的中断信号

有些硬件问题完全在硬件内部可以解决，因此我们无须过多地介绍。

公共（common）内存

所有的CPU都共享同一内存，也就是说，所有的CPU都连接到一个公共总线上。这就意味着RAM芯片可以同时被多个独立的CPU访问。由于对RAM芯片的读写操作必须被串行地执行，因此在总线和每个RAM芯片之间插入一个称为内存仲裁器（arbiter）的硬件电路。其作用是如果芯片空闲就给CPU授权访问，如果芯片正忙就将访问请求推迟。即使单处理器系统也会使用内存仲裁器，因为它包含了一个称为DMA的专用处理器，DMA和CPU是并发操作的［请参看第十三章的“直接内存访问（DMA）”一节］。

在多处理器系统中，这个仲裁器的结构更加复杂，因为它有多个输入端口。例如，双Pentium的CPU在每个芯片的入口处都有一个两端口的仲裁器，在CPU使用总线之前需要两个CPU交换同步信息。从编程的观点来看，这个仲裁器是不可见的，因为它是由硬件电路进行管理的。

对高速缓存同步的硬件支持

第二章的“硬件高速缓存”一节说明了硬件高速缓存的内容以及RAM在硬件层维

护这些内容的一致性。在双处理器的情况下也使用了同样的方法。正如图11-1中说明的那样，每个CPU都有自己的本地硬件高速缓存。但是现在的更新就更费时间了：只要一个CPU修改它自己的硬件高速缓存，就必须检测在其他硬件缓存中否也包含了相同的数据；如果是，就要通知其他CPU使用正确的值进行更新。这种行为通常称为高速缓存探听（cache snooping）。幸运的是，这些工作都是在硬件层上实现的，和内核毫无瓜葛。

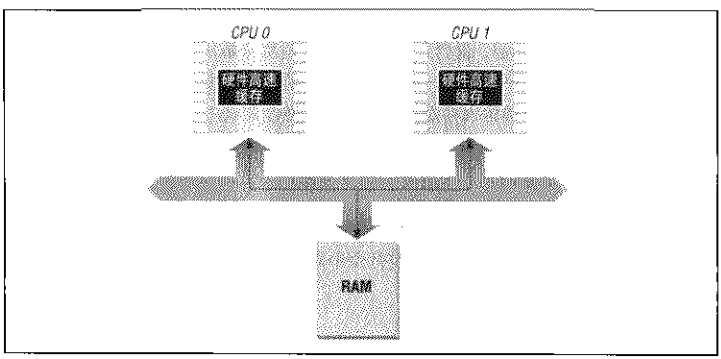


图11-1 双处理器中的缓存

SMP原子操作

有关单处理器系统的原子操作已经在“原子操作”一节中介绍过了。由于标准的读－修改－写指令实际上要两次访问内存总线，因此这些指令在多处理器系统中就不可能是原子的。

让我们给出一个简单的例子来说明SMP内核使用这个标准指令时可能发生什么。考虑在本章前面“使用内核信号量加锁”一节中所介绍的信号量的实现，假设down()函数使用一个简单的decl汇编语言指令减少count域的值并对其进行检查。如果在两个不同的CPU上运行的两个进程同时对同一信号量执行了decl指令会发生什么情况呢？decl是一条读－修改－写指令，它要对同一内存单元进行两次存取：第一次读出原来的值，第二次写入新的值。

开始时，两个CPU都要试图读取同一内存单元，但是内存仲裁器会给其中一个CPU授权访问，同时延迟另外一个CPU的访问请求。但是，当第一个读操作完成时，被延迟的那个CPU也会从相同的位置读出相同（原来）的值。接下来两个CPU都试图对同一内存单元写入相同（新）的值，内存仲裁器又让这个总线内存存取请求串行地进行，最终两个写操作都能成功执行，这个内存单元的值仍就是原来的值减去1。当然，最终的结果完全不对。例如，如果count开始时设置成了1，那么这两个内核控制路径会同时获得对这个受保护资源的互斥访问权限。

从早期的Intel 80286开始，就引入了lock指令前缀来解决这种问题。从程序员的观点来看，lock只是一个用于一条汇编指令之前的特殊字节。当控制单元检测到一个lock字节时，就锁定内存总线，这样其他处理器就不能存取下一条汇编语言指令的目的操作数所指定的内存单元。只有在这条指令执行完时总线锁才会被释放。因此，带有lock前缀的读－修改－写指令即使在多处理器环境中也是原子的。

Pentium允许在18条不同的指令之前使用lock前缀。此外，一些类似于xchg之类的指令并不需要lock前缀，因为这些指令中总线锁定是由CPU的控制单元隐含地强制执行。

分布式中断处理

能够向系统中的任意CPU发出中断对于充分利用SMP体系结构的并行能力是相当重要的。正是出于这个原因，Intel引入了一种新的称为I/O APIC（I/O Advanced Programmable Interrupt Controller，I/O高级可编程中断控制器）的部件，使用它来代替原来的8259A可编程中断控制器。

图11-2采用示意图的形式说明了多APIC系统的结构。每个CPU芯片都有自己完整的本地APIC。中断控制器通信（Interrupt Controller Communication，ICC）总线把一个前端I/O APIC连接到本地APIC上。设备上的IRQ线连接到I/O APIC上，这样对本地APIC而言，这个I/O APIC起一个路由器的作用。

每个本地APIC都有几个32位的寄存器，一个内部时钟，一个定时器设备，240个不同的中断向量，以及另外两条为局部中断保留的IRQ线路，这两条线路用来重启系统。

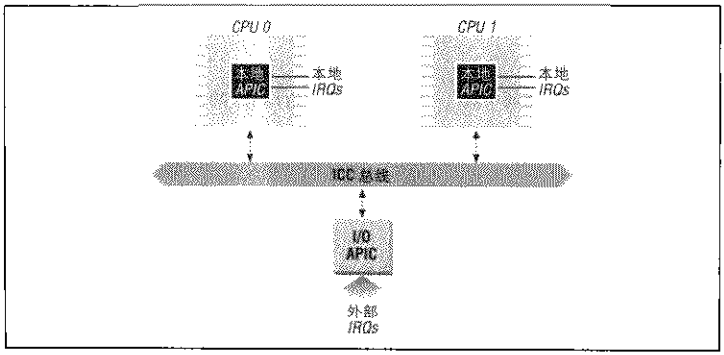


图11-2 APIC系统

I/O APIC由一组IRQ线路、一个有24个表项的中断重定向表(Interrupt Redirection Table)、一个可编程寄存器和一个用来发送和接收经过ICC总线的APIC消息的消息单元组成。和8259A的IRQ引脚不同，中断优先级和引脚号无关，重定向表中的每个表项都可以被单独编程来说明中断向量和优先级、目的处理器以及如何选定处理器。重定向表中的信息用来把外部IRQ信号转换成通过ICC总线发往一个或多个本地APIC单元的消息。

中断请求可以使用两种方法在可用CPU之间进行分布：

固定模式

把IRQ信号发送到相应的重定向表表项所列出的本地APIC上。

最低优先级模式

把IRQ信号发送到正在执行优先级最低的进程的处理器的本地APIC上。所有的本地APIC都有一个可编程任务优先级寄存器（task priority register），它包含了当前正在运行的进程的优先级。在每次任务切换时这个寄存器的值必须由内核进行修改。

APIC的另外一个重要特性是允许CPU产生处理器间中断(interprocessor interrupt，IPI)。当一个CPU想要向其他CPU发送中断时，它就把中断向量和目的处理器的本地APIC标志符保存到自己本地APIC的中断命令寄存器中。然后通过ICC总线向

目的处理器的本地APIC发送一条消息，这样，目的处理器就向自己的CPU发出一个相应的中断。

在本章后面的“处理器间中断”一节中我们会讨论Linux的SMP版本是如何使用这些处理器间中断的。

Linux/SMP 内核

Linux 2.2对于SMP的支持遵守Intel多处理器规范1.4版本，这份规范制定了多处理器平台接口的标准，同时完整地维护了PC/AT二进制兼容性。

正如我们在本章前面“内核态进程的非抢占性”一节中所看到的一样，在Linux的单处理器系统中，竞争条件的作用是相当有限的，因此可以使用开中断和内核信号量来对那些会被中断或异常处理程序异步访问的数据结构进行保护。但是在多处理器系统中，这些问题就更加复杂了：可能有几个进程同时在内核态中运行，这样即使正在运行的进程都没有被抢占，那么也可能发生数据结构崩溃的情况。对SMP内核数据结构进行同步访问的常用方法是使用信号量和自旋锁（请参看后面的“自旋锁”一节）。

在详细讨论Linux 2.2如何串行访问多处理器系统中的内核数据结构之前，让我们额外简要介绍一下，在Linux首次引入SMP支持时这个目标是如何达到的。为了利于从单处理器的内核转化成多处理器内核，原来的2.0版本的Linux/SMP采用了这样一条强制规则：

在任意给定的时刻，最多只有一个处理器可以访问内核数据结构并处理中断。

这条规则规定每个想访问内核数据结构的处理器都必须先获得一个全局锁。只要持有这个全局锁，这个处理器就可以独占访问所有的内核数据结构。当然，由于这个处理器还要处理所有产生的中断，因此中断和异常处理程序异步访问的这些数据结构必须还使用关中断和内核信号量进行保护。

这种方法虽然非常简单，却有一个严重的缺陷：进程要花较长的时间计算它在内核所花费的时间，因此这条规则可以强制I/O范围的进程顺序地被执行。这种情况远远不能令人满意，因此在下一个稳定版本的Linux/SMP（2.2）中，就不严格强调这条规则了。相反，2.2版本中增加了很多锁，一个锁可以提供对一个内核数据结构或

临界区的独占访问。因此，只要几个进程访问的是由锁保护的不同数据结构，那么这些进程就可以在内核态并行地运行。但是，2.2版本中仍旧保留了一个全局内核锁（请参看本章后面的“全局内核锁和局部内核锁”一节），这是因为并非所有的内核数据结构都可以使用特定的锁来保护。

图11-3对更灵活的Linux 2.2系统进行了说明。图中有5个内核控制路径（P_0，P_1，P_2，P_3和P_4）试图对两个临界区（C_1和C_2）进行访问。内核控制路径P_0正在C_1中，而P_2和P_4正等待进入C_1。同时，P_1正在C_2中，而P_3正在等待进入C_2。注意P_0和P_1可以并行运行。临界区C_3的锁现在打开着，因为没有内核控制路径需要进入C_3。

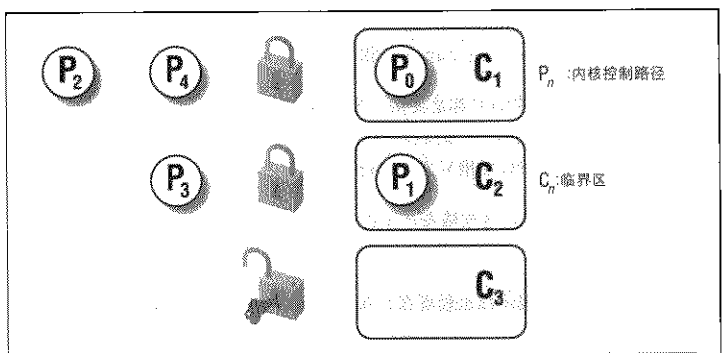


图11-3 使用锁保护临界区

主要的SMP数据结构

为了处理多个CPU的情况，内核必须能够表示每个CPU上发生的行为。在本节中，我们要考虑一些为处理多处理器情况而加入的重要的内核数据结构。

最重要的信息是现在每个CPU上正在运行的是什么进程，但是这些信息实际上并不需要特定CPU的新数据结构。相反，每个CPU都可以用单处理器系统所定义的相同的`current`宏来获取当前进程，因为每个CPU可以从`esp`堆栈指针寄存器中导出进程描述符的地址，所以这个宏会产生一个依赖于CPU的值。

第一组特定CPU的新变量涉及SMP体系结构。Linux/SMP对于CPU的个数有硬件连线的限制，该值是由NR_CPUS宏（通常是32）来定义的。

在系统初始化阶段，Linux在启动的那个CPU上运行，探测是否存在其他CPU（有些SMP主板的CPU插槽可能是空的）。其结果是初始化一个计数器和一个位图：max_cpus记录了现有的CPU数目，而cpu_present_map则说明了哪个插槽中有CPU。

现有的CPU未必一定要激活，也就是说，内核未必对所有的CPU都进行初始化，也未必识别出所有的CPU。另外一对变量跟踪记录活动的CPU，它们是计数器smp_num_cpus和位图cpu_online_map。如果有些CPU不能被正确地初始化，那么内核就清除cpu_online_map中的相应位。

在Linux中，每个活动的CPU都是使用一个称为CPU ID的逻辑序列号进行标识的，该值未必要和CPU插槽号一一对应。cpu_number_map和__cpu_logical_map数组允许在CPU ID和CPU插槽号之间进行转换。

进程描述符包括以下域，这些域表示了进程和处理器之间的关系：

has_cpu
: 一个标志，用来表示该进程现在正在运行（值为1）还是不在运行（值为0）。

processor
: 表示正在运行进程的CPU的逻辑号；或者，如果CPU不在运行，那么就是NO_PROC_ID。

smp_processor_id()宏返回current->processor的值，也就是执行该进程的CPU的逻辑号。

在使用fork()创建一个新进程时，该进程描述符的has_cpu和descriptor域分别被初始化成0和NO_PROC_ID。当schedule()选择一个新进程来运行时，就把has_cpu域设置成1，并把processor域设置成正在执行任务切换的CPU的逻辑号。被替换的那个进程的相应域则分别设置成0和NO_PROC_ID。

在初始化的过程中系统创建了smp_num_cpus个不同的*swapper*进程，每个*swapper*进程的PID都为0，而且被限定到特定的CPU上。通常，只有在相应的CPU空闲时，*swapper*进程才被执行。

自旋锁

自旋锁是用来在多处理器环境中工作的一种加锁机制。它和前面介绍的内核信号量类似，不同之处在于当一个进程发现这个锁已经被其他进程锁定时，该进程就重复“旋转”，执行一条紧（tight）指令循环。

当然，自旋锁在单处理器环境中是毫无用处的，因为等待进程可能保持运行，因此持有锁的那个进程就没有机会释放锁。但是在多处理器环境中自旋锁要方便得多，因为系统的负载很小。换句话说，由于上下文切换会占用大量的时间，因此对于每个进程来说，在等待资源时拥有自己的CPU并简单旋转效率就会更高。

每个自旋锁都使用一个spinlock_t结构来表示，该结构只含有一个lock域，该域值为0和1，分别表示“未加锁”和“已加锁”状态。SPIN_LOCK_UNLOCKED宏把一个自旋锁初始化成0。

对自旋锁进行操作的函数是基于读/修改/写原子操作的，这样就可以保证即使在不同CPU上运行的其他进程试图同时修改自旋锁，在这个CPU上运行的进程也可以适当地更新自旋锁（注3）。

spin_lock宏用来获得一个自旋锁。该宏使用这个自旋锁的slp地址作为参数，本质上展开成下列代码：

```
1: lock; btsl $0, slp
   jnc  3f
2: testb $1,slp
   jne 2b
   jmp 1b
3:
```

btsl原子指令把*slp中的第0位的值拷贝到进位标志中，然后再置该位。接下来对这个进位标志进行测试：如果为0，就说明这个自旋锁还没有被加锁，因此可以正常地在标号3继续执行［后缀f说明这个标号是一个“前向（forward）”标号：它在后面的程序中出现］。否则，就一直执行标号2［后缀b说明这是一个“后向（backward）”标号］处的紧循环，直到自旋锁认为该值是0为止。然后从标号1处

注3：　具有讽刺意义的是：自旋锁是全局的，因此本身必须防止并发访问。

重新执行，这是因为不检查是否已经有其他进程已获得了这个锁就继续下去可能是不安全的（注4）。

spin_unlock宏释放以前所获得的一个自旋锁，它本质上展开成下面的代码：

```
lock; btrl $0, slp
```

btrl原子汇编语言指令清除自旋锁*slp的第0位。

系统中还引入了其他几个宏来处理自旋锁，这些宏在多处理器系统中的定义如表11-3所示（这些宏在单处理器系统中的定义请参看表11-2）。

表11-3 多处理器系统中的自旋锁宏

宏	说明
spin_lock_init(slp)	把slp->lock设置成0
spin_trylock (slp)	把slp->lock设置成1，如果获得锁就返回1，否则返回0
spin_unlock_wait(slp)	循环，直到slp->lock变成0为止
spin_lock_irq(slp)	__cli(); spin_lock(slp)
spin_unlock_irq(slp)	spin_unlock(slp); __sti()
spin_lock_irqsave(slp,flags)	__save_flags(flags); __cli(); spin_lock(slp)
spin_unlock_irqrestore(slp,flags)	spin_unlock(slp); __restore_flags(flags)

注意只有在中断处理程序从不访问内核数据结构时，spin_lock和spin_unlock才能真正发挥保护作用。由于中断处理程序所增加的复杂性，内核必须使用自旋锁宏来禁用或重新启用对本地CPU的中断。否则，如果一个中断处理程序试图获得一个繁忙的自旋锁就可能引起死锁（注5）。

注4：spin_lock的实际实现要稍微复杂一点。标号2处的代码只有在自旋锁忙时才会执行到，这段代码包含在一个附加段中，因此在通常的情况中（自旋锁空闲）硬件缓存中不会有不用执行的代码。在我们的讨论中省略了这些优化细节。

注5：只有在本地CPU上中断才需要禁用，如果中断处理程序试图得到分配给其他CPU上的内核控制路径的自旋锁时，就会发生死锁。

读/写自旋锁

读/写自旋锁（read/write spin lock）的引入是为了增加内核的并发能力。只要没有内核控制路径对数据结构进行修改，读/写自旋锁就允许多个内核控制路径同时读取同一数据结构。如果一个内核控制路径想对这个结构进行写操作，那么它必须首先获取读/写锁的写锁，写锁授权独占访问这个资源。当然，允许对数据结构并发读可以提高系统性能。

图11-4中有两个使用读/写锁保护的临界区C1和C2。内核控制路径R0和R1正在同时读取C1中的数据结构，而W0正等待获取写锁。内核控制路径W1正对C2中的数据结构进行写入操作，而R2和W2分别等待获取读锁和写锁。

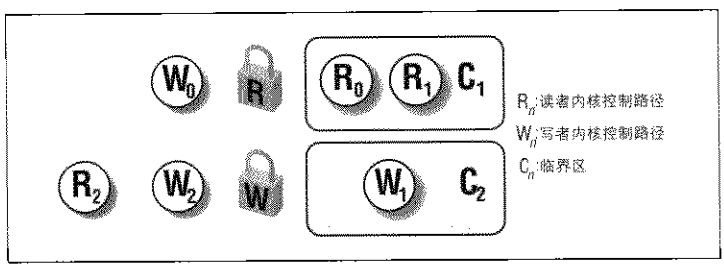


图11-4 读/写自旋锁

每个读/写自旋锁都是一个rwlock_t结构。其lock域是一个32位的计数器，该计数器表示当前正在对受保护的数据结构进行读取操作的内核控制路径的数目。lock域的最高位是写锁，该位在内核控制路径对数据结构进行修改时置位（注6）。RW_LOCK_UNLOCKED宏把一个读/写自旋锁的lock域初始化为0。read_lock宏被应用到读/写自旋锁的地址rwlp，本质上展开成下列代码：

```
1: lock; incl rwlp
   jns 3f
   lock; decl rwlp
2: cmpl $0, rwlp
   js 2b
   jmp 1b
3:
```

注6：当有超时2 147 483 647个读者时，该位也被置位，当然这个极限不可能达到。

在把rwlp->lock的值加1之后，该函数就检查这个域的值是否为负数——也就是说，是否这个锁已经为写入操作加锁了。如果没有，执行过程就继续，可以执行到标号3。否则，该宏就恢复前面的值，一直旋转，直到最高位变成0为止；然后从开始处重新执行。

read_unlock函数被应用到读/写自旋锁的地址rwlp，展开成下列的汇编语言指令：

```
lock; decl rwlp
```

write_lock函数被应用到读/写自旋锁的地址rwlp，展开成下列指令：

```
1: lock; btsl $31, rwlp
   jc 2f
   testl $0x7fffffff, rwlp
   je 3f
   lock; btrl $31, rwlp
2: cmp $0, rwlp
   jne 2b
   jmp 1b
3:
```

这段代码对rwlp->lock的最高位置位。如果原来的值是1，那么写锁早就使用了，因此执行过程从标号2开始。此处，该宏执行一个紧循环来等待lock域变成0（从而说明写锁已经释放了）。如果最高位的原来值是0（说明没有写锁），那么该宏就检查是否有读锁。如果有，就释放写锁，该宏一直等待到lock变成0；否则，CPU就独占访问这个资源，这样，执行过程就从标号3处继续执行。

最后，write_unlock宏被应用读/写自旋锁的地址rwlp，展开成如下指令：

```
lock; btrl $31, rwlp
```

表11-4给出了本节所描述的所有宏的安全中断版本。

表11-4 多处理器系统中的读/写自旋锁宏

函数	说明
read_lock_irq(rwlp)	__cli(); read_lock(rwlp)
read_unlock_irq(rwlp)	Read_unlock(rwlp); __sti()
write_lock_irq(rwlp)	__cli(); write_lock(rwlp)

表11-4 多处理器系统中的读/写自旋锁宏（续）

函数	说明
write_unlock_irq(rwlp)	Write_lock(rwlp); __sti()
read_lock_irqsave(rwlp,flags)	__save_flags(flags); __cli(); read_lock(rwlp)
read_unlock_irqrestore(rwlp,flag)	Read_unlock(rwlp); __restore_flags(flags)
write_lock_irqsave(rwlp,flags)	__save_flags(flags);__cli(); write_lock(rwlp)
write_unlock_irqrestore(rwlp,flags)	Write_unlock(rwlp); __restore_flags(flags)

Linux/SMP 中断处理

前面我们已经说明，在Linux/SMP中中断是由I/O APIC广播到所有的本地APIC（也就是所有的CPU）的。这就意味着IF标志置位的所有CPU都会接收到相同的中断。不过，尽管所有的CPU都必须对接收中断的本地APIC做出应答，但是只有一个CPU必须处理这个中断。

为了实现这个功能，每个IRQ主描述符（参见第四章的“IRQ数据结构”一节）都包括一个IRQ_INPROGRESS标志。如果这个标志被置位，那么相应的中断处理程序就已经在某个CPU上运行了。因此，当每个CPU对接收中断的它的本地APIC做出应答时，都要检查这个标志是否已经设置了。如果是，这个CPU就不处理这个中断，直接退回到刚才正在运行的地方；否则，这个CPU就设置这个标志并开始执行中断处理程序。

当然，对于IRQ主描述符的访问必须是互斥的。因此，每个CPU在检查IRQ_INPROGRESS值之前通常都会获得一个irq_controller_lock自旋锁。同一个锁还可以用来防止多个CPU同时无足轻重地使用中断控制器，这种预防措施对于原来那些所有的CPU都共享一个外部中断控制器的SMP机器来说是必须的。

IRQ_INPROGRESS标志确保每个特定的中断处理程序本身在所有的CPU之间都是原子的。但是，几个CPU可以同时处理不同的中断。global_irq_count变量包含

在给定时刻所有CPU上正在处理的中断处理程序的个数。这个值可能大于CPU的个数，因为任何中断处理程序都可以被其他种类的中断处理程序所中断。同理，local_irq_count数组保存了每个CPU上正在执行的中断处理程序的个数。

正如我们已经看到的一样，为了防止中断处理程序所访问的内核数据结构的崩溃，内核必须经常关中断。当然，由__cli()宏所提供的本地CPU关中断还不足以满足要求，因为它不能防止其他CPU访问内核数据结构。通常的解决方法包括使用一个安全IRQ宏（例如spin_lock_irqsave）来获取一个自旋锁。

但是在少数情况下，应该在所有的CPU上都禁用中断。为了达到这个目的，内核并不会清除所有CPU的IF标志，相反，内核使用global_irq_lock自旋锁来延迟中断处理程序的执行。global_irq_holder变量中包含了持有锁的CPU的逻辑标志符。get_irqlock()函数获得这个自旋锁，并等待在其他CPU上运行的中断处理程序完成。如果调用程序本身不是下半部分（bottom half），那么这个函数就要等待其他CPU上运行的所有下半部分执行完。直到通过调用release_irqlock()释放该锁之后，其他CPU上的中断处理程序才可以继续执行。

全局关中断是使用cli()宏来实现的，该宏只是简单地调用__global_cli()函数：

```
__save_flags(flags);
if (.(flags & (1 << 9))) /* testing IF flag */
    return;
cpu = smp_processor_id();
__cli();
if (.local_irq_count[cpu])
    return;
get_irqlock(cpu);
```

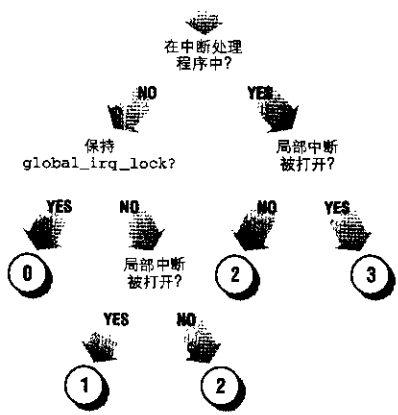
注意：当CPU正在执行已经禁用的局部中断或中断处理程序本身时，全局关中断是不能执行的（注7）。

注7：如果这个限制取消了，那么很容易就发生死锁条件。例如，假设cli()可以把一个局部关中断提升为一个全局关中断。考虑一个正在执行局部关中断的、由一些自旋锁保护的内核控制路径。这个临界区可以合法地包含一个cli()宏，因此它可以调用一个局部开中断的CPU所访问的函数。get_irqlock()函数开始等待其他CPU上的中断处理程序完成。然而，另外一个内核控制路径中的一个中断处理程序可能正在瞄准这个对临界区进行保护的自旋锁，等待第一个内核控制路径释放它呢，这样就产生了死锁！

全局开中断是使用sti()宏来实现的，该宏只是简单地调用__global_sti()函数：

```
cpu = smp_processor_id();
if (!local_irq_count[cpu])
    release_irqlock(cpu);
__sti();
```

Linux还提供了SMP版本的__save_flags和__restore_flags宏，分别称为save_flags和restore_flags：这两个宏分别为正在执行中断处理的CPU保存和恢复控制中断处理的信息。正如图11-5所说明的一样，save_flage会根据3个条件产生一个值，restore_flags则根据save_flags所产生的值执行操作。



save_flags()值	restore_flags()操作
0	__global_cli()
1	__global_sti()
2	__cli()
3	__sti()

图11-5 save_flags()和restor_flags()所执行的操作

最后，在内核控制路径想要和所有的中断处理程序进行同步时，就要调用synchronize_irq()函数：

```
if (atomic_read(&global_irq_count)) {
    cli();
    sti();
}
```

程序执行到cli()时，该函数就获得了global_irq_lock自旋锁，然后一直等待，直到所有正在执行的中断处理程序都完成为止。这个步骤完成之后，就重新启用中断。在设备驱动程序要确认中断处理程序所执行的所有操作都已经完成时，synchronize_irq()函数通常由设备驱动程序调用。

Linux/SMP 下半部分处理

下半部分的处理方法和中断处理程序类似，但是一个下半部分不能和其他下半部分并发运行。而且，禁用中断的同时也禁用了下半部分的运行。global_bh_count变量是一个标志，它定义了一个下半部分现在在某个CPU上是否是活动的。在内核控制路径必须等待当前正在执行的下半部分结束时就要调用synchronize_bh()函数。

global_bh_lock变量用来禁止所有的CPU执行下半部分，换句话说它确保某个临界区对所有CPU上运行的所有下半部分都是原子的。

start_bh_atomic()函数用来锁定下半部分，包含以下内容：

```
atomic_inc(&global_bh_lock);
synchronize_bh();
```

另外一个函数end_bh_atomic()用来重新启用下半部分，该函数要执行：

```
atomic_dec(&global_bh_lock);
```

因此，do_bottom_half()函数只有在以下情形中才会执行下半部分：

- 当前在任一CPU上都没有运行其他下半部分（global_bh_count为0）。
- 下半部分没有被禁用（global_bh_lock为0）。
- 任一CPU上都没有运行中断处理程序（global_irq_count为0）。

- 中断是全局启用的（global_irq_lock为0）。

这种下半部分的串行执行过程是从以前的Linux中版本继承下来的。如果允许下半部分并发执行，那么所有使用下半部分的设备驱动程序都得全部重写。

全局内核锁和局部内核锁

正如我们前面已经介绍的一样，在2.2版本的Linux/SMP中仍然广泛使用了一个名为kernel_flag的全局内核锁。在2.0版本中，这个自旋锁的粒度很粗，只能保证某时刻只有一个处理器可以在内核态中运行。2.2版本的内核要复杂得多，不能再只依赖于这一个自旋锁了。但是，仍然使用这个自旋锁来对很多内核数据结构进行保护，这些数据结构有：

- 所有和虚拟文件系统和文件处理有关的数据结构（请参看第十二章）
- 和网络有关的大部分内核数据结构
- 进程间通信（IPC）所使用的所有内核数据结构，请参看第十八章
- 一些不太重要的内核数据结构

这个全局内核锁之所以仍旧存在，是因为不值得再引入新的锁：死锁和竞争条件必须谨慎地避免。

所有和文件有关的（包括和文件内存映射有关的）系统调用服务例程在开始操作之前必须获得这个全局内核锁，在执行完成时必须把这个全局内核锁释放。因此，很多系统调用就不能在Linux/SMP上并发执行了。

每个进程描述符都包含了一个lock_depth域，该域允许同一进程多次获得全局内核锁。因此，对全局内核锁的连续两次请求不会挂起处理器（与普通的自旋锁一样）。如果进程不需要这个全局内核锁，那么该域的值就是-1。如果进程需要全局内核锁，那么该域的值加1的结果就说明这个全局内核锁已被请求多少次。lock_depth域对于中断处理程序、异常处理程序以及下半部分来说都是相当重要的。假如没有这个域，如果当前进程已经持有全局内核锁了，那么试图获取全局内核锁的任何异步函数都会产生死锁。

lock_kernel()和unlock_kernel()函数分别用来获取/释放这个全局内核锁。lock_kernel()函数等同于：

```
if (++current->lock_depth == 0)
spin_lock(&kernel_flag);
```

而unlock_kernel()函数等同于:

```
if (--current->lock_depth < 0)
    spin_unlock(&kernel_flag);
```

注意lock_kernel()和unlock_kernel()函数中的if语句并不需要原子地执行，因为lock_depth并不是一个全局变量，每个CPU都对它自己的当前进程描述符中的一个域进行寻址。if语句内部的局部中断也不会产生竞争条件：即使新的内核控制路径调用了lock_kernel()，它也必须在结束之前释放这个全局内核锁。

虽然仍然使用这个全局内核锁来保护很多内核数据结构，但是通过引入很多其他粒度更细的锁来取代全局内核锁的作用，这项工作已经开始进行。表11-5给出了已经使用特定的（读/写）自旋锁保护的一些内核数据结构。

表11-5 各种内核自旋锁

自旋锁	所保护的资源
console_lock	控制台
dma_spin_lock	DMA的数据结构
inode_lock	索引节点的数据结构
io_request_lock	块IO子系统
kbd_controller_lock	键盘
page_alloc_lock	伙伴系统（buddy system）的数据结构
runqueue_lock	运行队列链表
semaphore_wake_lock	信号量的waking域
tasklist_lock (rw)	进程链表
tasksiot_lock	在task中空闲表项的链表
timerlist_lock	动态定时器链表
tqueue_lock	任务队列的链表
uidhash_lock	UID hash表
waitqueue_lock (rw)	等待队列的链表
xtime_lock (rw)	Xtime和lost_ticks

正如前面所说明的一样，细粒度的加锁机制可以提高系统的性能，因为很少有大段的代码在处理器之间执行。例如，一个访问运行队列的内核控制路径可以和另外一个对有关文件的系统调用提供服务的内核控制路径并发执行。同理，只要两个内核控制路径中的一个不想修改进程链表，那么两个内核控制路径就可以使用一个读/写锁同时访问进程链表。

处理器间中断

处理器间中断（IPI）是SMP体系结构的一部分，Linux用它来在CPU之间交换消息。Linux/SMP提供了以下函数来处理处理器间中断：

`send_IPI_all()`

向所有的CPU（包括自己）发送一个IPI

`send_IPI_allbutself()`

向除自己之外的所有的CPU发送一个IPI

`send_IPI_self()`

向自己发送一个IPI

`send_IPI_single()`

向一个指定的CPU发送IPI

根据I/O APIC的配置情况，内核有时要调用`send_IPI_self()`函数。其他函数用来实现处理器间的消息。

Linux/SMP可以识别五种消息，这些消息是由接收消息的CPU作为不同的中断向量来解释的：

`RESCHEDULE_VECTOR(0x30)`

向一个CPU发送这个消息，强制在这个CPU上执行`schedule()`函数。对应的中断服务例程（ISR）名为`smp_reschedule_interrupt()`。`reschedule_idle()`和`send_sig_info()`使用这个消息抢占正在CPU上运行的进程。

`INVALIDATE_TLB_VECTOR(0x31)`

向除自己之外的所有CPU发送这个消息，强制这些CPU对自己的转换后援缓冲区（TLB）进行初始化。对应的ISR名为`smp_invalidate_interrupt()`

(注8)，它会调用_flush_tlb()函数。内核每次对某个进程的页表进行修改时都会使用这个消息。

STOP_CPU_VECTOR(0x40)

向除自己之外的所有CPU发送这个消息，强制这些CPU停止。相应的ISR名为smp_stop_cpu_interrupt()。这个消息只有在内核检测到无法解决的内部错误时才使用。

LOCAL_TIMER_VECTOR(0x41)

I/O APIC把定时中断自动发给所有的CPU。相应的ISR名为smp_apic_timer_interrupt()。

CALL_FUNCTION_VECTOR(0x50)

向除自己之外的所有CPU发送这个消息，强制这些CPU运行自己所传递的函数。相应的ISR名为smp_call_function_interrupt()。这个消息的典型用法是强制CPU同步并重新装载内存类型范围寄存器（MTRR）的状态。从Pentium Pro开始，Intel微处理器就包含了这些附加寄存器来方便地定制高速缓存的操作。Linux使用这些寄存器在维护“合并写(write combining)”操作模式的同时，禁止PCI/AGP图形卡的显存进行地址映射时使用硬件高速缓存，在把数据拷贝到显存中之前，分页单元“合并写”机制首先将其转换成大块数据。

对Linux 2.4的展望

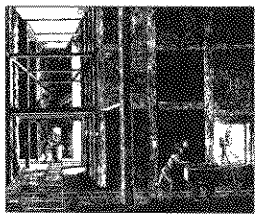
Linux 2.4对于信号量的实现方法进行了一点修改。从根本上来说，信号量现在是更高效了，因为在释放一个信号量时，通常只会唤醒一个正在睡眠的进程。

如前所述，Linux 2.4增强了对高端SMP体系结构的支持。现在可以使用多个外部I/O APIC芯片，所有处理处理器间中断（IPI）的代码都重写了。

但是，和Linux 2.2相比，Linux 2.4最大的改进在于它可以更好地支持多线程。换

注8：如果一些处理器关中断了，那么在试图刷新所有处理器的转换旁路缓冲区时会发生并发问题。因此，在内核控制路径在紧循环中旋转时，要检查是否有CPU已经发出了“无效TLB”消息。

言之，Linux 2.4采用了很多新的自旋锁，大大减少了全局内核锁的作用，特别是在网路部分的代码中更是如此。因此，Linux 2.4在SMP体系结构上效率更高，更适合做高端服务器。



第十二章
虚拟文件系统

Linux成功的关键因素之一是其具有兼容其它操作系统的能力。你能够透明地安装磁盘或分区，在这些磁盘或分区上可以驻留其它操作系统所用的文件格式，这些操作系统如Windows、其他版本的Unix、或者甚至像Amiga那样的小市场份额系统。Linux设法支持多种磁盘类型，其使用的方法与其他Unix变体相同，即所谓的虚拟文件系统的概念。

虚拟文件系统所隐含的思想是：在内核的内存中表示文件和文件系统的对象包含着广泛的信息。其中有一个域或函数，它能支持Linux支持的任何实际文件系统所提供的任何操作。对于所访问的每个读、写或其他的函数，内核都能把它们替换成实际的函数，这种实际的函数支持Linux的本地文件系统、NT文件系统，或者文件所在的任何其他文件系统。

本章讨论Linux虚拟文件系统的设计目标、结构及其实现。集中讨论五个Unix标准文件类型中的三个文件类型，即正规文件、目录文件和符号链接文件。设备文件将在第十三章中进行介绍。管道文件会在第十八章中进行讨论。为了进一步说明实际文件系统如何工作，在第十七章中对第二扩展文件系统(Second Extended Filesystem)进行讨论（几乎所有的Linux系统都使用了Ext2）。

虚拟文件系统的作用

虚拟文件系统（Virtual Filesystem）也可以称之为虚拟文件系统开关（Virtual

Filesystem Switch）或VFS，属于内核软件层，用来处理与Unix标准文件系统有关的所有系统调用。其强壮性表现在能为各种文件系统提供一个通用的接口。

例如：假设一个用户输入以下shell命令：

```
$ cp /floppy/TEST /tmp/test
```

其中*/floppy*是MS-DOS磁盘的一个安装点，而*/tmp*是一个标准的Ext2（第二扩展文件系统）的目录。正如图12-1(a)所示，VFS是用户的应用程序与文件系统实现之间的抽象层。因此，*cp*程序并不需要知道*/floppy/TEST*和*/tmp/test*的文件系统是什么类型．相反，*cp*程序直接与VFS进行交互，这是通过Unix程序设计人员都熟悉的一般系统调用来实现的（参见第一章中的“文件操作的系统调用”一节）。*cp*所执行的代码如图12-1(b)所示。

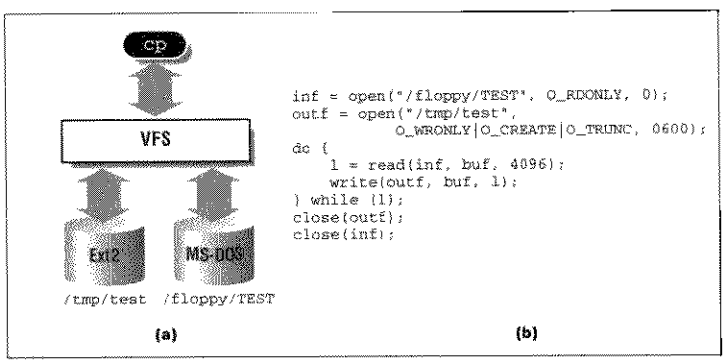


图12-1 VFS在一个简单的文件拷贝操作中的作用

VFS支持的文件系统可以划分为三种主要类型：

基于磁盘的文件系统

管理在本地磁盘分区中可用的存储空间。正式的基于磁盘的Linux文件系统是Ext2。VFS支持的基于磁盘的其他著名文件系统还有：

— Unix家族系统（如Unix System V及BSD）的文件系统

— 微软公司的文件系统，如MS-DOS、VFAT(Windows 98)及NTFS(Windows NT)

— ISO9660 CD-ROM 文件系统（以前的 High Sierra 文件系统）

— 其他有专利权的文件系统，如 HPFS（IBM 公司的 OS/2）、HFS（苹果公司的 Macintosh）、FFS（Amiga 公司的快速文件系统）以及 ADFS（Acorn 公司的系列机）

网络文件系统

允许轻易地访问属于其他网络计算机的文件系统所包含的文件。虚拟文件系统所支持的一些著名的网络文件系统有：NFS、Coda、AFS（Andrew 公司的文件系统）、SMB（微软公司的 Windows 和 IBM 公司的 OS/2 局域网管理器）以及 NCP（Novell 公司的 NetWare 内核协议）。

特殊文件系统（也称虚拟文件系统）

无须管理磁盘空间。Linux 的 */proc* 文件系统提供一个简单的接口，允许用户访问一些内核数据结构的内容。*/dev/pts* 文件系统用于支持伪终端（pseudo-terminal），这在开放组的 Unix98 标准中有详细说明。

由于篇幅所限，本书只在第十七章详细介绍 Ext2 文件系统，其他文件系统不做介绍。

正如第一章的“Unix 文件系统概述”一节中所提到的：Unix 的目录建立了一棵树，其根为目录“/”。根目录包含在根文件系统（root filesystem）中，在 Linux 中这个根文件系统通常就是 Ext2 类型。其他所有的文件系统都可以被“安装（mount）”在根文件系统的子目录中（注 1）。

基于磁盘的文件系统通常被存放在硬件块设备中（比如硬盘、软盘或者 CD-ROM）。Linux 虚拟文件系统一个有用的功能是能够处理虚拟块设备，如 */dev/loop0* 就可以用来安装存放正规文件的文件系统。作为一个可能的应用，用户可以保护自己的私有文件系统，这可以通过把自己文件系统的加密版本存放在一个正规文件中来实现。

第一个虚拟文件系统包含在 1986 年由 Sun 公司发布的 SunOS 操作系统中。从那时起，多数 Unix 文件系统都包括 VFS。然而，Linux 的虚拟文件系统对多种类型的文件系统提供更为广泛的支持。

注 1：当一个文件系统被安装在某一目录上时，在父文件系统中的目录内容不再能够访问，因为包含在安装点中的路径名将指向已安装的文件系统。但是，当被安装的文件系统卸载时，原目录的内容又可再现。这种令人惊讶的 Unix 文件系统特点可以由系统管理员用来隐藏文件，他们只需把一个文件系统安装在要隐藏文件的目录中即可。

通用文件模型

虚拟文件系统所隐含的主要思想在于引入了一个通用的文件模型，这个模型能够表示所有支持的文件系统。该模型严格遵守传统Unix文件系统提供的文件模型。这并不奇怪，因为Linux希望以最小的额外开销运行它本机上的文件系统。不过要实现每个具体的文件系统，必须将其物理组织结构转换为虚拟文件系统的通用文件模型。

例如，在通用文件模型中，每个目录被看作常规文件，可以包含若干文件和其他的子目录。但是，存在几个非Unix基于磁盘的文件系统，它们利用文件分配表（File Allocation Table，FAT）存放每个文件在目录树中的位置，在这些文件系统中，存放的是目录而不是文件。为了符合VFS的通用文件模型，对上述基于FAT的文件系统的实现，Linux必须在必要时能够快速建立对应于目录的文件。这样的文件只作为内核内存的对象而存在。

从本质上说，Linux的内核不能对一个特定的函数进行硬编码来执行诸如read()或ioctl()这样的操作，而是对每个操作都必须使用一个指针，这个指针指向要访问的具体文件系统的适当函数。

为了进一步说明这一概念，参见图12-1，其中显示了内核如何把read()转换为专对MS-DOS文件系统的一个系统调用。应用程序对read()的调用引起内核调用sys_read()，这完全与其他系统调用类似。我们在本章后面会看到，文件在内核中是由一个文件数据结构来表示的。这种数据结构中包含一个称为f_op的域，该域中包含一个指向MS-DOS文件的具体函数的指针，当然包括读文件的函数。sys_read()查找到指向该函数的指针，并调用它。这样一来，应用程序的read()就被转化为相对间接的调用：

```
file->f_op->read(...);
```

与之类似，write()操作也会引发一个适当的Ext2写函数的执行，当然这个写函数与输出文件相关。简而言之，内核负责把恰当的指针集合分配给每个打开文件的file变量，然后调用由f_op域指向的每个具体文件系统对应的系统调用。

你可以把通用文件模型看作是面向对象的，在这里，对象是一个软件结构，其中既定义了数据结构也定义了其上的操作方法。出于效率的考虑，Linux的编码并未采用面向对象的程序设计语言（比如C++）。因此对象作为数据结构来实现，数据结构中指向函数的域就对应于对象的方法。

通用文件模型由下列对象类型组成：

超级块对象（superblock object）

存放已安装文件系统的有关信息。对于基于磁盘的文件系统，这类对象通常对应于存放在磁盘上的文件系统控制块（filesystem control block）。

索引节点对象（inode object）

存放关于具体文件的一般信息。对于基于磁盘的文件系统，这类对象通常对应于存放在磁盘上的文件控制块（file control block）。每个索引节点对象都有一个索引节点号，这个号唯一地标识文件系统中的指定文件。

文件对象（file object）

存放打开文件与进程之间进行交互的有关信息。这类信息仅当进程访问文件期间存在于内核内存中。

目录项对象（dentry object）

存放目录项与对应文件进行链接的信息。每个基于磁盘的文件系统都以自己特有的方式将该类信息存在磁盘上。

图12-2显示一个简单的实例，说明进程怎样与文件进行交互。三个不同进程已打开同一个文件，其中两个进程使用同一个硬链接。在这种情况下，每个进程都使用自己的文件对象，但只需要两个目录项对象，每个硬链接对应一个目录项对象。这两个目录项对象指向同一个索引节点对象，这个索引节点对象标识的是超级块对象以及普通磁盘文件。

VFS除了能为所有文件系统的实现提供一个通用接口外，它还具有另一个重要的作用，即提高系统性能。最近最常使用的目录项对象被放在所谓目录项高速缓存（dentry cache）的磁盘高速缓存中，以加速从文件路径名到最后一个路径分量的索引节点的转换过程。

一般说来，磁盘高速缓存（disk cache）属于软件机制，它允许内核将原本存在磁盘上的某些信息驻留在RAM中，以便对这些数据的进一步访问能快速进行，而不必慢速访问磁盘本身（注2）。除了目录项高速缓存之外，Linux还使用其他磁盘高速缓存：比如缓冲区高速缓存、页高速缓存，我们将在后续章节进行介绍。

注2：注意，磁盘高速缓存不同于硬件高速缓存或内存高速缓存，后两者都与磁盘或其他设备无关。硬件高速缓存是一个静态快速RAM，它加快了直接对慢速动态RAM的请求（参见第二章中的“硬件高速缓存”一节）。内存高速缓存是一种软件机制，引入它是为了绕过内核内存分配器（参见第六章中的“slab分配器”一节）。

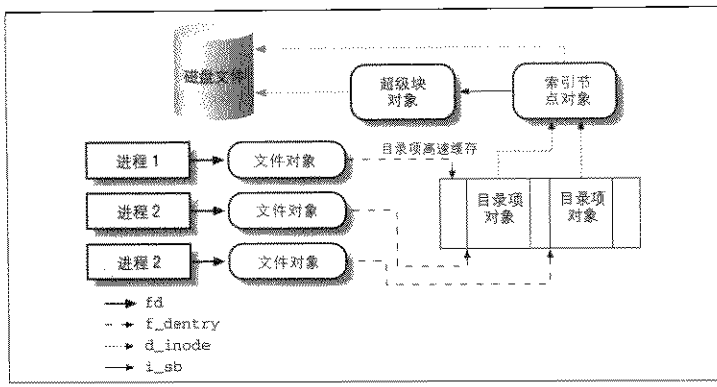


图 12-2 进程与 VFS 对象的交互

VFS 所处理的系统调用

表 12-1 列出了 VFS 的系统调用，这些系统调用涉及文件系统、正规文件、目录文件及符号链接文件。另外还有少数几个由 VFS 处理的其它系统调用：诸如 ioperm()、ioctl()、pipe()和 mknod()，涉及设备文件和管道文件，因此在后续章节进行讨论。最后一组由 VFS 处理的系统调用，诸如 socket()、connect()、bind()和 protocols()，属于套接字系统调用并用于实现网络功能，本书不予讨论。与表 12-1 列出的系统调用对应的内核服务例程，我们会在本章和第十七章中陆续进行讨论。

表 12-1 VFS 执行的部分系统调用

系统调用名	描述
mount() umount()	安装 / 卸载文件系统
sysfs()	获取文件系统信息
statfs() fstatfs() ustat()	获取文件系统统计信息
chroot()	更改根目录
chdir() fchdir() getcwd()	更改当前目录
mkdir() rmdir()	创建 / 删除目录
getdents() readdir() link() unlink() rename()	对目录项进行操作

表 12-1 VFS 执行的部分系统调用（续）

系统调用名	描述
readlink() symlink()	对软链接进行操作
chown() fchown() lchown()	更改文件所有者
chmod() fchmod() utime()	更改文件属性
stat() fstat() lstat() access()	读取文件状态
open() close() creat() umask()	打开 / 关闭文件
dup() dup2() fcntl()	对文件描述符进行操作
select() poll()	异步 I/O 通信
truncate() ftruncate()	更改文件长度
lseek() _llseek()	更改文件指针
read() write() readv() writev() sendfile()	文件 I/O 操作
pread() pwrite()	搜索并访问文件
mmap() munmap()	文件内存映射
fdatasync() fsync() sync() msync()	同步访问文件数据
flock()	处理文件锁

前面我们已经提到，VFS 是应用程序和具体的文件系统之间的一个层。不过，在某些情况下，一个文件操作可能由 VFS 本身去执行，无需调用下一层程序。例如，当某个进程关闭一个打开的文件时，并不需要涉及磁盘上的相应文件，因此，VFS 只需释放对应的文件对象。类似地，如果系统调用 lseek() 修改一个文件指针，而这个文件指针指向有关打开的文件与进程交互的一个属性，那么 VFS 只需修改对应的文件对象，而不必访问磁盘上的文件，因此，无需调用具体的文件系统子程序。从某种意义上说，可以把 VFS 看成“通用”文件系统，它在必要时依赖某种具体的文件系统。

VFS 的数据结构

每个 VFS 对象都存放在一个适当的数据结构中，其中包括对象的属性和指向对象方法表的指针。内核可以动态地修改对象的方法，因此可以为对象设置专用的行为。后续部分将要详细介绍 VFS 的对象及其内在关系。

超级块对象

超级块对象由 super_block 结构组成，表 12-2 列举了其中的域。

表 12-2 超级块对象的域

类型	域	描述
struct list_head	s_list	指向超级块链表的指针
kdev_t	s_dev	设备标识符
unsigned long	s_blocksize	以字节为单位的块大小
unsigned char	s_blocksize_bits	以位为单位的块大小
unsigned char	s_lock	锁标志
unsigned char	s_rd_only	只读标志
unsigned char	s_dirt	修改（脏）标志
struct file_system_type *	s_type	文件系统类型
struct super_operations *	s_op	超级块方法
struct dquot_operations *	dq_op	磁盘限额方法
unsigned long	s_flags	安装标志
unsigned long	s_magic	文件系统的魔数
unsigned long	s_time	最后一次修改超级块的时间
struct dentry *	s_root	安装目录的目录项对象
struct wait_queue *	s_wait	安装等待队列
struct inode *	s_ibasket	待开发
short int	s_ibasket_count	待开发
short int	s_ibasket_max	待开发
struct list_head	s_dirty	已修改的索引节点的链表
union	u	具体文件系统信息

所有超级块对象（每个安装的文件系统都有一个）都以循环双向链表的形式链接在一起。链表中第一个元素和最后一个元素的地址分别存放在 super_blocks 变量的 s_list 域的 next 和 prev 域中。s_list 域的数据类型为 struct list_head，在超级块的 s_dirty 域以及内核的其他很多地方都可以找到这样的数据类型，这种数据类型仅仅包括指向链表中的前一个元素和后一个元素的指针。因此，超级块对象

的s_list域包含指向链表中两个相邻超级块对象的指针。图12-3说明了list_head元素、next和prev是如何嵌入到超级块对象中的。

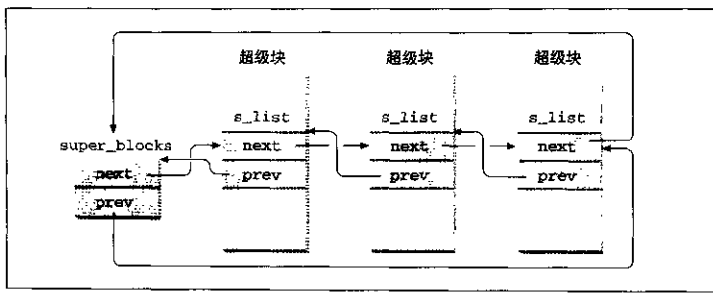


图12-3 超级块链表

最后一个u联合体域包括属于具体文件系统的超级块信息。例如，我们在第十七章会看到，假如超级块对象指的是Ext2文件系统，该域就存放ext2_sb_info数据结构，其中包括磁盘分配位屏蔽和其他与VFS的通用文件模型无关的的数据。

通常，为了效率起见u域的数据被复制到内存。任何基于磁盘的文件系统都需要访问和更改自己的磁盘分配位图，以便分配和释放磁盘块。VFS允许这些文件系统直接对内存超级块的u联合体域进行操作，无需访问磁盘。

但是，这种方法带来一个新问题：有可能VFS超级块最终不再与磁盘上相应的超级块同步。因此，有必要引入一个s_dirt标志，来表示该超级块是否是脏的，也就是说，磁盘上的数据是否必须要更新。缺乏同步还导致我们熟悉的一个问题：当一台机器的电源突然断开而用户来不及正常关闭系统时，就会出现文件系统崩溃。我们会在第十四章的“把脏缓冲区写入磁盘”部分看到，Linux是通过周期性地将所有“脏”的超级块写回磁盘来减少该问题带来的危害。

与超级块关联的方法就是所谓的超级块操作（superblock operation）。这些操作是由数据结构super_operations来描述的，该结构的起始地址存放在超级块的s_op域中。

每个具体的文件系统都可以定义自己的超级块操作。当VFS需要调用其中之一时，比如说read_inode()，VFS执行：

```
sb->s_op->read_inode(inode);
```

这里sb存放的是相关超级块对象的地址。super_operations表的read_inode域存放这一函数的地址，因此，这一函数被直接调用。

我们简要描述一下超级块操作，其中实现了一些高级操作，比如删除文件或安装磁盘。按照这些操作在super_operation表中出现的顺序来对它们进行排列：

read_inode(inode)

用磁盘上的数据填充参数指定的索引节点对象的域。该索引节点对象的i_ino域标识从磁盘上要读取的具体文件系统的索引节点。

write_inode(inode)

用参数指定的索引节点对象的内容更新文件系统的一个索引节点。索引节点的i_ino域标识指定磁盘上文件系统的索引节点。

put_inode(inode)

释放参数指定的索引节点对象。释放一个对象照样并不意味着释放内存，因为其他进程可能仍然在使用这个对象。

delete_inode(inode)

删除那些包含文件、磁盘索引节点及VFS索引节点的数据块。

notify_change(dentry, iattr)

依照参数iattr的值修改索引节点的一些属性。如果notify_change域为空，VFS转而执行write_inode()。

put_super(super)

释放超级块对象，超级块的地址通过参数传递（因为相应的文件系统没有被安装）。

write_super(super)

用指定对象的内容更新文件系统的超级块。

statfs(super, buf, bufsize)

将文件系统的统计信息返回，填写在buf缓冲区中。

remount_fs(super, flags, data)

用新的选项重新安装文件系统（当某个安装选项必须被修改时进行调用）。

clear_inode(inode)

与put_inode类似，但同时也释放指定索引节点对应的文件所包含的数据占用的所有页。

umount_begin(super)

中断一个安装操作。因为相应的解除安装操作已经启动（只在网络文件系统中使用）。

前述的方法对所有可能的文件系统类型均是可用的。但是，其中的子集只应用到每个具体的文件系统；未实现的方法对应的域被置为NULL。注意，系统没有定义read_super方法来读超级块，那么，内核如何能够调用一个对象的方法而从磁盘读出该对象？我们将在另一个文件系统类型的对象中找到read_super方法（参见后面的“文件系统安装”一节）。

索引节点对象

文件系统处理文件所需要的所有信息都放在一个称为索引节点的数据结构中。文件名可以随时更改，但是索引节点对文件是唯一的，并且随文件存在而存在。内存中的索引节点对象由一个inode数据结构组成，其域如表12-3所描述。

表12-3 索引节点对象的域

类型	域名	描述
struct list_head	i_hash	指向散列链表的指针
struct list_head	i_list	指向索引节点链表的指针
struct list_head	i_dentry	指向目录项链表的指针
unsigned long	i_ino	索引节点号
unsigned int	i_count	引用计数器
kdev_t	i_dev	设备标识符
umode_t	i_mode	文件类型与访问权限
nlink_t	i_nlink	硬链接数目
uid_t	i_uid	所有者标识符

表 12-3 索引节点对象的域（续）

类型	域名	描述
gid_t	i_gid	组标识符
kdev_t	i_rdev	实设备标识符
off_t	i_size	文件的字节数
time_t	i_atime	上次访问文件的时间
time_t	i_mtime	上次写文件的时间
time_t	i_ctime	上次修改索引节点的时间
unsigned long	i_blksize	块的字节数
unsigned long	i_blocks	文件的块数
unsigned long	i_version	版本号（每次使用后自动递增）
unsigned long	i_nrpages	包含文件数据的页数
struct semaphore	i_sem	索引节点信号量
struct semaphore	i_atomic_write	自动写的索引节点信号量
struct inode_operations *	i_op	索引节点的操作
struct super_block *	i_sb	指向超级块对象的指针
struct wait_queue *	i_wait	索引节点等待队列
struct file_lock *	i_flock	指向文件锁链表的指针
struct vm_area_struct *	i_mmap	指向对文件进行映射所使用的线性区的指针
struct page *	i_pages	指向页描述符的指针
struct dquot **	i_dquot	索引节点的磁盘限额
unsigned long	i_state	索引节点的状态标志
unsigned int	i_flags	文件系统的安装标志
unsigned char	i_pipe	如果是管道文件则为真
unsigned char	i_sock	如果是套接字文件则为真
int	i_writecount	写进程的引用计数器
unsigned int	i_attr_flags	文件创建标志
__u32	i_generation	为以后的开发保留
union	u	具体文件系统的信息

最后一个联合域u用于存放属于具体文件系统的索引节点信息。例如，在第十七章我们会了解到，如果索引节点对象指的是一个Ext2文件，该域就存放一个名为ext2_inode_info的数据结构。

每个索引节点对象都会复制磁盘索引节点包含的一些数据，比如文件占用的磁盘块数。如果i_state域的值等于I_DIRTY，该索引节点就是“脏”的，也就是说，对应的磁盘索引节点必须被更新。i_state域的其他值有I_LOCK（这意味着该索引节点对象已加锁），I_FREEING（这意味着该索引节点对象正在被释放）。

每个索引节点对象总是出现在下列循环双向链表的某个链表中：

- 未使用索引节点链表。变量inode_unused的next域和prev域分别指向该链表中的首元素和尾元素。这个链表用做内存高速缓存。
- 正在使用索引节点链表。变量inode_in_use指向该链表中的首元素和尾元素。
- 脏索引节点链表。由相应超级块对象的s_dirty域指向该链表中的首元素和尾元素。

这些链表都是通过适当的索引节点对象的域i_list链接在一起的。

属于“正在使用”或“脏”链表的索引节点对象也同时存放在一个称为inode_hashtable的散列表中。散列表加快了对索引节点对象的搜索，前提是系统内核要知道索引节点号及对应文件所在文件系统的超级块对象的地址（注3）。由于散列技术可能引发冲突，所以，索引节点对象设置一个i_hash域，其中包含向前和向后的两个指针，分别指向散列到同一地址的前一个索引节点和后一个索引节点，该域因此创建了由这些索引节点组成的一个双向链表。

与索引节点对象关联的方法也叫索引节点操作（inode operation）。由inode_operations结构来描述，该结构的地址存放在i_op域中，该结构也包括一个指向文件操作方法的指针（参见下一节“文件对象”）。以下是索引节点的操作，以它们在表inode_operations中出现的次序来排列：

注3：实际上，Unix的进程可以打开一个文件，然后又可以解开与该文件的链接，索引节点的i_nlink域可能变为0，但是这个进程还能作用于该文件。在这种特殊情况下，就要从散列表中删除这个索引节点，即使这个索引节点还属于正在使用的链表或脏链表。

create(dir, dentry, mode)

在dir目录下，为与dentry目录项相关的正规文件创建一个新的磁盘索引节点。

lookup(dir, dentry)

查找一个索引节点所在的目录，这个索引节点所对应的文件名就包含在dentry目录项对象中。

link(old_dentry, dir, new_dentry)

创建一个新的名为new_dentryr的硬链接，这个新的硬连接指向dir目录下名为old_dentryr的文件。

unlink(dir, dentry)

从dir目录删除dentry目录项对象所指文件的硬链接。

symlink(dir, dentry, symname)

在某个目录下，为与目录项对象相关的符号链创建一个新的索引节点。

mkdir(dir, dentry, mode)

在某个目录下，为与目录项对象相关的目录创建一个新的索引节点。

rmdir(dir, dentry)

从dir目录删除一个子目录，子目录的名字包含在目录项对象中。

mknod(dir, dentry, mode, rdev)

在dir目录下，为与目录项对象相关的特殊文件创建一个新的磁盘索引节点。其中参数mode和rdev分别表示文件的类型和该设备的主码。

rename(old_dir, old_dentry, new_dir, new_dentry)

将old_dir目录下的文件old_dentry移到new_dir目录下，新文件名包含在new_dentry指向的目录项对象中。

readlink(dentry, buffer, buflen)

将dentry所指定的符号链中对应的文件路径名拷贝到buffer所指定的内存区。

follow_link(inode, dir)

解释inode对象索引节点所指定的符号链。如果该符号链是相对路径名，从指定的dir目录开始进行查找。

readpage(file, pg)

从一个打开的文件中读出一个数据页。我们会在第十五章中看到，正规文件使用这一方法进行读。

writepage(file, pg)

将一个数据页写入一个打开的文件中。多数文件系统进行正规文件写入操作时并不使用这一方法。

bmap(inode, block)

返回索引节点inode所指文件的磁盘块号对应的逻辑块号。

truncate(inode)

修改索引节点inode所指文件的长度。在调用该方法之前，必须将inode对象的i_size域设置为需要的新长度值。

permission(inode, mask)

确认是否允许对inode索引节点所指的文件进行指定模式的访问。

smap(inode, sector)

与bmap()类似，但确定的是磁盘扇区号，由基于FAT的文件系统使用。

updatepage(inode, pg, buf, offset, count, sync)

如果需要，更新索引节点inode所指文件的数据页(通常由网络文件系统调用，更新远程文件之前，可能需要等待较长时间)。

revalidate(dentry)

更新由目录项对象所指定文件的已缓存的属性(通常由网络文件系统调用)。

上述列举的方法对所有可能的索引节点和文件系统类型都是可用的。不过，只有其中的一个子集应用到任一特定的索引节点和文件系统，未实现的方法对应的域被置为NULL。

文件对象

文件对象描述的是进程怎样与一个打开文件交互的过程。文件对象是在文件被打开时创建的，由一个file结构组成，其中包含的域如表12-4所示。注意，文件对象在磁盘上没有对应的映像，因此，file结构中没有设置“脏”域来表示文件对象是否已被修改。

存放在文件对象中的主要信息是文件指针，即文件中当前操作的位置。由于几个进程可能并发访问同一个文件，因此文件指针不能存放在索引节点对象中。

表 12-4 文件对象的域

域	类型	描述
struct file *	f_next	指向下一个文件对象的指针
struct file **	f_pprev	指向前一个文件对象的指针
struct dentry *	f_dentry	指向相关目录项对象的指针
struct file_operations *	f_op	指向文件操作表的指针
mode_t	f_mode	进程的访问模式
loff_t	f_pos	文件当前的位移量（文件指针）
unsigned int	f_count	文件对象的引用计数器
unsigned int	f_flags	当打开文件时所指定的标志
unsigned long	f_reada	预读标志
unsigned long	f_ramax	要预读的最多页数
unsigned long	f_raend	上次预读后的文件指针
unsigned long	f_ralen	预读的字节数
unsigned long	f_rawin	预读的页数
struct fown_struct	f_owner	通过信号进行异步I/O数据的传送
unsigned int	f_uid	用户的 UID
unsigned int	f_gid	用户的 GID
int	f_error	网络写操作的错误码
unsigned long	f_version	版本号（每次使用后自动递增）
void *	private_data	tty 驱动程序所需

每个文件对象总是包含在下列的一个循环双向链表之中：

- “未使用”文件对象的链表。该链表既可以用作文件对象的内存高速缓存，又可以当作超级用户的备用存储器，即使系统的动态内存被用完，也允许超级用户打开文件。由于这些对象是未使用的，它们的 f_count 域是 NULL，该链表首元素的地址存放在变量 free_filps 中，内核必须确认该链表总是至少包含 NR_RESERVED_FILES 个对象，通常该值设为 10。
- “正在使用”文件对象的链表。该链表中的每个元素至少由一个进程使用，因

此，各个元素的f_count域不会为NULL，该链表中第一个元素的地址存放在变量inuse_filps中。

不论文件对象当前属于哪个链表，它的f_next域都指向所在链表的下一个元素，而f_pprev域则指向上一个元素的f_next域。

“未使用”文件对象链表的长度存放在变量nr_free_files中。如果VFS需要分配一个新的文件对象，就调用函数get_empty_filp()。该函数检测“未使用”文件对象链表的元素个数是否多于NR_RESERVED_FILES，如果是，新打开的文件可以使用其中的一个元素；如果没有，则退回到正常的内存分配。

正如在“通用文件模型”一节中讨论过的一样，每个文件系统都有自己的文件操作（file operation）集合，执行诸如读写文件的操作。当内核将一个索引节点从磁盘装入内存时，会在file_operations结构中存放一个指向这些文件操作的指针，该结构的地址存放在该索引节点对象的inode_operations结构的default_file_ops域中。当进程打开这个文件时，VFS就用存放在索引节点中的这个地址初始化新文件对象的f_op域，使得对文件操作的后续调用能够使用这些函数。如果需要，VFS随后也可以通过在f_op域存放一个新值而修改这一文件操作的集合。

下面列表描述了文件的操作，以它们在file_operations表中出现的次序来排列：

llseek(file, offset, whence)

修改文件指针。

read(file, buf, count, offset)

从文件的offset处开始读出count个字节，然后增加*offset的值（一般与文件指针对应）。

write(file, buf, count, offset)

从文件的*offset处开始写入count个字节，然后增加*offset的值（一般与文件指针对应）。

readdir(dir, dirent, filldir)

返回dir所指目录的下一个目录项，这个值存入参数dirent。参数filldir存放一个辅助函数的地址，该函数可以提取目录项的各个域。

poll(file, poll_table)

检查是否存在关于某文件的操作事件，如果没有则睡眠，直到发生该类操作事件为止。

ioctl(inode, file, cmd, arg)

向一个基本硬件设备发送命令。该方法只适用于设备文件。

mmap(file, vma)

执行文件的内存映射，并将这个映射放入进程的地址空间。(参见第十五章中的“内存映射”一节)。

open(inode, file)

通过创建一个新的文件对象而打开一个文件，并把它链接到相应的索引节点对象（参见本章后面的“open()系统调用”一节)。

flush(file)

当关闭对一个打开文件的引用时，就调用该方法，也就是说，减少该文件对象f_count域的值。该方法的实际用途是依赖于文件系统的。

release(inode, file)

释放文件对象。当关闭对打开文件的最后一个引用时，调用该方法，也就是说，该文件对象f_count域的值变为0。

fsync(file, dentry)

将file文件在高速缓存中的全部数据写入磁盘。

fasync(file, on)

通过信号来启用或禁用异步I/O通告。

check_media_change(dev)

检测自上次对设备文件操作以来是否存在介质的改变(可以对块设备使用这一方法，因为它支持可移动介质，比如软盘和CD-ROM)。

revalidate(dev)

恢复设备的一致性(由网络文件系统使用，这是在确认某个远程设备上的介质已被改变之后才使用)。

lock(file, cmd, file_lock)

对file文件申请一个锁（参见本章后面的“文件加锁”一节)。

以上描述的方法对所有可能的文件类型都是可用的。不过，对于一个具体的文件类型，只使用其中的一个子集，那些未实现的方法对应的域被置为NULL。

对目录文件对象的特殊处理

对目录的处理必须特别当心，因为可能存在几个进程同时改变其内容的情况。对正规文件频繁进行的显式加锁（参见本章后面的“文件加锁”一节）就不适用于目录，因为这样一来，就阻止其他进程对加锁目录下的所有子目录进行访问。因此，将文件对象的f_version域和索引节点对象的i_version域结合使用，就可以确保对每个目录文件访问的一致性。

我们通过描述最普通的操作［如系统调用readdir()］所需要的域来对这些域进行解释。假定该系统调用的每次调用都返回一个目录项，并更新该目录文件的指针，以使该系统调用的下一次执行返回下一个目录项。但是，该目录可能由其他并发访问它的进程进行了修改。如果不进行某种一致性检查，readdir()系统调用可能返回错误的目录项。在进程调用readdir()和进程随时选择停止调用它之间，可能已经过去了较长的时间段（潜在时段），因此我们不想锁住该目录。我们想到的一种方式是让readdir()适应这一改变。

问题的解决从global_event变量入手，这个变量起一个版本标记的作用。只要某目录文件的索引节点对象被修改，变量global_event的值就增1，并把这个新的版本标记值存放在该对象的i_version域中。只要一个新的文件对象被创建或者其文件指针被修改，global_event变量的值就增1，并把这个新的版本标记值存放在该对象的f_version域中。这样一来，当VFS处理readdir()系统调用时，首先检查存放在该对象i_version域中的版本标记值与f_version域中的版本标记值是否一致。如果不一致，说明该目录可能在上一次执行readdir()后被其他进程修改过。

当readdir()系统调用检测到这种不一致性问题时，就通过再次读整个目录的内容来重新计算该目录的文件指针。该系统调用返回进程上次调用readdir()时返回的目录项的下一个目录项，并将f_version域的值置为i_version域的值，以表明readdir()系统调用现在与该目录的实际状态是同步的。

目录项对象

在“通用文件模型”一节我们提到，VFS把每个目录看作一个由若干子目录和文件组成的普通文件。在第十七章我们会讨论如何对具体的文件系统实现这些目录。然而，一旦目录项被读入内存，VFS就把它转换为基于dentry结构的一个目录项对象，该结构的域如表12-5所示。对于进程查找的路径名中的每个分量，内核都为其创建一个目录项对象。目录项对象将每个分量与其对应的索引节点相联系。例如，在查找路径名*/tmp/test*时，内核为根目录“/”创建一个目录项对象，为根目录下的*tmp*项创建一个第二级目录项对象，为*/tmp*目录下的*test*项创建一个第三级目录项对象。

请注意，目录项对象在磁盘上并没有对应的映像，因此在dentry结构中不包含“该对象已被修改”的域。目录项对象存放在称为dentry_cache的slab分配器高速缓存中。因此，目录项对象的创建和删除是通过调用kmem_cache_alloc()和kmem_cache_free()实现的。

表12-5 目录项对象的域

类型	域	描述
int	d_count	目录项对象引用计数器
unsigned int	d_flags	目录项标志
struct inode *	d_inode	与文件名关联的索引节点
struct dentry *	d_parent	父目录的目录项对象
struct dentry *	d_mounts	对于安装点而言，表示被安装文件系统根项
struct dentry *	d_covers	对文件系统的根而言，表示安装点的目录项
struct list_head	d_hash	散列表表项的指针
struct list_head	d_lru	未使用链表的指针
struct list_head	d_child	父目录中目录项对象的链表的指针
struct list_head	d_subdirs	对目录而言，表示子目录目录项对象的链表
struct list_head	d_alias	相关索引节点（别名）的链表
struct qstr	d_name	文件名

表 12-5 目录项对象的域（续）

类型	域	描述
unsigned long	d_time	由 d_revalidate 方法使用
structdentry_operations*	d_op	目录项方法
struct super_block *	d_sb	文件的超级块对象
unsigned long	d_reftime	目录项被删除的时间
void *	d_fsdata	与文件系统相关的数据
unsigned char	d_iname[16]	存放短文件名

每个目录项对象属于以下四种状态之一：

空闲状态（free）

处于该状态的目录项对象不包含有效的信息，还没有被VFS使用。它对应的内存区由slab分配器进行管理。

未使用状态（unused）

处于该状态的目录项对象当前还没有被内核使用。该对象的引用计数器d_count的值为NULL。但其d_inode域仍然指向相关的索引节点。该目录项对象包含有效的信息，但为了在必要时回收内存，它的内容可能被丢弃。

正在使用状态（inuse）

处于该状态的目录项对象当前正在被内核使用。该对象的引用计数器d_count的值为正数，而其d_inode域指向相关的索引节点对象。该目录项对象包含有效的信息，并且不能被丢弃。

负状态（negative）

与目录项相关的索引节点不复存在，那是因为相应的磁盘索引节点已被删除。该目录项对象的d_inode域被置为NULL，但该对象仍然被保存在目录项高速缓存中，以便后续对同一文件目录名的查找操作能够快速完成，术语“negative”容易使人误解，因为根本不涉及任何负值。

目录项高速缓存

由于从磁盘读入一个目录项并构造相应的目录项对象需要花费大量的时间，所以，在完成对目录项对象的操作后，可能后面还要使用它，因此在内存仍保留它有重要

的意义。例如，我们经常需要编辑文件，随后进行编译或编辑，然后打印或拷贝，再进行编辑。诸如此类的情况中，同一个文件需要被反复访问。

为了最大限度地提高处理这些目录项对象的效率，Linux使用目录项高速缓存，它由两种类型的数据结构组成：

- 处于正在使用、未使用或负状态的目录项对象的集合。
- 一个散列表，从中能够快速获取与给定的文件名和目录名对应的目录项对象。如果访问的对象不在目录项高速缓存中，散列函数返回一个空值。

目录项高速缓存的作用也相当于索引节点高速缓存（inode cache）的控制器。内核内存中，与未使用目录项相关的索引节点未被丢弃，这是由于目录项高速缓存仍在使用它们，因此，它们的i_count域不为空。因此，这些索引节点对象保存在RAM中，并能够借助相应的目录项快速引用它们。

所有“未使用”目录项对象都存放在一个“最近最少使用（LRU: Least Recently Used）”的双向链表中，该链表按照插入的时间排序。换句话说，最后释放的目录项对象放在链表的首部，所以最近最少使用的目录项对象总是靠近链表的尾部。一旦目录项高速缓存的空间开始变小，内核就从链表的尾部删除元素，使得多数最近经常使用的对象得以保留。LRU链表的首元素和尾元素的地址存放在变量dentry_unused中的next域和prev域中。目录项对象的d_lru域包含的指针指向该链表中相邻目录的对象。

每个“正在使用”的目录项对象都被插入一个双向链表中，该链表由相应索引节点对象的i_dentry域所指向（由于每个索引节点可能与若干硬连接关联，所以需要一个链表）。目录项对象的d_alias域存放链表中相邻元素的地址。这两个域的类型都是struct list_head。

当指向相应文件的最后一个硬链接被删除后，一个“正在使用”的目录项对象可能变成“负”状态。在这种情况下，该目录项对象被移到“未使用”目录项对象组成的LRU链表中。每当内核缩减目录项高速缓存时，“负”状态目录项对象就朝着LRU链表的尾部移动，这样一来，这些对象就逐渐被释放（参见第十六章中的“从目录项高速缓存和索引节点高速缓存中回收页”一节）。

散列表是由dentry_hashtable数组实现的。数组中的每个元素是一个指向链表的

指针，这种链表就是把具有相同散列表值的目录项进行散列而形成的。该数组的长度取决于系统已安装RAM的数量。目录项对象的d_hash域包含指向具有相同散列值的链表中的相邻元素。散列函数产生的值是由目录及文件名的目录项对象的地址计算出来的。

与目录项对象关联的方法称为目录项操作（dentry operation）。由dentry_operations结构加以描述，该结构的地址存放在目录项对象的d_op域中。尽管一些文件系统定义了它们自己的目录项方法，但是一些域通常为NULL，VFS就使用缺省函数代替这些方法。以下按照它们在dentry_operations表出现的顺序列举一些方法。

d_revalidate(dentry)

为了转换一个文件路径名而使用目录项对象前，判定该目录项对象是否仍然有效。缺省的VFS函数什么也不做，而网络文件系统可以指定自己的函数。

d_hash(dentry, hash)

生成一个散列值；对目录项散列表而言，这是一个具体文件系统的散列函数。参数dentry标识包含该路径分量的目录。参数hash指向一个结构，该结构包含要查找的路径名分量以及由散列函数生成的散列值。

d_compare(dir, name1, name2)

比较两个文件名。name1应该属于dir所指目录。缺省的VFS函数是常用的字符串匹配函数。不过，每个文件系统可用自己的方式实现这一方法。例如，MS-DOS文件系统不区分大写和小写字母。

d_delete(dentry)

如果对目录项对象的最后一个引用被删除（d_count变为"0"），就调用该方法。缺省的VFS函数什么也不做。

d_release(dentry)

当要释放一个目录项对象时（放入slab分配器），就调用该方法。缺省的VFS函数什么也不做。

d_iput(dentry, ino)

当一个目录项对象变为"负"状态（即丢弃它的索引节点），就调用该方法。缺省的VFS函数调用iput()释放索引节点对象。

与进程相关的文件

在第一章的“Unix文件系统概述”一节我们提到，每个进程都有它自己当前的工作目录和它自己的根目录。该信息存放在一个类型为fs_struct的内核表中，该表的地址包含在进程描述符的fs域中。

```
struct fs_struct {
    atomic_t count;
    int umask;
    struct dentry * root, * pwd;
};
```

count域表示共享同一fs_struct表的进程的数目[参见第三章中的“clone(), fork()及vfork()系统调用”一节]。umask域由umask()系统调用使用，用于为新创建的文件设置初始文件许可权。

第二个表表示进程当前打开的文件，表的地址存放在进程描述符的files域。该表的类型为files_struct结构，它的各个域及其说明列在表12-6中。一个进程不能拥有多于NR_OPEN(通常为1024)个文件描述符。对于允许打开文件的最大数，也可以定义一个更小的、动态的界限值，通过改变进程描述符中的rlim[RLIMIT_NOFILE]结构的值即可。

表12-6 files_struct结构的域

类型	域	描述
int	count	共享该表的进程数目
int	max_fds	当前文件对象的最大数目
int	max_fdset	当前文件描述符的最大数目
int	next_fd	所分配的最大文件描述符加1
struct file **	fd	指向文件对象指针数组的指针
fd_set *	close_on_exec	指向执行exec()时需要关闭的文件描述符
fd_set *	open_fds	指向打开文件描述符的指针
fd_set	close_on_exec_init	执行exec()时需要关闭的文件描述符的初值集合
fd_set	open_fds_init	文件描述符的初值集合
struct file *	fd_array[32]	文件对象指针的初始化数组

fd域指向文件对象的指针数组。该数组的长度存放在max_fds域中。通常，fd域指向files_struct结构的fd_array域，该域包括32个文件对象指针。如果进程打开的文件数目多于32，内核就分配一个新的、更大的文件指针数组，并将其地址存放在fd域中。内核同时也更新max_fds域的值。

对于在fd数组中有入口地址的每个文件来说，数组的索引就是文件描述符（file descriptor）。通常，数组的第一个元素（索引为0）是进程的标准输入文件，数组的第二个元素（索引为1）是进程的标准输出文件，数组的第三个元素（索引为2）是进程的标准错误文件（参见图12-4）。Unix进程将文件描述符作为主文件标识符。请注意，借助于dup()、dup2()和fcntl()系统调用，两个文件描述符就可以指向同一个打开的文件，也就是说，数组的两个元素可能指向同一个文件对象。当用户使用shell结构（如2>&1）将标准错误文件重定向到标准输出文件上时，用户总能看到这一点。

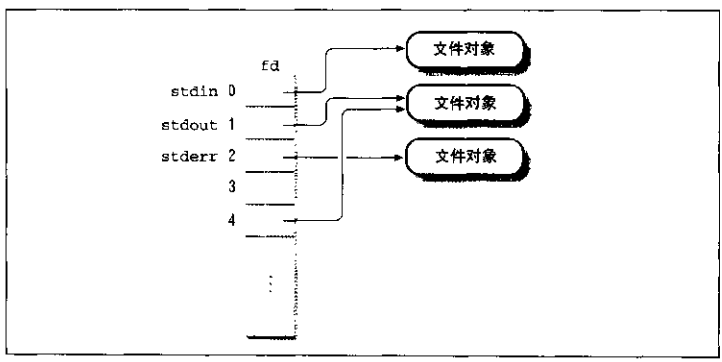


图12-4 fd数组

open_fds域包含open_fds_init域的地址，open_fds_init域表示当前已打开文件的文件描述符的位图。max_fdset域存放位图中的位数。由于数据结构fd_set有1024位，通常不需要扩大位图的大小。不过，如果确实必须的话，内核仍能动态增加位图的大小，这非常类似于文件对象的数组的情形。

当开始使用一个文件对象时调用内核提供的fget()函数。这个函数接收文件描述符fd作为参数，返回在current->files->fd[fd]中的地址，即对应文件对象的地

址，如果没有任何文件与 fd 对应，则返回 NULL。在第一种情况下，fget()使文件对象引用计数器 f_count 的值增 1。

当内核控制路径完成对文件对象的使用时，调用内核提供的 fput()函数。该函数将文件对象的地址作为参数，并递减文件对象引用计数器 f_count 的值，另外，如果这个域变为 NULL，该函数就调用文件操作的“释放”方法（如果已定义），释放相应的目录项对象，并递减对应索引节点对象的 i_writeaccess 域的值（如果该文件是写打开），最后，将该文件对象从“正在使用”链表移到“未使用”链表。

文件系统安装

现在，我们集中讨论VFS怎样跟踪它所支持的文件系统。在使用一个文件系统之前，必须执行两个基本的操作：注册和安装。

或者在系统启动时，或者在安装某个文件系统的模块时，都需要进行注册。一旦一个文件系统完成注册，那么它具体的函数对内核就是可用的了。因此，这个文件系统就可以安装在系统的目录树上。

每个文件系统都有它自己的根目录，如果某文件系统的根目录是系统目录树的根，那么该文件系统称为根文件系统。而其他文件系统可以安装到系统的目录树上，把这些文件系统要插入的那些目录就称为安装点（mount point）。

文件系统的注册

通常，用户在为自己的系统编译内核时可以把Linux配置为能够识别所有需要的文件系统。但是，文件系统的源代码实际上要么包含在内核映像中，要么作为一个模块被动态装入（参见附录二）。VFS必须对其代码在内核映像的所有文件系统进行跟踪。执行文件系统注册就实现了这一目标。

每个注册的文件系统都用一个类型为 file_system_type 的对象来表示，该对象的所有域在表12-7中列出。所有具有同一文件系统类型的对象都被插入一个简单的链表中，由变量 file_systems 指向链表的第一项。

表 12-7 file_system_type 对象的域

类型	域	描述
const char *	name	文件系统名
int	fs_flag	安装标志
struct super_block *(*)()	read_super	读出超级块的方法
struct file_system_type *	next	指向下一个链表元素的指针

在系统初始化期间，调用函数filesystem_setup()来注册编译时指定的文件系统。对于每个不同类型的文件系统，以指向file_system_type类型的某个对象作为参数来调用register_filesystem()函数，因此，该对象就被插入到文件系统类型的链表中。

当某个文件系统的模块被装入时，也要调用register_filesystem()函数。在这种情况下，当该模块被卸载时，对应的文件系统也可以被注销（调用unregister_filesystem()函数）。

get_fs_type()函数（接受一个文件系统名作为它的参数）扫描已注册的文件系统链表，并返回指向相应的file_system_type对象的指针（如果存在）。

安装根文件系统

安装根文件系统是系统初始化的关键部分。当系统启动时，就要在变量ROOT_DEV中寻找包含根文件系统的磁盘主码。当编译内核或向最初的启动装入程序传递一个合适的选项时，根文件系统可以被指定为*/dev*目录下的一个设备文件。类似地，根文件系统的安装标志存放在root_mountflags变量中。用户可以指定这些标志，这是通过对已编译的内核映像使用*/sbin/rdev*外部程序，或者向最初的启动装入程序传递一个合适的选项来达到的（参看附录一）。

在系统初始化期间，正好在filesystem_setup()调用之后执行mount_root()函数。该函数执行下列操作（假定被安装的文件系统是基于磁盘的文件系统，注4）：

注4：无磁盘工作站可以通过基于网络的文件系统（如NFS）安装根目录，但是我们在此不打算描述。

1. 初始化一个哑元（dummy）局部文件对象 filp。根据根文件系统的安装标志设置 f_mode 域，而其他域均被置为0。
2. 通过把 i_rdev 域置为 ROOT_DEV，创建一个哑元索引节点对象。
3. 调用 blkdev_open() 函数，传递的参数为这个哑元索引节点和哑元文件对象。我们在第十三章会看到，该函数还检查磁盘是否存在，是否正常地工作。
4. 释放哑元索引节点对象，因为它仅仅被用来验证磁盘的存在。
5. 扫描文件系统类型链表。对于每个 file_system_type 对象，调用 read_super() 试图读取相应的超级块。这个函数检查出设备还没有被安装，并用 file_system_type 对象的 read_super 域所指向的方法填充超级块对象。因为每个具体文件系统的方法都使用唯一的魔数，因此所有文件系统对 read_super() 的调用都将失败，除了试图填充超级块的根文件系统对这个方法的调用。read_super() 方法也为根目录创建一个索引节点对象和一个目录项对象。目录项对象把“/”映射到索引节点对象。
6. 把 current（*init* 进程）的 fs_struct 表的 pwd 域置为根目录的目录项对象。
7. 调用 add_vfsmnt() 把第一个元素插入到已安装文件系统的链表中（参看下一节）。

安装一个普通的文件系统

一旦完成对根文件系统的初始化，就可以安装其他的文件系统。其中的每一个都有自己的安装点，安装点仅仅是系统目录树中现有的一个目录。

所有已安装的文件系统都包含在一个链表中，链表的第一个元素由 vfsmntlist 变量所指向。每个元素都是类型为 vfsmount 的一个结构，其中的域如表 12-8 所示。

表 12-8 vfsmount 数据结构的域

类型	域	描述
kdev_t	mnt_dev	设备号
char *	mnt_devname	设备名
char *	mnt_dirname	安装点
unsigned int	mnt_flags	设备标志

表 12-8 vfsmount 数据结构的域（续）

类型	域	描述
struct super_block *	mnt_sb	超级块指针
quota_mount_options	mnt_dquot	磁盘限额安装点
struct vfsmount *	mnt_next	指向下一个链表元素的指针

三个低级的函数用来处理链表，并由 mount()和umount()系统调用的服务例程来调用。add_vfsmnt()和remove_vfsmnt()函数分别在链表中增加和删除一个元素。lookup_vfsmnt()函数搜索一个指定的已安装的文件系统，并返回相应 vfsmount 数据结构的地址。

mount()系统调用用来安装一个文件系统。它的服务例程 sys_mount()作用于以下参数：

- 包含文件系统的设备文件的路径名，或者如果不需要的话就为空（例如，当要安装的文件系统是基于网络时）
- 文件系统被安装的某个目录的目录路径名（安装点）
- 文件系统的类型，必须是已注册文件系统的名字
- 安装标志（所允许的值如表 12-9 所示）
- 指向一个与文件系统相关的数据结构的指针（也许为 NULL）

表 12-9 文件系统的安装选项

宏	值	描述
MS_MANDLOCK	0x040	所允许的强制锁
MS_NOATIME	0x400	不更新文件的访问时间
MS_NODEV	0x004	禁止对设备文件访问
MS_NODIRATIME	0x800	不更新目录的访问时间
MS_NOEXEC	0x008	不允许程序执行
MS_NOSUID	0x002	忽略 *setuid* 和 *setgid* 标志
MS_RDONLY	0x001	文件只能被读
MS_REMOUNT	0x020	重新安装文件系统

表 12-9 文件系统的安装选项（续）

宏	值	描述
MS_SYNCHRONOUS	0x010	立即执行写操作
S_APPEND	0x100	只允许追加文件
S_IMMUTABLE	0x200	允许不可变的文件
S_QUOTA	0x080	初始化磁盘限额

sys_mount()函数执行下列操作：

1. 检查进程是否具有安装一个文件系统所需要的能力。
2. 如果指定了MS_REMOUNT安装选项，则调用do_remount()修改安装标志并终止。
3. 否则，调用get_fs_type()来获得指向某一file_system_type对象的指针。
4. 如果要安装的文件系统指向诸如*/dev/hda1*一类的硬件设备，则要检查该设备是否存在、是否是可操作的。其执行的操作如下：
   a. 调用namei()来获得相应设备文件的目录项对象（参看本章后面的“路径名的查找”一节）。
   b. 检查与该设备文件相关的索引节点是否指向一个有效的块设备（参看第十三章中的“设备文件”一节）。
   c. 初始化指向该设备文件的哑元文件对象，然后用该文件操作的open方法打开这个设备文件。如果这个操作成功，则说明该设备是可操作的。
5. 如果要安装的文件系统并不指向一个硬件设备，则通过调用get_unnamed_dev()来获得一个主号为0的虚拟块设备。
6. 调用do_mount()，给它传递的参数为dev（设备号）、dev_name（设备文件名）、dir_name（安装点）、type（文件系统类型）、flags（安装标志）以及data（指向可选数据区的指针）。这个函数通过执行下列操作来安装所需的文件系统：
   a. 调用namei()来确定dir_name对应的dir_d目录项对象的位置。如果这个目录项不存在，则创建它［参看图12-5(a)］。

b. 获得mount_sem信号量，用这个信号量是为了让安装和卸载操作串行化。

c. 检查并确定dir_d->d_inode是一个目录的索引节点，并确定这个目录不是已安装的文件系统的根（dir_d->d_covers必须等于dir_d）。

d. 调用read_super()以获得新文件系统的超级块对象sb。（如果这个对象不存在，就创建它，并用从dev设备读取的信息填充它）。该超级块对象的s_root域指向要安装文件系统根目录的目录项对象［参看图12-5(b)］。

e. 前一个操作可能已挂起了当前进程，因此，要检查没有其他进程正在使用这个超级块，也没有进程已经成功地安装了这一文件系统。

f. 调用add_vfsmnt()来把一个新元素插入到已安装文件系统的链表中。

g. 把dir_d的d_mounts域置为超级块的s_root域，也就是说，设置为已安装文件系统的根目录。

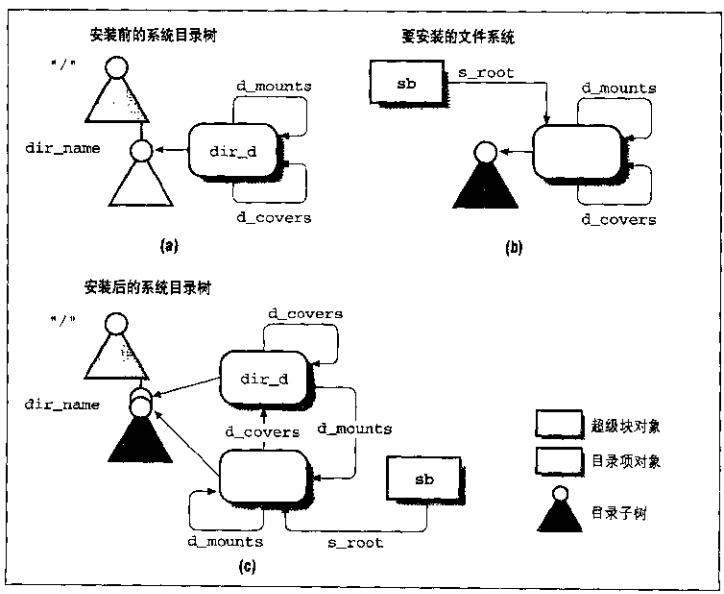


图12-5 安装一个文件系统

h. 把已安装文件系统根目录目录项对象的d_covers域置为dir_d[参看图12-5 (c)]。

i. 释放mount_sem信号量。

现在，通过d_mounts域把安装点的dir_d目录项链接到已安装文件系统的根目录的目录项对象；又通过d_covers域把这个根目录项对象链接到dir_d目录项对象。

卸载一个文件系统

umount()系统调用用来卸载一个文件系统。相应的服务例程sys_umount()作用于两个参数：文件名（或是安装目录或是块设备文件）和一组标志。该函数执行下列操作：

1. 检查进程是否具有卸载文件系统所需的能力。
2. 对这个文件名调用namei()来获得指向相关目录项对象的dentry指针。
3. 如果文件名指向安装点，则从dentry->d_inode->i_sb->s_dev得出设备的标识符。换言之，该函数从安装点的目录项开始到相关的索引节点，然后到相应的超级块，最后到设备标识符。
4. 否则，如果文件名指向设备文件，则从dentry->d_inode->i_rdev得出设备的标识符。
5. 调用dput(dentry)释放该目录项对象。
6. 刷新设备的缓冲区（参看第十四章中的“缓冲区高速缓存”一节）。
7. 获得mount_sem信号量。
8. 调用do_umount()执行下列操作：
   a. 调用get_super()获得已安装文件系统的超级块的指针sb。
   b. 调用shrink_dcache_sb()来删除指向dev设备的目录项对象而不扰乱其他的目录项。这个已安装的文件系统根目录的目录项对象将不被删除，因为它一直还由进行卸载的进程使用着。
   c. 调用fsync_dev()来强迫与dev设备相关的所有“脏”缓冲区都写到磁盘。

d. 如果dev是根设备（dev == ROOT_DEV），就不能卸载它。如果它还没有被重新安装，就以MS_RDONLY标志的设置来重新安装它，并返回。

e. 检查要卸载的文件系统的根目录所对应的目录项对象的引用计数器是否大于1。如果是，说明某个进程正在访问该文件系统的一个文件，因此，返回一个错误码。

f. 减少sb->s_root->d_covers（安装点目录的目录项对象）的引用计数器。

g. 把sb->s_root->d_covers->d_mounts置为sb->s_root->d_covers。这就删除了从安装点的索引节点到文件系统根目录的索引节点的链接。

h. 释放sb->s_root（前一个已安装文件系统的根目录）所指向的目录项对象，并把sb->s_root置为NULL。

i. 如果这个超级块已被修改，并且定义了超级块的write_super方法，则执行该方法。

j. 如果定义了超级块的put_super()方法，就调用它。

k. 把sb->s_dev置为0。

l. 调用remove_vfsmnt()从已安装文件系统的链表中删除合适的元素。

9. 对于与dev相关的所有依然“脏”的缓冲区（据推测，包含超级块信息），就调用fsync_dev()强迫向磁盘进行一次写操作，然后，调用这个设备文件操作中的release()方法。

10. 释放mount_sem信号量。

路径名的查找

在这一部分我们要说明VFS如何从文件路径名导出相应的索引节点。当进程必须标识一个文件时，它把它的文件路径名传递给某个VFS系统调用［如open()、mkdir()、rename()、stat()等等］。执行这一任务的标准过程就是分析路径名并把它拆分成一组文件名。除了最后一个文件名，所有的文件名都必定是目录。

如果路径名的第一个字符是“/”，那么这个路径名是绝对路径，因此从current->fs->root（进程的根目录）所标识的目录开始搜索。否则，路径名是相对路径，因此从current->fs->pwd（进程的当前目录）所标识的目录开始搜索。

在对最初目录的索引节点进行处理的过程中，代码要检查与第一个名字匹配的目录项以获得相应的索引节点。然后，从磁盘读出包含那个索引节点的目录文件，并检查与第二个名字匹配的目录项以获得相应的索引节点。对于包含在路径中的每个名字，反复执行这个过程。

目录项高速缓存极大地加速了这一过程，因为它把最近最常使用的目录项对象保留在内存。正如我们以前看到的，这样的每个对象使特定目录中的一个文件与它相应的索引节点相联系。因此，在很多情况下，路径名的分析可以避免立即从磁盘读取目录。

但是，事情并不像看起来那么简单，因为必须考虑如下的Unix和VFS文件系统的特点：

- 每个目录的访问权限必须进行检查，以验证是否允许进程读取这一目录的内容。
- 文件名可能是与任意一个路径名对应的符号链。在这种情况下，分析必须扩展到那个路径名的所有分量。
- 文件名可能是一个已安装文件系统的安装点。这种情况必须被察觉，这样查找操作必须延伸到这个新的文件系统。

namei()和lnamei()函数从一个路径名获得一个索引节点。二者之间的差别在于：如果符号链出现在路径名的最后一个分量中（尾部没有"/"），则namei()就要对它追踪下去，而lnamei()不进行这种追踪。这两个函数都通过调用lookup_dentry()而把繁重的任务委托给该函数。这个函数作用于三个参数：name指向文件名，base指向目录的目录项对象，从这个目录开始进行查找，而lookup_flags是一个位数组，它包含下列标志：

LOOKUP_FOLLOW

如果路径名的最后一个分量是一个符号链，则解释（追踪）它。当namei()调用lookup_dentry()时设置这个标志，当lnamei()调用它时清除这个标志。

LOOKUP_DIRECTORY

路径名的最后一个分量必须是一个目录。

LOOKUP_SLASHOK

即使最后一个文件名不存在，也允许路径名的末尾是"/"。

LOOKUP_CONTINUE

在路径名中还有文件名要检查。这个标志由lookup_dentry()在内部使用。

lookup_dentry()函数是递归的，因为它可能最终调用自己。因此，name可能表示一个完整路径名的还没有被分解的尾部部分。在这种情况下，base指向最后一个已分解路径名分量的目录项对象。lookup_dentry()执行下列操作：

1. 检查name的第一个字符和base的值，以确定搜索应该开始的目录。有三种情况可能发生。
   - name的第一个字符是"/"：说明路径名是一个绝对路径名，因此，把base置为current->fs->root。
   - name的第一个字符不同于"/"且base为NULL：则说明路径名是一个相对路径名，并把base置为current->fs->pwd。
   - name的第一个字符不同于"/"且base不为NULL：则说明路径名是一个相对路径名，且base保持不变。[只有在lookup_dentry()被递归调用时才发生这种情况]。
2. 从base->d_inode获得初始目录的索引节点。
3. 清除lookup_flags中的LOOKUP_FOLLOW标志。
4. 对包含在路径中的每个文件名，重复执行下列过程。如果遇到一个错误条件，就从循环退出并返回一个空的目录项指针，否则返回与文件路径名对应的目录项指针。在每次循环的开始，name指向要检查的下一个文件名，而base指向当前目录的目录项对象。
   a. 检查是否允许进程访问base目录（如果定义了索引节点的permission方法，就使用它）。
   b. 在目录项高速缓存中搜索相应目录项的过程中，从要使用的name的第一个分量名计算出一个散列值。此外，如果base目录是一个文件系统，它具有自己的d_hash()目录项散列方法，就调用base->d_op->d_hash()计算基于这个目录的散列值、分量名以及前一个散列值。
   c. 更新name，以使它指向下一个分量名（如果存在）的第一个字符，跳过任何"/"分割符。

d. 把flag局部变量置为以前在lookup_flags中设置的值。另外，如果当前已分解的分量尾部跟有“/”，就设置LOOKUP_DIRECTORY标志（要求检查这个分量是不是一个目录）和LCOKUP_FOLLOW标志（即使这个分量是一个符号链，也解释它）。此外，如果当前的分量被分解之后还有非空的分量，就设置LOOKUP_FOLLOW标志。

e. 调用reserved_lookup()执行下列操作：

1. 如果第一个分量名是单个句号（.），就把dentry局部变量置为base。
2. 如果第一个分量名是双个句号（..）且base等于current->fs->root，则把dentry局部变量置为base（因为进程已经在它的根目录中）。
3. 如果第一个分量名是双个句号（..）且base不等于current->fs->root，则把dentry局部变量置为base->d_covers->d_parent。通常，d_covers指向base本身，而且dentry被置为包含base的那个目录。不过，如果base不是某个已安装文件系统的根，则d_covers域指向安装点的索引节点，且dentry被置为包含安装点的那个目录。

f. 如果第一个分量名既不是（.）也不是（..），就调用cached_lookup()，并给它传递参数base和以前获得的散列值。如果目录项散列表包含需要的对象，就返回存放在dentry中目录项的地址。

g. 如果需要的目录项对象不在索引节点高速缓存中，就调用real_lookup()从磁盘读取这个目录，并创建一个新的目录项对象。这个函数作用于base和name两个参数，执行下列步骤：

1. 获取这个目录的索引节点的信号量i_sem。
2. 再次检查目录项对象是否在高速缓存中，因为当进程正在等待这个目录的信号量时，需要的目录项对象可能已被插入到高速缓存中。
3. 我们假定前一个尝试失败。调用d_alloc()分配一个新的目录项对象。
4. 调用与base目录相关的索引节点的lookup方法，以找到包含name的目录项且填充这个新的目录项对象。这个方法是与文件系统相关的。我们会在第十七章中进行描述。
5. 释放这个目录的索引节点的i_sem信号量。

6. 返回存放在dentry中的新对象的地址，或者，如果这个目录项没找到就返回一个错误码。

h. 调用follow_mount()检查dentry的d_mounts域是否与dentry具有相同的值，如果不是，dentry就是某个文件系统的安装点。在这种情况下，就用在dentry->d_mounts中的地址对应的目录项对象代替原来的目录项对象。

i. 调用do_follow_link()检查name是否是一个符号链。这个函数接收的参数有base、dentry及flags，并执行下列步骤：

1. 如果LOOKUP_FOLLOW标志没有被设置，则立即返回。因为这个标志是由lnamei()设置的，这就确保如果一个符号链出现在路径名的最后一个分量中（尾部没有"/"），lnamei()不解释这个符号链。

2. 检查dentry->d_inode是否包含follow_link方法。如果没有，这个索引节点就不是一个符号链，因此，该函数返回dentry输入参数。

3. 调用索引节点方法follow_link。这个依赖于文件系统的函数从磁盘读取与这个符号链相关的路径名，并且递归地调用lookup_dentry()来分解它。然后，该函数返回指向符号链所涉及的索引节点的指针（我们将在第十七章中描述Ext2怎样处理符号链）

因为lookup_dentry()调用do_follow_link()，do_follow_link()依次调用索引节点方法follow_link，而follow_link又调用lookup_dentry()，这样就产生了函数的递归调用。current的link_count域用来避免由于符号链内的循环引用而产生的死循环递归调用。这个域在每次递归执行follow_link()之前加1，而在执行之后又减1。如果这个值变为5，do_follow_link()以返回错误码而终止。因此，当路径名中的符号链个数没有限制时，嵌套的层数最多为5。

j. 如果一切进展顺利，base现在就指向与当前已分解的分量相关的目录项对象，因此把inode置为base->d_inode。

k. 如果flag的LOOKUP_DIRECTORY标志没有设置，则当前已分解的分量就是文件路径名中的最后一个路径名，返回存放在base中的这个地址。注意，base可能指向一个"负"状态的索引节点对象，也就是说，可能没有相关的索引节点。这对查找操作是件好事，因为不必再去解释最后一个分量。

1. 否则，如果LOOKUP_DIRECTORY被设置，当前分解的分量后面有一个“/”。有两种情况还需考虑：

   - inode指向一个有效的索引节点对象。在这种情况下，检查它是不是一个目录，这是通过查看这个索引节点操作的lookup方法是否已定义来确定的。如果不是，就返回一个错误码。然后，如果在flag中的标志LOOKUP_CONTINUE被设置（意味着当前分解的分量不是最后一个分量），就开始一次新的循环，或者返回存放在base中的地址（意味着这个分量是最后一个分量，即使在末尾跟着“/”）。
   - inode为空（意味着base指向一个“负”状态的索引节点对象）。只有在LOOKUP_CONTINUE被清0而LOOKUP_SLASHOK被设置时才返回base。否则，返回一个错误码。因为“负”状态的目录项对象表示一个被删除的文件，因此它不必出现在所查找路径名的中间（这发生在LOOKUP_CONTINUE被设置时）。此外，当路径名尾部的“/”存在时，“负”状态的索引节点也不必作为路径名中的最后一个分量出现，除非通过设置LOOKUP_SLASHOK来显式地允许它这么做。

VFS系统调用的实现

为了简短起见，我们不打算对表12-1中列出的所有系统调用的实现进行讨论。不过，概略叙述几个系统调用的实现还是有用的，这里仅仅说明VFS的数据结构怎样互相作用。

让我们重新考虑一下在本章开始所提到的例子：用户发出了一条shell命令：把*/floppy/TEST*中的MS-DOS文件拷贝到*/tmp/test*中的Ext2文件中。命令shell调用一个外部程序（如*cp*），我们假定*cp*执行下列代码片段：

```
inf = open("/floppy/TEST", O_RDONLY, 0);
outf = open("/tmp/test", O_WRONLY | O_CREAT | O_TRUNC, 0600);
do {
    l = read(inf, buf, 4096);
    write(outf, buf, l);
} while (l);
close(outf);
close(inf);
```

实际上，真正的*cp*程序的代码要更复杂些，因为它还必须检查由每个系统调用返回的错误代码。在我们的例子中，我们只把注意力集中在拷贝操作的“正常”行为上。

open()系统调用

open()的服务例程为sys_open()函数，该函数接收的参数为：要打开文件的路径名filename、访问模式的一些标志flags以及如果该文件被创建所需要的许可权位掩码mode。如果该系统调用成功，就返回一个文件描述符，也就是指向文件对象的指针数组current->files->fd的索引；否则返回-1。

在我们的例子中，open()被调用两次：第一次是为读（O_RDONLY标志）而打开*/floppy/TEST*，第二次是为写（O_WRONLY标志）而打开*/tmp/test*。如果*/tmp/test*不存在，该文件被创建（O_CREAT标志），文件主对该文件具有独占的读写访问权限（在第三个参数中的八进制数0600）。

相反，如果该文件已经存在，则它被从头开始重写（O_TRUNC标志）。表12-10列出了open()系统调用的所有标志。

表12-10 open()系统调用的标志

标志名	描述
FASYNC	通过信号发出异步I/O的通告
O_APPEND	总是在文件末尾写
O_CREAT	如果文件不存在，就创建它
O_DIRECTORY	如果文件不是一个目录，则失败
O_EXCL	对于O_CREAT标志来说，如果文件已经存在，则失败
O_LARGEFILE	大型文件（长度大于2 GB）
O_NDELAY	与O_NONBLOCK相同
O_NOCTTY	从不把文件看作控制终端
O_NOFOLLOW	不解释路径名中尾部的符号链
O_NONBLOCK	没有系统调用在这个文件上阻塞
O_RDONLY	为读而打开
O_RDWR	为读和写而打开

表 12-10 open()系统调用的标志（续）

标志名	描述
O_SYNC	同步写（阻塞，直到物理写终止）
O_TRUNC	截断文件
O_WRONLY	为写而打开

我们来描述一下 sys_open() 函数的操作。它执行如下操作：

1. 调用 getname() 从进程的地址空间读取该文件的路径名。
2. 调用 get_unused_fd() 在 current->files->fd 中查找一个空的插槽。相应的索引（新文件描述符）存放在 fd 局部变量中。
3. 调用 filp_open() 函数，传递给它的参数为路径名、访问模式标志以及许可权位掩码。这个函数依次执行下列步骤：
   a. 调用 get_empty_filp() 获得一个新的文件对象。
   b. 根据参数 flags 和 modes 的值设置该文件对象的 f_flags 和 f_mode 域。
   c. 调用 open_namei() 执行下列操作：
      1. 调用 lookup_dentry() 解释该文件路径名，并获得与这个文件相关的目录项对象。
      2. 执行一系列检查，以验证是否允许进程以 flags 参数所指定的值来打开该文件。如果是，返回该目录项对象的地址；否则，返回一个错误码。
   d. 如果是为了写入而访问，就检查该索引节点对象的 i_writecount 域的值。负值意味着该文件已经被映射到内存，说明写访问必须被拒绝（参看第十五章中的“内存映射”一节）。在这种情况下，返回一个错误码。其他的值都表示对该文件进行写入的进程个数。在后一种情况下，增加计数器的值。
   e. 初始化该文件对象的域。特别是把 f_op 域置为该索引节点对象的 i_op->default_file_ops 域的内容。这就为以后的文件操作建立起所有正确的函数。
   f. 如果定义了文件操作的 open 方法（缺省），就调用它。
   g. 清除 f_flags 中的 O_CREAT、O_EXCL、O_NOCTTY 和 O_TRUNC 标志。

h. 返回该文件对象的地址

4. 把current->files->fd[fd]置为该文件对象的地址。

5. 返回fd。

read()和 write()系统调用

让我们再回到*cp*例子的代码。open()系统调用返回两个文件描述符，分别存放在inf和outf变量中。然后，程序开始循环。在每次循环中，*/floppy/TEST*文件的一部分被拷贝到一个局部缓冲区[read()系统调用]，然后，在这个局部缓冲区中的数据又被拷贝到*/tmp/test*文件[write()系统调用]。

read()和write()系统调用非常相似。它们都需要三个参数：一个文件描述符fd、一个内存区的地址buf（该缓冲区包含要传送的数据），以及一个数count（指定应该传送多少字节）。当然，read()把数据从文件传送到缓冲区，而write()执行相反的操作。两个系统调用都返回所成功传送的字节数，或者发一个错误条件的信号并返回-1（注5）。

读或写操作总是发生在由当前文件指针所指定的文件偏移量处（文件对象的f_pos域）。两个系统调用都通过把所传送的字节数加到文件指针而更新文件指针。

简而言之，sys_read()[read()的服务例程]和sys_write()(write()服务例程）几乎执行相同的步骤：

1. 调用fget()从fd获取相应文件对象的地址file，并把引用计数器file->f_count减1。

2. 检查file->f_mode中的标志是否允许所请求的访问（读或写操作）。

注5：返回值小于count并不意味着发生了一个错误。因为总是允许内核在所请求的字节还没有被全部传送完时终止系统调用，因此用户程序必须检查返回值并在必要时重新发出系统调用。典型的情况是，当从管道或终端设备读取时，当在超过文件末尾的位置读取时，或者当系统调用被一个信号中断时，都会返回一个小于count的值。文件结束条件（EOF）可以通过read()返回的空值很容易地识别出。不要把这个条件与信号引起的异常终止相混淆，因为，如果读取数据之前read()被信号所中断，则说明发生了一个错误。

3. 调用 locks_verify_area()检查对要访问的文件部分是否有强制锁（参看本章后面的“文件加锁”一节）。

4. 如果执行一个写操作，则获得包含在索引节点中的 i_sem 信号量。这个信号量强迫一个进程对文件进行写入，而另一个进程对同一文件进行磁盘缓冲区的刷新（参看第十四章中的“把脏缓冲区写入磁盘”一节）。该信号量还强迫两个进程同时对同一文件进行写入。请注意，除非设置了 O_APPEND 标志，否则，POSIX 不需要对文件进行串行访问，如果一个程序员想互斥访问一个文件，那他就必须使用一个文件锁（参看下一节）。因此，实际上就有可能在一个进程从文件读取时而另一个进程对同一文件进行写入。

5. 调用 file->f_op->read 或 file->f_op->write 来传送数据。两个函数都返回实际传送的字节数。其产生的副作用是，文件指针被适当地更新。

6. 调用 fput()以减少引用计数器 file->f_count 的值

7. 返回实际传送的字节数。

close()系统调用

在我们例子的代码中，循环结束发生在 read()系统调用返回 0 时，也就是说，发生在*/floppy/TEST* 中的所有字节被拷贝到*/tmp/test* 中时。然后，程序关闭打开的文件，这是因为拷贝操作已经完成。

close()系统调用接收的参数为要关闭文件的文件描述符 fd。sys_close()服务例程执行下列操作：

1. 获得存放在 current->files->fd[fd]中的文件对象的地址，如果它为 NULL，就返回一个错误码。

2. 把 current->files->fd[fd]置为 NULL。释放文件描述符 fd，这是通过清除 current->files 中的 open_fds 和 close_on_exec 域的相应位来进行的（参见第十九章中有关对执行标志的关闭的内容）。

3. 调用 filp_close()，该函数执行下列操作：

   a. 调用文件操作的 flush 方法（如果已定义）

   b. 释放文件上的任何强制锁

c. 调用 fput()释放文件对象

4. 返回 flush 方法的错误码（通常为 0）。

文件加锁

当多个进程对一个文件能够进行访问时，就会出现同步问题，即，如果两个进程试图对同一文件位置进行写入会出现什么情况？或者，如果一个进程从文件的某个位置进行读取而另一个进程正在对同一位置进行写入会出现什么情况？

在传统的Unix系统中，对同一位置的并发访问会产生不可预料的结果。但是，系统提供一种允许进程对一个文件区进行加锁的机制，以使并发访问可以很容易地避免。POSIX 标准规定了基于 fcntl()系统调用的文件加锁机制。这样就有可能对文件的任意一部分（甚至一个字节）加锁或对整个文件（包含以后要追加的数据）加锁。因为进程可以选择仅仅对文件的一部分加锁，因此，它也可以在文件的不同部分保持多个锁。

这种锁不禁止不知道加锁的其他进程。与代码中的临界区类似，这种锁被认为是"劝告的"锁，因为，除非其他进程在访问文件之前合作检查一个锁的存在，否则这种锁不起作用。因此，POSIX 的锁被称为劝告锁（advisory lock）。

传统的 BSD 变体通过 flock()系统调用来实现劝告锁。这个调用不允许进程对文件的一个区域进行加锁，而仅仅是对整个文件进行加锁。

传统的 System V 变体提供了 lockf()系统调用，它仅仅是对 fcntl()提供的一个接口。更重要的是，System V 版本 3 引入了强制加锁（mandatory locking）。内核要检查open()、read()和write()系统调用的每次调用都不违背在要访问文件上的强制锁。因此，强制锁甚至在非合作的进程之间也被强迫执行（注 6）。

进程不管是使用劝告锁还是强迫锁，它们都能利用共享读锁（read lock）和独占写锁（write lock）。在文件的某个区域上，可以有任意多个进程进行读，但在同一时

注 6：很奇怪，即使其他进程拥有某一进程上的强制锁，该进程仍然会释放（或删除）一个文件。这种令人困惑的情况是可能发生的，因为当一个进程删除文件硬链接时，它不修改其内容而只是修改它的父目录的内容。

刻只能有一个进程进行写。此外，当其他进程对同一个文件都拥有自己的读锁时，就不可能获得一个写锁，反之亦然（参见表 12-11）。

表 12-11 是否允许加锁

	请求的锁	
当前锁	读	写
没有锁	是	是
读锁	是	否
写锁	否	否

Linux 文件加锁

Linux 支持文件加锁的所有方式为：劝告锁和强制锁，以及 fcntl()、flock()和 lockf()系统调用。不过，lockf()系统调用仅仅是库的一个封装函数，在此不打算进行讨论。

利用 mount()系统调用的 MS_MANDLOCK 标志，强制锁就可以对每个文件系统的主要成分进行开锁和加锁。缺省操作是打开强制锁，在这种情况下，flock()和 fcntl()都创建劝告锁。当这个标志被设置时，则 flock()还是产生劝告锁，而如果文件的组设置位为 1 且组的执行位为 0，则 fcntl()产生强制锁；否则产生劝告锁。

除了检查 read()和 write()系统调用外，如果一些系统调用可能修改文件的内容，内核对这些系统调用的处理还要考虑强制锁的存在。例如，如果文件的任意一个强制锁存在，则设置了 O_TRUNC 标志的 open()系统调用就会失败。

由 fcntl()产生的锁是 FL_POSIX 类型的锁，而由 flock()产生的锁是 FL_LOCK 类型的锁。这两种类型的锁可以安全地共存，但是谁也不影响谁。因此，通过 fcntl()加锁的文件看起来与通过 flock()加锁的文件不一样，反之亦然。

FL_POSIX 锁总是与一个进程和一个索引节点相关联。当进程死亡或一个文件描述符被关闭时（即使该进程对同一文件打开了两此或复制了一个文件描述符），这种锁会被自动地释放。

FL_LOCK锁总是与一个文件对象相关联。当一个锁被请求时，内核就把对同一文件对象加的所有其他锁都替换掉。这只发生在进程想把一个已经拥有的读锁改变为一个写锁或把一个写锁改变为一个读锁时。此外，当fput()函数正在释放一个文件对象时，对这个文件对象加的所有FL_LOCK锁都被撤消。不过，也有可能由其他进程对这同一文件（索引节点）设置了其他FL_LOCK读锁，它们依然是有效的。

文件锁的数据结构

file_lock数据结构表示文件锁，它的域如表12-12所示。所有的file_lock数据结构都包含在一个双向链表中。第一个元素的地址存放在file_lock_table中，而域fl_nextlink和fl_prevlink存放链表中相邻元素的地址。

表12-12 file_lock数据结构的域

类型	域	描述
struct file_lock *	fl_next	在索引节点链表中的下一个元素
struct file_lock *	fl_nextlink	在全局链表中的下一个元素
struct file_lock *	fl_prevlink	在全局链表中的前一个元素
struct file_lock *	fl_nextblock	在进程链表中的下一个元素
struct file_lock *	fl_prevblock	在进程链表中的前一个元素
struct files_struct *	fl_owner	文件主的files_struct
unsigned int	fl_pid	进程拥有者的PID
struct wait_queue *	fl_wait	阻塞进程的等待队列
struct file *	fl_file	指向文件对象的指针
unsigned char	fl_flags	锁标志
unsigned char	fl_type	锁类型
off_t	fl_start	被锁区域的起始偏移量
off_t	fl_end	被锁区域的末尾偏移量
void (*)(struct file_lock *)	fl_notify	当锁未被阻塞时的回收函数
union	u	具体文件系统的信息

磁盘上同一文件的所有file_lock结构都收集在一个简单的链表中，其第一个元素

由索引节点对象的i_flock域所指向file_lock结构的fl_next域指向链表中的下一个元素。

当进程试图获得一个劝告锁或强制锁时，它必须被挂起，直到分配给同一文件区域的前一个锁被释放为止。在某个锁上睡眠的所有进程被插入到一个等待队列中，而等待队列的地址存放在file_lock结构的fl_wait域中。此外，在任一文件锁上睡眠的所有进程都被插入到一个全局循环链表中，该链表是通过fl_nextblock和fl_prevblock实现的。

下面考察这两种锁之间的不同。

FL_LOCK锁

flock()系统调用作用于两个参数：要加锁文件的文件描述符fd和指定锁操作的参数cmd。如果cmd参数为LOCK_SH，则请求一个共享的读锁；为LOCK_EX，则请求一个排它的写锁；为LOCK_UN，则释放一个锁。如果LOCK_NB的值与LOCK_SH操作或LOCK_EX操作进行“或”，则这个系统调用不阻塞，换句话说，如果不能立即获得该锁，则该系统调用就返回一个错误码。注意，在文件内部不能指定一个区域，这种锁总是应用于整个文件。

当sys_flock()服务例程被调用时，则它执行下列步骤：

1. 检查fd是否是一个有效的文件描述符。如果不是，就返回一个错误码。否则，获得相应文件对象的地址。
2. 通过把fl_flags设置为FL_LOCK，调用flock_make_lock()初始化file_lock结构。把fl_type域置为F_RDLCK、F_WRLCK或F_UNLCK，这取决于cmd的值，并且把fl_file域置为该文件对象的地址。
3. 如果这种锁肯定已获得，就检查进程对打开的文件是否具有读和写权限，如果没有，就返回一个错误码。
4. 调用flock_lock_file()，传递给它的参数为文件对象指针filp、指向已初始化的数据结构file_lock的指针以及标志wait。如果该系统调用应该阻塞，则最后一个参数被设置，否则被清除。这个函数依次执行下列步骤：

a. 搜索filp->f_dentry->d_inode->i_flock指向的链表。如果在同一文件对象中还找到FL_LOCK锁，且请求一个F_UNLCK操作，则从索引节点链表和全局链表中删除这个file_lock元素，唤醒在该锁的等待队列上睡眠的所有进程，释放file_lock结构，并返回。

b. 否则，再次搜索索引节点链表以验证现有的FL_LOCK锁并不与所请求的锁产生冲突。在索引节点链表中，肯定没有FL_LOCK写锁，此外，如果处理的是正在请求的写锁，那根本就没有FL_LOCK锁。但是，进程可以要求把锁的类型改变为它已拥有的锁类型，这是通过第二次发出flock()系统调用完成的。如果找到一个冲突的锁且指定了LOCK_NB标志，则返回一个错误码，否则把当前进程插入到阻塞进程的循环双向链表中，并调用interruptible_sleep_on()挂起这个进程。

c. 否则，如果不存在不相容锁，就把file_lock结构插入到全局锁链表和该索引节点链表中，然后返回0（成功）。

FL_POSIX锁

当使用fcntl()系统调用锁定文件时，该系统调用作用于三个参数：要加锁文件的文件描述符fd、指向锁操作的参数cmd以及指向flock结构的指针fl。

FL_POSIX类型的锁能够保护任意一个文件区，甚至一个单独的字节。Flock结构的三个域指定要加锁的区域。l_start是区域的起始偏移量，并且是相对于文件的起始位置的（如果域l_whence被置为SEEK_SET）、相对于文件的当前指针（如果l_whence被置为SEEK_CUR）、或者相对于文件的末尾（l_whence被置为SEEK_END）。l_len域指定文件区的长度（或为0，意味着文件区扩展到文件末尾之外）。

sys_fcntl()服务例程执行的操作取决于在cmd参数中所设置的标志值：

F_GETLK

决定由flock结构描述的锁是否与另一个进程已获得的某个L_POSIX锁互相冲突。在冲突的情况下，用现有锁的有关信息重写flock结构。

F_SETLK

设置由flock结构描述的锁。如果不能获得该锁，则这个系统返回一个错误码。

F_SETLKW

设置由flock结构描述的锁。如果不能获得该锁，则这个系统调用阻塞，也就是说，调用进程进入睡眠状态。

当sys_fcntl()获得一个锁，则执行下列操作：

1. 从用户空间读取flock结构。
2. 获得fd对应的文件对象。
3. 检查这个锁是否是一个强制锁。在是的情况下，如果该文件有共享内存映射，则返回一个错误码（参见第十五章中的“内存映射”一节）。
4. 调用posix_make_lock()函数初始化一个新的file_lock结构。
5. 如果文件不允许由请求的锁类型指定的访问模式，则返回一个错误码。
6. 调用文件操作的lock方法（如果已定义）。
7. 调用posix_lock_file()函数，该函数执行下列操作：
   a. 对于索引节点的锁链表中的每个FL_POSIX锁，调用posix_locks_conflict()。该函数检查这个锁是否与所请求的锁互相冲突。从本质上说，在索引节点的链表中，同一区域中必定没有FL_POSIX锁，并且，如果进程正在请求一个写锁，同一个区域也可能根本没有FL_POSIX锁。但是，同一个进程所拥有的锁从不会冲突；这就允许进程把一个锁的特性改变为它已经拥有的锁。
   b. 如果找到一个冲突锁，并且以F_SETLK标志调用了fcntl()，则返回一个错误码，否则，当前进程应当被挂起。在这种情况下，调用posix_locks_deadlock()来检查在等待FL_POSIX锁的进程之间没有产生死锁条件，然后把当前进程插入到阻塞进程的循环链表中，并调用interruptible_sleep_on()挂起当前进程。
   c. 只要索引节点的锁链表中不包含冲突的锁，就检查当前进程的所有FL_POSIX锁，而当前进程想按需要对相邻的区域进行锁定、组合及拆分。例如，如果进程为某个文件区请求一个写锁，而这个文件区落在一个较宽的读锁区域内，那么，以前的读锁就会被拆分为两部分，这两部分覆盖非重叠的区域，而中间区域由新的写锁进行保护。在重叠区域，新锁总是代替旧锁。

d. 把新的 file_lock 结构插入到全局锁链表和索引节点锁链表中。

8. 返回值 0（成功）。

对 Linux 2.4 的展望

Linux 2.4 的 VFS 处理八种新的文件系统，其中有处理 DVD 的 udf。文件长度的最大值已经被极大地增加（至少从 VFS 的观点看），这是通过把索引节点的 I_size 域从 32 位扩充到 64 位实现的。

第十三章
管理I/O设备

上一章的虚拟文件系统依靠低层函数以适合每个设备的方式进行读、写或其他操作，对不同文件系统如何处理这些操作也进行了简单讨论。本章我们将看一下内核如何在实际的设备上调用这些操作。

在“I/O体系结构”一节，我们简单考察一下Intel 80x86的I/O体系结构。在“与I/O设备相关的文件”一节，我们说明VFS如何把“设备文件”与每个不同的硬件设备关联在一起，从而使应用程序可以以相同的方式使用所有的设备。本章着重介绍两种类型的设备：字符设备和块设备。

本章的目的是要说明Linux设备驱动程序的整体组织结构。有志于以自己的方法开发设备驱动程序的读者最好看一下由O'Reilly出版、Alessandro Rubini编写的《Linux Device Drivers》一书（译注1）。

译注1：本书中文版《Linux设备驱动程序》已由中国电力出版社出版，请访问*www.infopower.com.cn*。

I/O 体系结构

为了确保计算机能够正常工作，必须提供数据通道，让信息在连接到个人计算机的CPU、RAM和I/O设备之间流动。这些数据通道总称为总线，它是计算机中主要的通信通道。

目前使用的总线有很多种，例如ISA、EISA、PCI以及MCA。本节我们将讨论所有PC体系结构共有的功能性特点，而不具体介绍特定总线类型的技术细节。

实际上，一般可以把总线分为三种专用类型：

数据总线（data bus）

并行传送数据的一组线。Pentium的数据总线为64位宽。

地址总线（address bus）

并行传送地址的一组线。Pentium的地址总线为32位宽。

控制总线（control bus）

把控制信息传送到所连接的电路中的一组线。例如，Pentium使用控制线来指定总线是用来在处理器和RAM之间传送数据还是在处理器和I/O设备之间传送数据。控制线还可以决定必须执行读传送还是写传送。

当总线连接的是CPU和I/O设备时，就称它为I/O总线。在这种情况下，Intel 80x86微处理器只使用了32位地址总线中的16位对I/O设备进行寻址，使用了64位数据总线中的8位、16位或32位传送数据。实际上，每个I/O设备都依次连接到I/O总线上，这种连接使用了有3个元素的硬件构件层次：I/O端口、I/O接口和设备控制器。图13-1显示了I/O体系结构的这些构件。

I/O 端口

每个连接到I/O总线上的设备都有自己的I/O地址集，即所谓的I/O端口（I/O port）。在IBM PC体系结构中，I/O地址空间一共提供了65536个8位的I/O端口。可以把两个连续的8位端口看成一个16位端口，但是这必须是从偶数地址开始。同理，也可以把两个连续的16位端口看成一个32位端口，但是这必须是从4的整数倍地址开始。有四条专用的汇编语言指令可以允许CPU对I/O端口进行读写：它们分别是`in`、`ins`、`out`和`outs`。在执行其中的一条指令时，CPU使用地址总线选择所请求的I/O端口，使用数据总线在CPU寄存器和端口之间传送数据。

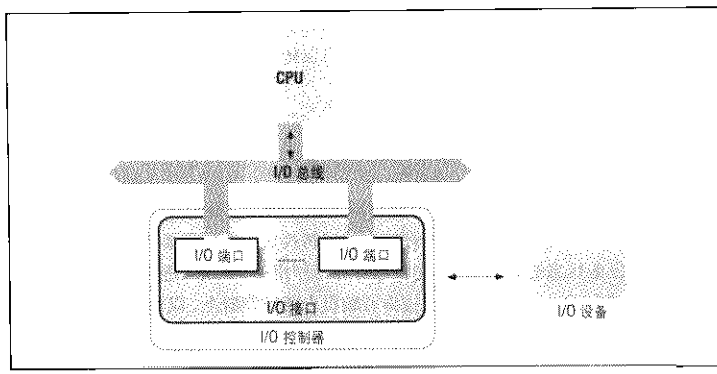


图13-1 PC的I/O体系结构

I/O端口还可以被映射到物理地址空间，因此，处理器和I/O设备之间的通信就可以直接使用对内存进行操作的汇编语言指令（例如，mov、and、or等等）。现代的硬件设备更倾向于映射I/O，因为这样处理的速度较快，并可以和DMA结合起来使用。

系统设计者的主要目的是提供对I/O编程的统一方法，但又不牺牲性能。为了达到这个目的，每个设备的I/O端口都被组织成如图13-2所示的一组专用寄存器。CPU把要发给设备的命令写入控制寄存器（control register），并从状态寄存器（status register）中读出表示设备内部状态的值。CPU还可以通过读取输入寄存器（input register）的内容从设备取得数据，也可以通过向输出寄存器（output register）中写入字节而把数据输出到设备。

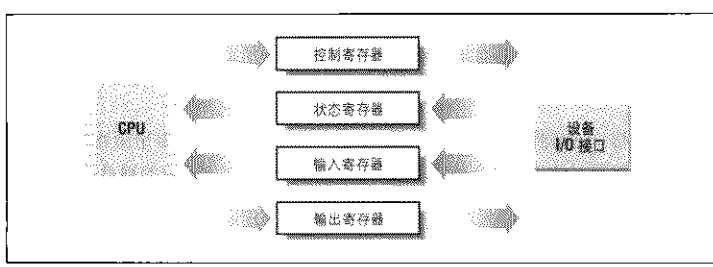


图13-2 专用I/O端口

为了降低成本，通常把同一I/O端口用于不同目的。例如，某些位描述设备的状态，而其他位指定发布给设备的命令。同理，也可以把同一I/O端口用作输入寄存器或输出寄存器。

I/O接口

I/O接口（I/O interface）是处于一组I/O端口和对应的设备控制器之间的一种硬件电路。它起翻译器的作用，即把I/O端口中的值转换成设备所需要的命令和数据。反之，它检测设备状态的变化，并对起状态寄存器作用的I/O端口进行相应地更新。还可以通过一条IRQ线把这种电路连接到可编程中断控制器上，以使它代表相应的设备发出中断请求。

有两种类型的接口：

专用I/O接口（custom I/O interface）

专门用于一个特定的硬件设备。在一些情况下，设备控制器与这种I/O接口处于同一块卡中（注1）。连接到专用I/O接口上的设备可以是内部设备（internal device）（位于PC机箱内部的设备），也可以是外部设备（external device）（位于PC机箱外部的设备）。

通用I/O接口（general-purpose I/O interface）

用来连接多个不同的硬件设备。连接到通用I/O接口上的设备通常都是外部设备。

专用I/O接口

专用I/O接口的种类很多，因此目前已装在PC上设备的种类也很多，我们无法一一列出，在此只列出一些最通用的接口：

键盘接口

键盘接口连接到一个键盘控制器上，这个控制器包含一个专用微处理器。这个微处理器对按下的组合键进行解码，产生一个中断并把相应的键盘扫描码写入输入寄存器。

注1：每块卡都要插入PC的一个可用空闲总线插槽中，如果一块卡通过一条外部电缆连接到一个外部设备上，那么在PC后面的面板中就有一个对应的连接器。

图形接口

图形接口和图形卡中对应的控制器封装在一起。图形卡有自己的显存（frame buffer），还有一个专用处理器，一些代码就存放在只读存储器芯片（ROM）中。显存是显卡上固化的内存，其中存放的是当前屏幕内容的图形描述。

磁盘接口

由一条电缆连接到磁盘控制器，通常磁盘控制器与磁盘放在一起。例如，IDE接口是由一条40线的带形导电电缆连接到智能磁盘控制器，在磁盘本身就可以找到这个控制器。

总线鼠标接口

鼠标中包含相应的控制器，一条电缆把鼠标和这个接口连接在一起。

网络接口

与网卡中的相应控制器封装在一起，用以接收或发送网络报文。虽然有很多广泛采用的网络标准，但以太网还是最为通用。

通用I/O接口

现代的PC都包含几个通用I/O接口，这些接口用于连接很多外部设备。最常用的接口有：

并口

传统上用于连接打印机，它还可以用来连接可移动磁盘、扫描仪、备份设备、其他计算机等等。数据的传送以1字节（8位）为单位并行进行。

串口

与并口类似，但在某一时刻，数据的传送是逐位进行的。串口包括一个通用异步收发器（UART）芯片，它可以把要发送的字节信息拆分成位序列，也可以把接收到的位流重新组装成字节信息。由于串口本质上速度低于并口，因此主要用于连接那些不需要高速操作的外部设备，如调制解调器、鼠标以及打印机。

通用串行总线（USB）

这是最近广泛流行的一种通用I/O接口。其操作速度较高，可适用于原来连接到并口和串口的外部设备。

PCMCIA 接口

大多数便携式计算机都包含这种接口。在不重新启动系统的情况下，形状类似于信用卡的外部设备可以被插入插槽或从插槽中被拔走。最常用的PCMCIA设备是硬盘、调制解调器、网卡和扩展RAM。

SCSI（小型计算机系统接口）接口

是把PC主总线连接到次总线（称为SCSI总线）的电路。SCSI-2总线允许一共8个PC和外部设备（硬盘、扫描仪、CR-ROM刻录机等等）连接在一起。如果有附加接口，广泛使用的SCSI-2和新的SCSI-3接口可以允许你连接多达16个以上的设备。SCSI标准是通过SCSI总线连接设备的通信协议。

设备控制器

复杂的设备可能需要一个设备控制器（device controller）来驱动。实际上，控制器的两个主要作用为：

- 对从I/O接口接收到的高级命令进行解释，并通过向设备发送适当的电信号序列强制设备执行特定的操作。
- 对从设备接收到的电信号进行转换和适当地解释，并修改（通过I/O接口）状态寄存器的值。

典型的设备控制器是磁盘控制器，它从微处理器（通过I/O接口）接收诸如“写这个数据块”之类的高级命令，并将其转换成诸如“把磁头定位在正确的磁道上”和“把数据写入这个磁道”之类的低级磁盘操作。现代的磁盘控制器相当复杂，因为它们可以把磁盘数据快速保存到内存的高速缓存中，还可以根据实际磁盘的几何结构重新安排CPU的高级请求，使其最优化。

比较简单的设备没有设备控制器，可编程中断控制器（参见第四章中的“中断和异常”一节）和可编程间隔定时器（参见第五章中的“可编程间隔定时器”一节）就是这样的设备。

直接内存访问（DMA）

所有的PC都包含一个称为DMAC（直接内存访问控制器，Direct Memory Access

Controller）的辅助处理器，它可以用来控制在RAM和I/O设备之间数据的传送。DMAC一旦被CPU激活，就可以自行传送数据；当数据传送完成之后，DMAC发出一个中断请求。当CPU和DMAC同时访问同一内存单元时，所产生的冲突由一个称为内存仲裁器（请参看第十一章中的“公共内存”一节）的硬件电路来解决。

使用DMAC最多的是磁盘驱动器和其他需要一次传送大量字节数的慢速设备。因为DMAC的设置时间相当长，所以在传送数量很少的数据时直接使用CPU效率更高。

原来的ISA总线所使用的第一个DMAC非常复杂，难于对其进行编程。PCI和SCSI总线所使用的最新DMAC依靠总线中的专用硬件电路，这就使设备驱动程序开发人员的开发工作变得简单。

到现在为止，我们已区分了三类内存地址：逻辑地址、线性地址以及物理地址，前两个在CPU内部使用，最后一个是CPU从物理上驱动数据总线所用的内存地址。但是，还有第四种内存地址，称为总线地址（bus address），它是除CPU之外的硬件设备驱动数据总线所用的内存地址。在PC体系结构中，总线地址和物理地址是一致的，但是在其他体系结构中，例如Sun的SPARC和康柏的Alpha体系结构中，这两种地址是不同的。

从根本上说，内核为什么应该关心总线地址呢？这是因为在DMA操作中数据传送不需要CPU的参与，I/O设备和DMAC直接驱动数据总线。因此，在内核开始DMA操作时，必须把所涉及的内存缓冲区总线地址或写入DMAC适当的I/O端口，或写入I/O设备适当的I/O端口。

与I/O设备相关的文件

正如在第一章中所提到的那样，Unix类操作系统都是基于文件的概念，文件是以字节序列而构成的信息载体器。根据这一点，可以把I/O设备当作文件来处理。因此，与磁盘上的正规文件进行交互所用的同一系统调用可直接用于I/O设备。例如，用同一`write()`系统调用既可以向正规文件中写入数据，也可以通过向*/dev/lp0*设备文件中写入数据从而把数据发往打印机。现在让我们详细地研究一下如何实现这种机制。

设备文件

设备文件（device file）是用来表示Linux所支持的大部分I/O设备的。除了文件名，每个设备文件都还有三个主要属性：

类型（type）

块设备或字符设备（稍后讨论二者之间的区别）。

主号（major number）

从1到255之间的一个数，用以标识设备的类型。通常，具有相同主号和相同类型的所有设备文件共享相同的文件操作集合，因为他们是由同一设备驱动程序来处理的。

次号（minor number）

在一组主号相同的设备之间唯一标识特定设备所使用的一个数字。

mknod()系统调用用来创建设备文件。其参数有设备文件名、设备类型、主号及次号。最后两个参数合并成一个16位的dev_t数：高8位标识主号，低8位标识次号。MAJOR和MINOR宏可以从这个16位数中抽取出这两个值，而MKDEV宏可以把主号和次号合并成一个16位的数字。实际上，dev_t是专用于应用程序的一个数据类型，内核使用kdev_t数据类型。在Linux 2.2中，这两个类型都会变为一个无符号短整型，但是在以后的Linux版本中，kdev_t会成为一个完整的设备文件描述符。

设备文件通常包含在*/dev*目录中。表13-1显示了一些设备文件（注2）的属性。注意同一主号既可以标识字符设备，也可以标识块设备。

表13-1 设备文件的例子

设备名	类型	主号	次号	说明
/dev/fd0	块设备	2	0	软盘
/dev/hda	块设备	3	0	第一个IDE磁盘
/dev/hda2	块设备	3	2	第一个IDE磁盘上的第二个主分区
/dev/hdb	块设备	3	64	第二个IDE磁盘

注2：分配给设备号的正式注册信息及*/dev*目录索引节点存放在*Documentation/devices.tex*文件中。也可以在*include/linux/major.h*文件中找到所支持的主设备号。

表13-1 设备文件的例子（续）

设备名	类型	主号	次号	说明
/dev/hdb3	块设备	3	67	第二个IDE磁盘上的第三个主分区
/dev/ttyp0	字符设备	3	0	终端
/dev/console	字符设备	5	1	控制台
/dev/lp1	字符设备	6	1	并口打印机
/dev/ttyS0	字符设备	4	64	第一个串口
/dev/rtc	字符设备	10	135	实时时钟
/dev/null	字符设备	1	3	空设备（黑洞）

一个设备文件通常与一个硬件设备（如硬盘，*/dev/hda*），或硬件设备的某一物理或逻辑分区（如磁盘分区，*/dev/hda2*）相关联。但在某些情况下，设备文件不会和任何实际的硬件关联，而是表示一个虚拟的逻辑设备。例如，*/dev/null*就是对应一个“黑洞”的设备文件，所有写入这个文件的数据都被简单地丢弃，因此，该文件看起来总为空。

就内核所关心的内容而言，设备文件名是无关紧要的。如果你建立了一个名为*/tmp/disk*的设备文件，类型为“块”，主号是3，次号为0，那么这个设备文件就和表中的*/dev/hda*等价。另一方面，对某些应用程序来说，设备文件名可能就很有意义。例如，通信程序可以假设第一个串口和*/dev/ttyS0*设备文件关联。但是，大部分应用程序通常都可以被配置为随意地与命名设备文件进行交互。

块设备和字符设备的比较

块设备（block device）具有以下特点：

- 可以在一次I/O操作中传送固定大小的数据块。
- 可以随机访问设备中所存放的块。传送数据块所需要的时间可以被认为独立于块在设备中的位置，也独立于当前设备的状态。

块设备典型的例子是硬盘、软盘及CD-ROM。也可以把RAM磁盘当作块设备来对待，这是通过把部分RAM配置成快速硬盘而获得的。因此，可以把这部分RAM作为应用程序高效存取数据的临时存储器。

字符设备（character device）具有以下特点：

- 可以在一次I/O操作中传送任意大小的数据。实际上，诸如打印机之类的字符设备可以一次传送一个字节，而诸如磁带之类的设备可以一次传送可变大小的数据块。
- 通常访问连续的字符。

网卡

有些I/O设备没有对应的设备文件。最明显的一个例子是网卡。实际上，网卡把向外发送的数据放入通往远程计算机系统的一条线上，把从远程系统中接收到的报文装入内核内存。虽然本书没有介绍网络的内容，但是有必要在内核与网卡编程接口上花点时间。

从BSD开始，所有的Unix系统为计算机中的每个网卡都分配一个不同的符号名。例如，第一个以太网卡名为`eth0`。然而，这个名字并没有对应的设备文件，也没有对应的索引节点。

由于没有使用文件系统，所以系统管理员必须建立设备名和网络地址之间的联系。因此，应用程序和网络接口之间的数据通信不是基于标准的有关文件的系统调用，而是基于`socket()`、`bind()`、`listen()`、`accept()`和`connect()`系统调用，这些系统调用对网络地址进行操作。这组系统调用是在Unix BSD中首先引入的，现在已经成为网络设备的标准编程模型。

VFS对设备文件的处理

虽然设备文件也在系统的目录树中，但是它们和正规文件以及目录文件有根本的不同。当进程访问正规文件时，它会通过文件系统访问磁盘分区中的一些数据块；而在进程访问设备文件时，它只要驱动硬件设备就可以了。例如，进程可以访问一个设备文件以从连接到计算机的温度计读取房间的温度。VFS的责任是为应用程序隐藏设备文件与正规文件之间的差异。

为了做到这点，VFS改变打开的设备文件的缺省文件操作。因此，可以把对设备文

件的任一系统调用转换成对设备相关的函数的调用，而不是对主文件系统相应函数的调用。设备相关的函数对硬件设备进行操作以完成进程所请求的操作（注3）。

控制I/O设备的一组设备相关的函数称为设备驱动程序（device driver）。由于每个设备都有一个唯一的I/O控制器，因此也就有唯一的命令和唯一的状态信息，所以大部分I/O设备类型都有自己的驱动程序。

设备文件类描述符

具有相同主号和类型的每类设备文件都是由device_struct数据结构来描述的，该结构包括两个域：设备类名（name）和指向文件操作表的一个指针（fops）。所有字符设备文件的device_struct描述符都包含在chrdevs表中。该表包含有255个元素，每个元素对应一个可能的主号。（所有设备文件的主号都不会是255，因为该值是为将来的扩展而保留的。）同理，块设备文件的255个描述符都包含在blkdevs表中。两个表的第一项都为空，因为没有一个设备文件的主号是0。

chrdevs和blkdevs这两个表最初都为空。register_chrdev()和register_blkdev()函数用来向其中的一个表中插入一个新项，而unregister_ chrdev()和unregister_blkdev()函数用来从表中删除一个项。

例如，可以按如下方式把并口打印机驱动程序类的描述符插入chrdevs表中：

```
register_chrdev(6, "lp", &lp_fops);
```

该函数的第一个参数表示主号，第二个参数表示设备类名，最后一个参数是指向文件操作表的一个指针。

如果设备驱动程序被静态地加入内核，那么，在系统初始化期间就注册相应的设备文件类。但是，如果设备驱动程序作为模块被动态装入内核（请参看附录二），那么，对应的设备文件类在装载模块时被注册，在卸载模块时被注销。

打开一个设备文件

我们在第十二章的“open()系统调用”一节中已经讨论了如何打开文件。让我们假

注3：注意，根据第十二章中的“路径名的查找”一节中介绍的命名解析机制，设备文件的符号连接与设备文件的作用相同。

设进程要打开一个设备文件。如果需要，VFS首先要对文件对象、目录项对象以及设备文件所对应的索引节点对象进行初始化。尤其是，如果索引节点对象不存在，那么VFS就调用合适的超级块对象的read_inode方法从磁盘上读取文件信息。在这样处理的过程中，这个方法把设备的主号和次号记录在索引节点对象的i_rdev域，并把设备文件的类型记录在i_mode域中（字符设备文件为S_IFCHR；块设备文件是S_IFBLK）。此外，还按如下方法设置一个指向适当的索引节点操作的指针：

```
if ((inode->i_mode & 0017000) == S_IFCHR)
    inode->i_op = &chrdev_inode_operations;
else if ((inode->i_mode & 0017000) == S_IFBLK)
inode->i_op = &blkdev_inode_operations;
```

chrdev_inode_operations和blkdev_inode_operations表中除default_file_ops域之外的所有域都是空的，这两个表的default_file_ops域分别指向def_chr_fops表和def_blk_fops表。def_chr_fops和def_blk_fops的表中除open之外的其他所有方法也都为空，两个open方法分别指向chrdev_open()函数和blkdev_open()函数。

filp_open()函数填充新的文件对象，特别是使用索引节点对象的i_op->default_file_ops域的内容初始化f_op域。于是，文件操作表就会变成def_chr_fops或者def_blk_fops。然后filp_open()调用open方法，从而执行chrdev_open()或者blkdev_open()。这两种open函数本质上执行3个操作：

1. 从索引节点对象的i_rdev域中取得设备驱动程序的主号：

```
major = MAJOR(inode->i_rdev);
```

2. 为设备文件安装合适的文件操作：

```
filp->f_op = chrdevs[major].fops;
```

（当然，这个例子是字符设备文件的情况；对于块设备文件，blkdev_open()函数使用blkdevs表。）

3. 如果文件操作表中定义了open方法，就调用它：

```
if (filp->f_op != NULL && filp->f_op->open != NULL)
    return filp->f_op->open(inode, filp);
```

注意，最后一次调用open()方法并不引起递归，因为此时这个域就包含了指向与

设备相关的函数的地址，而这个函数的工作是设置设备。通常，open()函数执行如下操作：

1. 如果设备驱动程序被包含在一个内核模块中，那么把引用计数器的值加1，以便只有把设备文件关闭之后才能卸载这个模块。（附录二介绍了用户如何装载和卸载模块。）
2. 如果设备驱动程序要处理多个同类型的设备，那么，就使用次号来选择合适的驱动程序，如果需要，还要使用专门的文件操作表选择驱动程序。
3. 检查该设备是否真正存在，现在是否正在工作。
4. 如果必要，向硬件设备发送一个初始化命令序列。
5. 初始化设备驱动程序的数据结构。

设备驱动程序

我们已经看到VFS使用了一组规范的通用函数（*open*，*read*，*lseek*等等）来控制设备。这些函数的实际实现由设备驱动程序全权负责。由于每个设备都有一个唯一的I/O控制器，因此就有唯一的命令和唯一的状态信息，所以大部分I/O设备都有自己的驱动程序。

我们与其描述数以百计现有的设备驱动程序，还不如集中描述内核是如何支持它们的。在介绍的过程中，我们会描述一些I/O体系结构的特征，这些特征是驱动程序编程人员必须考虑的。

内核支持的级别

内核以三种可能的方式支持对硬件设备的访问：

根本不支持

应用程序使用适当的in和out汇编语言指令直接与设备的I/O端口进行交互。

最小支持

内核不能识别硬件设备，但能识别I/O接口。用户程序把I/O接口视为能够读写字符流的顺序设备。

扩展支持

内核识别硬件设备，并处理I/O接口本身。事实上，这种设备可能就没有对应的设备文件。

第一种方法与内核设备驱动程序毫无关系，最普通的例子是X Window系统对图形显示的处理。这种方法效率很高，同时避免了X服务器使用I/O设备产生的硬件中断。为了让X服务器访问必须的I/O端口，这种方法还需要做一些其他努力。正如第三章的“任务状态段”一节中所介绍的那样，iopl()和ioperm()系统调用给进程授权访问I/O端口。只有具有root权限的用户才可以调用这两个系统调用。但是通过把可执行文件的fsuid域设置成0（超级用户的UID），普通用户也可以使用这些程序（请参看第十九章中的“进程的信任状和能力”一节）。

最小支持方法是用来处理连接到通用I/O接口上的外部硬件设备的。内核通过提供一个设备文件（由此而提供一个设备驱动程序）来了解I/O接口，应用程序通过读写设备文件来处理外部硬件设备。

最小支持优于扩展支持，因为它保持内核尽可能小。但是，在基于PC的通用I/O接口之中，只有串口的处理使用了这种方法。因此，诸如X服务器之类的应用程序可以直接控制串口鼠标，而串口调制解调器通常都需要一个诸如Minicom、Seyon或PPP（点对点协议）守护进程之类的通信程序。

最小支持的应用范围是有限的，因为当外部设备必须频繁地与内核内部数据结构进行交互时不能使用这种方法。例如，考虑一个连到通用I/O接口上的可移动硬盘。应用程序不能和所有的内核数据结构进程交互，也不能与识别磁盘、装载文件系统所需要的函数进行交互，因此，这种情况下就必须使用扩展支持。

一般情况下，直接连接到I/O总线上的任何硬件设备（如内置硬盘）都要根据扩展支持方法进行处理，内核必须为每个这样的设备都提供一个设备驱动程序。并口、USB、笔记本上的PCMCIA接口或者SCSI接口——简而言之，除串口之外的所有通用接口之上连接的外部设备都需要扩展支持。

值得注意的是，与标准文件相关的系统调用，如open()、read()和write()，不让应用程序完全控制底层硬件设备。事实上，VFS的最小公共命名方法不包含有些设备所需要的特殊命令的空间，或不让应用程序检查设备是否处于某一特殊的内部状态。

已引入的POSIX ioctl()系统调用可以满足这样的需要。除了设备文件的文件描述符和另一个表示请求的32位参数之外，这个系统调用还可以接收任意多个另外的参数。例如，特殊的ioctl()请求可以用来获得CD-ROM的音量或者弹出CD-ROM介质。应用程序可以用这类ioctl()请求来模拟一个CD播放器的用户接口。

监控I/O操作

I/O操作的持续时间通常是不可预知的。这可能和机械装置的情况有关(对于要传送的数据块来说是磁头的当前位置)，和实际的随机事件有关(数据报文什么时候到达网卡)，还和人为因素有关(用户在键盘上按下一个键或者发现打印机夹纸了)。在任何情况下，启动I/O操作的设备驱动程序都必须依靠一种监控技术在I/O操作终止或超时时发出信号。

在终止操作的情况下，设备驱动程序读取I/O接口状态寄存器的内容来确定这个I/O操作是否被成功执行。在超时的情况下，驱动程序知道一定出了问题，因为完成这个操作所允许的最大时间间隔已经用完，但什么也没做。

监控I/O操作结束的两种可用技术分别是轮询模式(polling mode)和中断模式(interrupt mode)。

轮询模式

CPU依照这种技术重复检查(轮询)设备的状态寄存器，直到寄存器的值表明I/O操作已经完成为止。我们已经在第十一章的“自旋锁”一节中提到一种基于轮询的技术：当处理器试图获得一个繁忙的自旋锁时，它就重复地查询变量的值，直到该值变成0为止。但是，应用到I/O操作中的轮询技术更加巧妙，这是由于有关的延时可能很大，而且驱动程序还必须记住检查可能出现的超时情况。为了避免浪费宝贵的机器周期，设备驱动程序在每次轮询操作之后会主动放弃CPU，以让其他可运行的进程可以继续执行：

```
for (;;) {
    if (read_status(device) & DEVICE_END_OPERATION)
        break;
    schedule();
    if (--count == 0)
        break;
}
```

在进入循环之前，count变量已被初始化，每次循环都对count的值减1，因此就可以使用这个变量实现一种粗略的超时机制。另外，更精确的超时机制可以通过这样的方法实现：在每次循环时读取节拍计数器jiffies的值（请参看第五章中的“PIT中断服务例程”一节），并将它与开始等待循环之前读取的原值进行比较。

中断模式

如果I/O控制器能够通过IRQ线发出I/O操作结束的信号，那么中断方式才被使用。设备驱动程序启动I/O操作，并调用interruptible_sleep_on()或sleep_on()函数（指向I/O设备等待队列的指针作为参数）。

当中断发生时，中断处理程序调用wake_up()来唤醒设备等待队列中所有睡眠的进程。因此，被唤醒的设备驱动程序可以检查I/O操作的结果。

超时控制的实现是通过静态或动态定时器完成的（参见第五章）。在开始执行I/O操作之前，给定时器设置合适的值，在操作终止时删除这个值。

访问I/O端口

in、out、ins和outs汇编语言指令都可以访问I/O端口。内核中包含了以下辅助函数来简化这种访问：

inb()、inw()、inl()

分别从I/O端口读取1、2或4个连续字节。后缀“b”、“w”、“l”分别代表一个字节（8位）、一个字（16位）以及一个长整型（32位）。

inb_p()、inw_p()、inl_p()

分别从I/O端口读取1、2或4个连续字节，然后执行一条“哑元（dummy，即空指令）”指令使CPU暂停。

outb()、outw()、outl()

分别向一个I/O端口写入1、2或4个连续字节。

outb_p()、outw_p()、outl_p()

分别向一个I/O端口写入1、2或4个连续字节，然后执行一条“哑元”指令使CPU暂停。

insb()、insw()、insl()

分别从I/O端口读入以1、2或4个字节为一组的连续字节序列。字节序列的长度由该函数的参数给出。

outsb()、outsw()、outsl()

分别向I/O端口写入以1、2或4个字节为一组的连续字节序列。

虽然访问I/O端口非常简单，但是检测哪些I/O端口已经分配给I/O设备可能就不这么简单了，对基于ISA总线的系统来说更是如此。通常，I/O设备驱动程序为了侦探硬件设备，需要盲目地向某一I/O端口写入数据，但是，如果其他硬件设备已经使用这个端口，那么系统就会崩溃。为了防止这种情况的发生，内核必须使用iotable表来记录分配给每个硬件设备的I/O端口。任何设备驱动程序都可以使用下面三个函数：

request_region()

把一个给定区间的I/O端口分配给一个I/O设备。

check_region()

检查一个给定区间的I/O端口是否空闲，或者其中一些是否已经分配给某个I/O设备。

release_region()

释放以前分配给一个I/O设备的给定区间的I/O端口。

当前分配给I/O设备的I/O地址可以从*/proc/ioports*文件中获得。

请求IRQ

我们在第四章的“IRQ线的动态处理”一节中已经看到：由于多个设备可以共享同一IRQ线，因此把IRQ分配给设备是在使用之前动态进行的。为了确保在需要时获得IRQ，在这个IRQ已被使用时不再对它请求，设备驱动程序通常采用下面的模式：

- 一个引用计数器记录目前正在对设备文件进行访问的进程个数。这个计数器的增加是在设备文件的open方法中进行的，而减少是在release方法中进行的（注4）。

注4：更准确地说，引用计数器记录的是引用设备文件的文件对象的个数，因为clone进程可能共享同一文件对象。

- open方法在增加这个计数器的值之前先要对其进行检查。如果这个计数器为0，那么设备驱动程序必须分配这个IRQ并启用这个硬件设备的中断。因此，驱动程序调用request_irq()，并适当地配置I/O控制器。
- release方法在减少这个计数器的值之后要对其进行检查。如果这个计数器为0，那么就说明现在没有其他进程在使用这个硬件设备。如果实际情况的确如此，那么该方法就调用free_irq()，因此也就释放了这条IRQ线，并禁止对这个I/O控制器的中断。

驱动DMA工作

在“直接内存访问（DMA）”一节中已经提到，很多I/O驱动程序都使用直接内存访问控制器（DMAC）来加快操作的速度。DMAC与设备的I/O控制器相互作用，共同实现数据传送。后文中我们还会看到，内核中包含一组易用的例程来对DMAC进行编程。当数据传送完成时，I/O控制器通过IRQ向CPU发出信号。

当设备驱动程序为某个I/O设备建立DMA操作时，必须使用总线地址指定所用的内存缓冲区。内核提供两个宏virt_to_bus和bus_to_virt，分别把线性地址转换成总线地址或把总线地址转换成线性地址。

与IRQ线一样，DMAC也是一种资源，必须把这种资源动态地分配给需要它的设备驱动程序。驱动程序开始和结束DMA操作的方法依赖于总线的类型。

ISA总线的DMA

每个ISA DMAC只能控制有限个通道。每个通道都包括一组独立的内部寄存器，所以，DMAC就可以同时控制几个数据的传送。

设备驱动程序通常使用下面的方式来申请和释放ISA DMAC。设备驱动程序照样要靠一个引用计数器来检测什么时候任何进程都不再访问设备文件。驱动程序执行以下操作：

- 在设备文件的open()方法中把设备的引用计数器加1。如果原来的值是0，那么，驱动程序执行以下操作：
  a. 调用request_irq()来分配ISA DMAC所使用的IRQ线。

b. 调用request_dma()来分配DMA通道。

c. 通知硬件设备应该使用DMA并产生中断。

d. 如果需要，为DMA缓冲区分配一个存储区域。

- 在必须启动DMA操作时，在设备文件的适当方法（通常是read和write）中执行以下操作：

a. 调用set_dma_mode()把通道设置成读/写模式。

b. 调用set_dma_addr()来设置DMA缓冲区的总线地址。（因为只有最低的24位地址会发给DMAC，所以缓冲区必须在RAM的前16MB中。）

c. 调用set_dma_count()来设置要发送的字节数。

d. 调用enable_dma()来启用DMA通道。

e. 把当前进程加入该设备的等待队列，并把它挂起。当DMAC完成数据传送操作时，设备的I/O控制器就发出一个中断，相应的中断处理程序会唤醒正在睡眠的进程。

f. 进程一旦被唤醒，就立即调用disable_dma()来禁用这个DMA通道。

g. 调用get_dma_residue()来检查是否所有的数据都已被传送。

- 在设备文件的release方法中，减少设备的引用计数器。如果该值变成0，就执行以下操作：

a. 禁用DMA和对这个硬件设备上的相应中断。

b. 调用free_dma()来释放DMA通道。

c. 调用free_irq()来释放DMA所使用的IRQ线。

PCI总线的DMA

PCI总线对于DMA的使用要简单得多，因为DMAC是集成到I/O接口内部的。在open方法中，设备驱动程序照样必须分配一条IRQ线来通知DMA操作的完成。但是，并没有必要分配一个DMA通道，因为每个硬件设备都直接控制PCI总线的电信号。要启动DMA操作，设备驱动程序在硬件设备的某个I/O端口中简单地写入DMA缓冲区的总线地址、传送方向以及数据大小；然后驱动程序就挂起当前进程。在最后一个进程关闭这个文件对象时，release方法负责释放这条IRQ线。

设备控制器的本地内存

很多硬件设备都有自己的内存，通常称之为I/O共享内存（I/O shared memory）。例如，所有比较新的图形卡都有几MB的RAM，称为显存（frame buffer），用它来存放要在屏幕上显示的屏幕影像。

地址映射

根据设备和总线类型的不同，PC体系结构中的I/O共享内存可以在三个不同的物理地址范围之间进行映射：

对于连接到ISA总线上的大多数设备

I/O共享内存通常被映射到从0xa0000到0xfffff的物理地址范围，这就在640KB和1MB之间留出了一段空间，就是我们在第二章的“保留的页框”一节中所介绍的那个“洞”。

对于使用VESA本地总线（VLB）的一些老设备

这是主要由图形卡使用的一条专用总线：I/O共享内存被映射到从0xe00000到0xffffff的地址范围中，也就是14MB到16MB之间。因为这些设备使页表的初始化更加复杂，因此已经不生产这种设备。

对于连接到PCI总线的设备

I/O共享内存被映射到很大的物理地址区间，位于RAM物理地址的顶端。这种设备的处理比较简单。

访问I/O共享内存

内核如何访问一个I/O共享内存单元？让我们从PC体系结构开始入手，这个问题很容易就可以解决，之后我们再进一步讨论其他体系结构。

不要忘了内核程序作用于线性地址，因此I/O共享内存单元必须表示成大于PAGE_OFFSET的地址。在后面的讨论中，我们假设PAGE_OFFSET等于0xc0000000，也就是说，内核线性地址是在第4G。

内核驱动程序必须把I/O共享内存单元的物理地址转换成内核空间的线性地址。在PC体系结构中，这可以简单地把32位的物理地址和0xc0000000常量进行或运算

得到。例如，假设内核需要把物理地址为0xc00b0fe4的I/O单元的值存放在t1中，把物理地址为0xfc000000的I/O单元的值存放在t2中，就可以使用下面的表达式来完成这项功能：

```
t1 = *((unsigned char *)(0xc00b0fe4));
t2 = *((unsigned char *)(0xfc000000));
```

在初始化阶段，内核已经把可用的RAM物理地址映射到线性地址空间第4G的开始部分。因此，分页单元把出现在第一个语句中的线性地址0xc00b0fe4映射回到原来的I/O物理地址0x000b0fe4，这正好落在从640KB到1MB的这段“ISA洞”中（请参看第二章的“Linux的分页”一节）。这工作起来很好。

但是，对于第二个语句来说，这里有一个问题，因为其I/O物理地址超过了系统RAM的最大物理地址。因此，线性地址0xfc000000就不需要与物理地址0xfc000000相对应。在这种情况下，为了在内核页表中包括对这个I/O物理地址进行映射的线性地址，必须对页表进行修改，这可以通过调用ioremap()函数来实现。ioremap()和vmalloc()函数类似，都调用get_vm_area()建立一个新的vm_struct描述符，其描述的线性地址区间为所请求I/O共享内存区的大小（请参看第六章中的“非连续内存区的描述符”一节）。然后，ioremap()函数适当地更新所有进程的对应页表项。

因此，第二个语句的正确形式应该为：

```
io_mem = ioremap(0xfb000000, 0x200000);
t2 = *((unsigned char *)(io_mem + 0x100000));
```

第一条语句建立一个2MB的线性地址区间，从0xfb000000开始；第二条语句读取地址0xfc000000的内存单元。驱动程序以后要取消这种映射，就必须使用iounmap()函数。

现在让我们考虑一下除PC之外的体系结构，在这种情况下，把I/O物理地址加上0xc0000000常量所得到的相应线性地址并不总是正确的。为了提高内核的可移植性，Linux特意包含了下面这些宏来访问I/O共享内存：

readb, readw, readl

分别从一个I/O共享内存单元读取1、2或者4个字节

writeb, writew, writel

分别向一个I/O共享内存单元写入1、2或者4个字节

memcpy_fromio, memcpy_toio

把一个数据块从一个I/O共享内存单元拷贝到动态内存中，另一个函数正好相反，把一个数据块从动态内存中拷贝到一个I/O共享内存单元

memset_io

用一个固定的值填充一个I/O共享内存区域

对于0xfc000000 I/O单元的访问推荐使用这样的方法：

```
io_mem = ioremap(0xfb000000, 0x200000);
t2 = readb(io_mem + 0x100000);
```

正是由于这些宏，就可以隐藏不同平台访问I/O共享内存所用方法的差异。

字符设备的处理

字符设备的处理相当容易，因为这既不需要对数据进行缓冲，也不涉及磁盘的高速缓存。当然，字符设备因各自的需求不同而有所不同：有些字符设备必须实现一种复杂的通信协议来驱动硬件设备，而有些字符设备只需从硬件设备的一对I/O端口中读取几个值。例如，多端口的串口卡设备（提供多个串口的一种硬件设备）的驱动程序要比总线鼠标的驱动程序复杂得多。

让我们概略叙述一下一种名为Logitech（罗技）总线鼠标的非常简单的字符设备驱动程序。这个驱动程序和*/dev/logibm*字符设备文件相关联，其主号为10，次号为0。

假设有一个进程打开*/dev/logibm*文件；正如我们在本章前面“VFS对设备文件的处理”一节中说明的一样，VFS最终调用主号为10的所有设备文件操作都通用的open方法。这类设备包括很多不同种类的设备，因此还要调用misc_open()方法，根据设备的次号装载一组更专用的文件操作。最终结果是，该文件对象的f_op域指向bus_mouse_fops表，并调用open_mouse()函数。该函数要执行以下这些操作：

1. 检查总线鼠标是否被连接。
2. 请求总线鼠标所使用的IRQ线，也就是IRQ5，并注册mouse_interrupt()中断服务例程。
3. 初始化一个类型为mouse_status的小mouse数据结构，该结构存放总线鼠标的有关状态信息。这些状态信息包括哪个键被按下，以及最后一次读取设备文件之后鼠标指针的水平、垂直位移。
4. 向0x23e控制寄存器中写入0，从而启用总线鼠标中断（Logitech总线鼠标使用从0x23c到0x23f的I/O端口）。

mouse数据结构的填充是异步进行的：每次用户移动鼠标或者按下一个鼠标键时，鼠标控制器都会产生一个中断，从而激活mouse_interrupt()函数。该函数执行以下操作：

1. 通过向0x23e控制寄存器写入适当的命令，并从0x23c输入寄存器中读取相应的值来查询鼠标设备的状态。
2. 更新mouse数据结构。
3. 向0x23e控制寄存器中写入0来重新启用鼠标中断（鼠标中断每次发生时，鼠标设备都会自动将其禁用）。

进程必须读取*/dev/logibm*文件的内容才能获得鼠标的状态。每个read()系统调用最终都会调用read_mouse()函数，也就是文件操作的read方法。该函数执行以下操作：

1. 检查进程是否最少请求了3个字节，否则就返回-EINVAL。
2. 检查对*/dev/logibm*的最后一次读操作之后鼠标状态是否已发生变化。如果没有，就返回-EAGAIN。
3. 调用disable_irq()来禁止IRQ5的中断处理，读取mouse数据结构中所存放的值。然后通过调用enable_irq()来重新启用IRQ5的中断处理。
4. 在最后一次读操作之后把代表鼠标状态（键的状态、水平位置和垂直位置）的3个字节写入用户态缓冲区。
5. 如果进程所请求的内容超过3个字节，就用0填充用户态缓冲区。

6. 返回所写字节的数量。

块设备的处理

诸如硬盘之类的典型块设备都有很高的平均访问时间。每个操作都需要几毫秒才能完成，主要是因为硬盘控制器必须在磁盘表面将磁头移动到记录数据的确切位置。但是，当磁头到达正确位置时，数据传送就可以稳定在每秒几十 MB 速率。

为了达到可以接受的性能，硬盘及类似的设备都是同时传送很多相邻的字节。在后面的讨论中，我们将说明当一次定位就可以全部访问存放在磁盘表面的一组数据时，这组数据就是相邻（adjacent）的。

Linux 块设备处理程序的组织相当复杂。我们不可能对内核中包含的所有用来支持这种处理程序的函数都进行详细的讨论。但是我们会提纲挈领地介绍一下一般的软件结构并引入主要的数据结构。内核对于块设备的支持具有以下特点：

- 通过 VFS 提供统一接口
- 对磁盘数据进行有效的预读
- 为数据提供磁盘高速缓存

内核基本上把 I/O 数据传送划分成两类：

缓冲区 I/O 操作

在这里，所传送的数据保存在缓冲区中，缓冲区是磁盘数据在内核中的普通内存容器。每个缓冲区都和一个特定的块相关联，而这个块由一个设备号和一个块号来标识。Linux 中把这些操作称为“同步的 I/O 操作”，实际上这是一个误导。术语“同步”在这里并不很恰当，因为缓冲区 I/O 操作实际上是异步的，换而言之，开始执行 I/O 操作的内核控制路径不用等待 I/O 操作完成就可以继续执行。这个术语可能是从旧版本的 Linux 中继承而来的。

页 I/O 操作

在这里，所传送的数据保存在页框中，每个页框包含的数据都属于正规文件。因为没有必要把这种数据存放在相邻的磁盘块中，所以就使用文件的索引节点和在文件内的偏移量来标识这种数据。Linux 又把这些操作错误地称为“异步 I/O 操作”。

缓冲区I/O操作经常被用在当进程直接读取块设备文件时，或者当内核读取文件系统中的特定类型的数据块（例如，包含有索引节点的块或超级块）时。在Linux 2.2中，也用缓冲区操作来写基于磁盘的正规文件。页I/O操作主要用于读取正规文件、文件内存映射和交换。这两种I/O数据传送依赖相同的驱动程序来访问块设备，但是，内核对这两种操作使用不同的算法和缓冲技术。

扇区、块和缓冲区

块设备的每次数据传送操作都作用于一组相邻字节，我们称之为扇区。在大部分磁盘设备中，扇区的大小是512字节，但是现在新出现的一些设备使用更大的扇区（1024和2014字节）。注意，应该把扇区作为数据传送的基本单元，不允许传送少于一个扇区的数据，而大部分磁盘设备都可以同时传送几个相邻的扇区。

内核在一个名为hardsect_size的表中存放每个硬件块设备的扇区大小。表中每个元素的索引就是对应块设备文件的主号和次号。因此，hardsect_size[3][2]就代表*/dev/hda2*（第一个IDE硬盘的第二个主分区，请参看表13-1）的扇区大小。如果hardsect_size[*M*]为NULL，共享主号*M*的所有块设备的扇区都为标准的512字节。

所谓块（block）就是块设备驱动程序在一次单独操作中所传送的一大块相邻字节。注意不要混淆块（block）和扇区（sector）：扇区是硬件设备传送数据的基本单元，而块只是硬件设备请求一次I/O操作所涉及的一组相邻字节。

在Linux中，块大小必须是2的幂，而且不能超过一个页框。此外，它必须是扇区大小的整数倍，因为每个块必须包含整数个扇区。因此，在PC体系结构中，允许块的大小为512、1024、2048和4096字节。同一个块设备驱动程序可以作用于多个块大小，因为它必须处理共享同一主号的一组设备文件，而每个块设备文件都有自己预定义的块大小。例如，一个块设备驱动程序可能会处理有两个分区的硬盘，一个分区包含Ext2文件系统，另一个分区包含交换分区（请参看第十六章和第十七章）。此时，块设备驱动程序利用两个不同的块大小：1024字节用于Ext2分区（注5），4096字节用于交换分区。

内核在一个名为blksize_size的表中存放块的大小，表中每个元素的索引就是相

注5：1024字节是标准的Ext2块大小，也允许设置其他块大小。

应块设备文件的主号和次号。如果blksize_size[*M*]为NULL，那么共享主号M的所有块设备都使用标准的块大小，即1024字节。

每个块都需要自己的缓冲区，它是内核用来存放块内容的RAM内存区。当设备驱动程序从磁盘读出一个块时，就用从硬件设备中所获得的值来填充相应的缓冲区；同样，当设备驱动程序向磁盘中写入一个块时，就用相关缓冲区的实际值来更新硬件设备上相应的一组相邻字节。缓冲区的大小一定要与块的大小相匹配。

缓冲区I/O操作概述

图13-3使用图例说明了通用块设备驱动程序的体系结构，以及在为缓冲区I/O操作服务时所涉及到的主要成分。

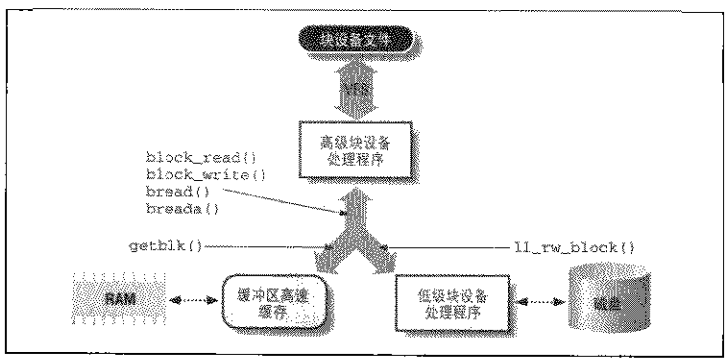


图13-3 块设备缓冲区I/O操作的处理程序体系结构

块设备驱动程序通常被划分为两部分：高级驱动程序和低级驱动程序。前者处理VFS层，后者处理硬件设备。

假设进程对一个设备文件发出read()或write()系统调用。VFS执行对应文件对象的read或write方法，由此就调用高级块设备处理程序中的一个过程。这个过程执行的所有操作都与对这个硬件设备的具体读写请求有关。内核提供两个名为block_read()和block_write()通用函数来留意所有事件的发生［请参看本章后面的“block_read()和block_write()函数”一节］。因此，在大部分情况下，高

级硬件设备驱动程序不必做什么，而设备文件的read和write方法分别指向block_read()和block_write()方法。

但是，有些块设备的处理程序需要自己专用的高级设备驱动程序。典型的例子是软驱的设备驱动程序，它必须检查从上次访问磁盘以来，用户一直没有改变驱动器中的磁盘。如果已插入一张新磁盘，那么设备驱动程序必须使缓冲区中所包含的旧数据无效。

即使高级设备驱动程序有自己的read和write方法，这两个方法通常最终还会调用block_read()和block_write()函数。这些函数把对I/O设备文件的访问请求转换成对相应硬件设备的块请求。正如我们在第十四章中的“缓冲区高速缓存”一节中所看到的那样，所请求的块可能已在主存，因此block_read()和block_write()函数调用getblk()函数来检查高速缓存中是否已经预取了块，还是从上次访问以来高速缓存一直都没有改变。如果块不在高速缓存中，getblk()就必须调用ll_rw_block()继续从磁盘中读取这个块［请参看“ll_rw_block()函数”一节］。后面这个函数激活操纵设备控制器的低级驱动程序，以执行对块设备所请求的操作。

当VFS直接访问某一块设备上的特定块时，也会触发缓冲区I/O操作。例如，如果内核必须从磁盘文件系统中读取一个索引节点，那么它必须从相应磁盘分区的块中传送数据。对于特定块的直接访问是由bread()和breada()函数［请参看后面的“bread()和breada()函数”一节］来执行的，这两个函数又会调用前面提到过的getblk()和ll_rw_block()函数。

由于块设备速度很慢，因此缓冲区I/O数据传送通常都是异步处理的：低级设备驱动程序对DMAC和磁盘控制器进行编程来控制其操作，然后结束。当数据传送完成时，就会产生一个中断，从而第二次激活这个低级设备驱动程序来清除这次I/O操作所涉及的数据结构。通过这种方法，所有的内核控制路径都不必挂起等待数据传送的完成（除非这个内核控制路径明确地要求等待某个数据块）。

预读的作用

对磁盘的访问大多数是顺序的。正如我们将在第十七章中所看到的，文件被存放在磁盘的一组大范围相邻扇区中，因此只要很少地移动磁头就可以快速读取文件的内

容。当程序读取或拷贝文件时，通常都是从第一个字节到最后一个字节顺序访问的。因此，只要几个I/O操作就可以读取磁盘上很多相邻扇区的内容。

所谓预读（read-ahead）就是在实际发出块请求以前提前读取块设备几个相邻块的一种技术。大部分情况下，预读都可以显著地提高磁盘的性能，因为大范围的相邻扇区只需较少的处理命令，从而减少了磁盘控制器的处理量。此外，系统的响应能力也得以提高。顺序读取块设备的进程可以更快速地得到所请求的数据，因为驱动程序执行的磁盘访问操作减少了。

然而，预读对于随机访问的块设备是没有什么用处的。在这种情况下，预读反而会带来负面的影响，因为它在磁盘的高速缓存中填充很多无用的信息，浪费了磁盘高速缓存的空间。因此，当内核判断出刚刚发出的I/O访问与前一个不是顺序的时，就停止预读。文件对象有一个f_reada标志域，对应文件（或块设备文件）启用预读技术时设置这个域，否则清0。

内核在一个名为read_ahead的表中存放在顺序读取设备文件时预读的字节数（确切地说是标准的512字节的扇区数）。值“0”表示预读的缺省数是8个标准的512字节扇区，即4KB主号相同的所有块设备文件具有相同的预读扇区数，因此表read_ahead中的每个元素是由主设备号来索引的。

block_read()和block_write()函数

只要进程对设备文件发出读写操作，高级设备驱动程序就调用block_read()和block_write()函数。例如，*superformat*程序通过把块写入*/dev/fd0*设备文件来格式化磁盘，相应文件对象的write方法就调用block_write()函数。

block_read()和block_write()函数接收以下参数：

filp

和这个设备文件关联的文件对象的地址。

buf

用户态地址空间中的内存区的地址。block_read()把从块设备中读出的数据写入这个内存区；反之，block_write()从这个内存区中读取要写入块设备的数据。

交换高速缓存是由页高速缓存数据结构实现的，实现过程在第十四章中的“页高速缓存”一节中已经介绍过。回想一下页高速缓存中包含的页是与正规文件相关的，并且散列表允许算法快速从索引节点对象的地址和文件内部偏移量中导出页描述符的地址。交换高速缓存中的页在页高速缓存中以其他页的形式存放，并对其进行以下特殊的处理：

- 页描述符的inode域存放的是包含在swapper_inode变量中的虚拟索引节点对象地址。
- offset 域存放的是与该页相关的被换出页的标识符。
- 在 flags 域中的 PG_swap_cache 标志被设置。

还有，当该页被放入交换高速缓存时，该页描述符的count域和页插槽引用计数器都被增加，因为交换高速缓存既要使用页框，也要使用页插槽。

内核使用几个函数来处理交换高速缓存，这些函数主要是基于第十四章中的“页高速缓存”一节中的讨论。稍后我们将说明这些相对低层的函数是如何被高层函数调用来根据需要换入换出页的。

处理交换高速缓存的函数有：

in_swap_cache()

检查页的PG_swap_cache标志来确定该页是否属于交换高速缓存；如果属于，就返回存放在offset域中的被换出页的标识符。

lookup_swap_cache()

对作为参数传递的被换出页标识符进行操作并返回该页的地址，如果该页不在这个高速缓存中就返回0。该函数调用find_page()函数，把swapper_inode的虚索引节点对象和被换出页标识符的地址作为参数传递来查找所需要的页。如果该页在交换高速缓存中，lookup_swap_cache()就检测该页是否被锁定；如果被锁定，就调用wait_on_page()把当前进程挂起，直到该页被取消锁定为止。

add_to_swap_cache()

把页插入交换高速缓存中。该页描述符的inode和offset域分别被设置成swapper_inode虚索引节点对象的地址和这个被换出页标识符的地址。然后

立即执行的，它不会等到所有的数据都已经发送之后才开始在缓冲区高速缓存中查找下一组块。不过，最终结果都是相同的。这个步骤完成之后，这些块的所有缓冲区中都包含了有效数据，用户进程所请求的数据都被拷贝到用户内存区。

7. 把拷贝到用户内存区的字节数加到 *ppos 上。

8. 设置 filp->f_reada 标志，这样就可以再次启用预读机制（除非进程修改了文件指针，在这种情况下要清除这个标志）。

9. 返回拷贝到用户内存区的字节数。

block_write()函数和 block_read()函数类似，因此，我们就不再详细描述。但是，二者之间有一些很大的差异需要注意：

- 在启动写操作之前，block_write()函数必须检查块硬件设备是否是只读的，如果是，就返回一个错误码。例如，在试图对 CD-ROM 盘相关的一个块设备文件进行写入操作时就出现这种情况。ro_bits表中的每一位对应一个块硬件设备：如果相应的设备不能被写，对应位被置位，如果可以被写，对应位被清除。

- block_write()函数必须对要写入的第一个块中的第一个字节的偏移量进行检查。如果偏移量不为零，而且缓冲区高速缓存中已不包含第一个块中的有效数据，那么这个函数就必须在重新写入这个块之前先从磁盘读出这个块。实际上，因为块设备驱动程序是对整个块进行操作的，所以写操作必须在第一个块要写入的字节之前保留部分位置。同理，除非要写入的最后一个字节处于最后一个块的最末尾的位置，否则在重写最后一块之前该函数必须从磁盘中读取要写入的最后一个块。

- block_write()函数不必调用 ll_rw_block()来强制对磁盘进行写操作。通常，它只是把被写块的缓冲区标记为“脏（dirty）”，因此就推迟了磁盘上相应扇区的实际更新（请参看第十四章中的“把脏缓冲区写入磁盘”一节）。但是，如果打开这个块设备文件的调用使用了O_SYNC标志，那么这个函数就要调用 ll_rw_block()函数。在这种情况下，调用进程希望一直等待（睡眠），直到数据被物理地写入硬件设备为止，这样磁盘就总能反映进程的真实情况。

bread()和breada()函数

bread()函数检查缓冲区高速缓存中是否已经包含了一个特定的块，如果还没有，该函数就从块设备中读取这个块。文件系统广泛使用bread()从磁盘位图、索引节点以及其他基于块的数据结构中读取数据。[回想一下当进程要读块设备文件时是使用block_read()函数，而不是使用bread()函数。] 该函数接收设备标志符、块号和块大小作为参数，执行以下操作：

1. 调用getblk()函数来查找缓冲区高速缓存中的一个块，如果这个块不在高速缓存，那么getblk()就为它分配一个新的缓冲区。
2. 如果这个缓冲区已经包含最新数据，getblk()结束。
3. 调用ll_rw_block()函数启动读操作。
4. 等待，直到数据传送完成为止。这是通过调用一个名为wait_on_buffer()的函数来实现的，该函数把当前进程插入b_wait等待队列中，并挂起当前进程直到这个缓冲区被开锁为止。

breada()和bread()十分类似，但是它除了读取所请求的块之外，还要另外预读一些其他块。注意不存在把块直接写入磁盘的函数。写操作永远都不会成为系统性能的瓶颈，因此写操作通常都会延时（请参看第十四章中的“把脏缓冲区写入磁盘”一节）。

缓冲区首部

缓冲区首部（buffer head）是一个与每个缓冲区相关的buffer_head类型的数据结构。它包含内核处理缓冲区所需要了解的所有信息。因此，在对每个缓冲区进行操作之前，内核都要首先检查其缓冲区首部。

缓冲区首部的域如表13-2所示。每个缓冲区首部的b_data域存放对应缓冲区的开始地址。由于一个页框可以存放多个缓冲区，因此b_this_page域就指向该页中的下一个缓冲区的首部。这个域简化了这些页框的存储和检索。b_blocknr域存放逻辑块号，即在磁盘分区内块的索引号。

表 13-2 缓冲区首部的域

类型	域	说明
unsigned long	b_blocknr	逻辑块号
unsigned long	b_size	块大小
kdev_t	b_dev	虚拟设备标志符
kdev_t	b_rdev	实际设备标志符
unsigned long	b_rsector	实际设备中原始扇区的个数
unsigned long	b_state	缓冲区状态标志
unsigned int	b_count	块引用计数器
char *	b_data	指向缓冲区的指针
unsigned long	b_flushtime	缓冲区刷新时间
struct wait_queue *	b_wait	缓冲区等待队列
struct buffer_head *	b_next	冲突散列表中的下一项
struct buffer_head **	b_pprev	冲突散列表中的上一项
struct buffer_head *	b_this_page	每页缓冲区链表
struct buffer_head *	b_next_free	链表中的下一个元素
struct buffer_head *	b_prev_free	链表中的上一个元素
unsigned int	b_list	包含缓冲区的 LRU 链表
struct buffer_head *	b_reqnext	请求的缓冲区链表
void (*)()	b_end_io	I/O 完成方法
void (*)	b_dev_id	专用设备驱动程序的数据

b_state 域中存放以下标志:

BH_Uptodate

如果缓冲区中包含了有效数据就置位。buffer_uptodate()函数会返回这个标志的值。

BH_Dirty

如果缓冲区中的数据是脏数据就置位，也就是说，如果缓冲区中包含有必须写入块设备的数据就置位。buffer_dirty()函数会返回这个标志的值。

BH_Lock

如果缓冲区加锁就置位，这发生在磁盘移动时所涉及的缓冲区。buffer_locked()函数会返回这个标志的值。

BH_Req

如果相应的块已经被请求（请参看下一节），就置位。buffer_req()函数会返回这个标志的值。

BH_Protected

如果缓冲区是受保护的（受保护的缓冲区永远不会被释放）就置位。buffer_protected()函数会返回这个标志的值。这个标志只用来实现缓冲区高速缓存之上的RAM盘。

b_dev域表示存放在缓冲区中的块所在的虚拟设备，而b_rdev域表示实际设备。二者之间的差别对于简单的硬盘来说毫无意义，但是对于并行操作的多个磁盘存储单元（即模型RAID，廉价冗余磁盘阵列）就很重要了。出于安全和效率的原因，存放在RAID阵列上的文件被分散到几个磁盘上，而应用程序把这些磁盘看作一个单独的逻辑盘。除了用b_blocknr域来表示逻辑块号外，还必须用b_rdev域表示特定的磁盘单元，b_rsector域表示相应的扇区号。

块设备请求

虽然块设备驱动程序可以一次传送一个单独的数据块，但是内核并不会为磁盘上每个被访问的数据块都单独执行一次I/O操作，因为这会导致磁盘性能的下降，确定磁盘表面块的物理位置是相当费时的。取而代之的是，只要可能，内核就试图把几个块合并在一起，并作为一个整体来处理，这样就减少了磁头的平均移动时间。

当进程、VFS层或者任何其他的内核部分要读写一个磁盘块时，就真正引起一个块设备请求（block device request）。从本质上说，这个请求描述的是所请求的块以及要对它执行的操作类型（读还是写）。然而，并不是请求一发出，内核就满足它，I/O操作仅仅被调度，稍后才会被执行。这种人为的延迟有悖于提高块设备性能的关键机制。当请求传送一个新的数据块时，内核检查能否通过稍微扩大前一个一直处于等待状态的请求而满足这个新请求，也就是说，能否不用进一步的搜索操作就能满足新请求。由于磁盘的访问大都是顺序的，因此这种简单机制就非常高效。

延迟请求复杂化了块设备的处理。例如，假设某个进程打开了一个正规文件，然后，文件系统的驱动程序就要从磁盘读取相应的索引节点。高级块设备驱动程序把这个请求加入一个等待队列，并把这个进程挂起，直到存放索引节点的块被传送为止。然而，高级块设备驱动程序不会被阻塞，因为试图访问同一磁盘的其他任何进程都会被阻塞。

为了防止挂起块设备驱动程序，正如我们在“缓冲区I/O操作概述”中介绍的一样，每个I/O操作都是异步处理的。因此，不会强制任何内核控制路径等到数据传送完成才执行。特别是块设备驱动程序是中断驱动的（请参看本章的“监控I/O操作”一节），因此，只要高级驱动程序一发出块请求，就可以终止它的执行。在稍后的时间低级驱动程序才被激活，它会调用一个所谓的策略程序（strategy routine）从一个队列中取得这个请求，并向磁盘控制器发出适当的命令来满足这个请求。当I/O操作完成时，磁盘控制器就产生一个中断，如果需要，相应的处理程序会再次调用这个策略程序来处理队列中进程的下一个请求。

每个块设备驱动程序都维护自己的请求队列（request queue）；每个物理块设备都应该有一个请求队列，以提高磁盘性能的方式对请求进行排序。因此策略程序就可以顺序扫描这种队列，并以最少地移动磁头而为所有的提供服务。

每个块设备请求都是用一个请求描述符（request descriptor）来表示的，请求描述符存放在如表13-3所示的request数据结构中。数据传送的方向存放在cmd域中：该值可能是READ（把数据从块设备读到RAM中）或者WRITE（把数据从RAM写到块设备中）。rq_status域用来定义请求的状态：对于大部分块设备来说，这个域的值可能为RQ_INACTIVE（请求描述符还没有使用）或者RQ_ACTIVE（有效的请求，低级设备驱动程序要对其服务或正在对其服务）。

表13-3 请求描述符的域

类型	域	说明
int	rq_status	请求状态
kdev_t	rq_dev	设备标识符
int	cmd	所请求的操作
int	errors	成功或错误代码
unsigned long	sector	第一个扇区号
unsigned long	nr_sector	请求的扇区个数

表13-3 请求描述符的域（续）

类型	域	说明
unsigned long	current_nr_sector	当前块的扇区个数
char *	buffer	I/O传送使用的内存区
struct semaphore *	sem	请求信号量
struct buffer_head *	bh	第一个缓冲区描述符
struct buffer_head *	bhtail	最后一个缓冲区描述符
struct request *	next	请求队列链表

一次请求可能包括同一设备中的很多相邻块。rq_dev域指定块设备，而sector域说明请求中第一个块对应的第一个扇区的编号。nr_sector和current_nr_sector给出要传送数据的扇区数。正如我们会在后面的“低级请求处理”一节中看到的一样，sector、nr_sector和current_nr_sector域都可以在请求得到服务的过程中而被动态修改。

请求块的所有缓冲区首部都被集中在一个简单链表中。每个缓冲区首部的b_reqnext域指向链表中的下一个元素，而请求描述符的bh和bhtail域分别指向链表的第一个元素和最后一个元素。

请求描述符的buffer域指向实际数据传送所使用的内存区。如果只请求一个单独的块，那么缓冲区只是缓冲区首部的b_data域的一个拷贝。然而，如果请求了多个块，而这些块的缓冲区在内存中又不是连续的，那么就使用如图13-4所示的缓冲区首部的b_reqnext域把这些缓冲区链接在一起。对于读操作来说，低级设备驱动程序可以选择先分配一个大的内存区来立即读取请求的所有扇区，然后再把数据拷贝到各个缓冲区。同样，对于写操作来说，低级设备驱动程序可以把很多不连续缓冲区中的数据拷贝到一个单独内存区的缓冲区中，然后再立即执行整个数据的传送。

图13-4说明包含3个块的一个请求描述符。其中的两个缓冲区在RAM中是连续的，而第三个缓冲区是独立的。相应的缓冲区首部表示块设备中的逻辑块。这些块必须是相邻的。每个逻辑块都包括两个扇区。请求描述符的sector域指向磁盘上第一个块的第一个扇区，每个缓冲区首部的b_reqnext域指向下一个缓冲区首部。

内核静态地分配固定数目的请求描述符来处理块设备的的所有请求：一共有NR_REQUST个描述符（通常是128个）存放在all_requests数组中。因为读操作的效

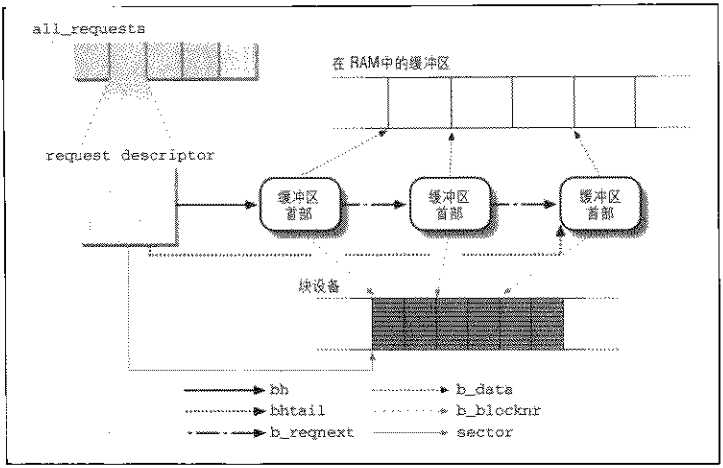


图 13-4 请求描述符和它的缓冲区及扇区

率对系统性能的影响要远远超过写操作对系统性能的影响(因为读数据可能需要进行一些计算),所以all_requests数组中倒数第三个请求描述符保留给读操作使用。

在严重负载和磁盘操作频繁的情况下,固定数目的请求描述符就可能成为一个瓶颈。空闲描述符的缺乏可能会强制进程等待直到正在执行的数据传送结束。因此,wait_for_request等待队列就用来对正在等待空闲请求描述符的进行排队。get_request_wait()试图获取一个空闲的请求描述符,如果没有找到,就让当前进程在等待队列中睡眠。get_request()函数与之类似,但是如果没有可用的空闲请求描述符,它只是简单地返回NULL。

请求队列和块设备驱动程序描述符

请求队列只是一个简单的链表,其元素是请求描述符。每个请求描述符中的next域都指向请求队列的下一个元素,最后一个元素为空。这个链表的排序通常是:首先根据设备标识符,其次根据最初的扇区号。

如前所述,对于所服务的每个硬盘,设备驱动程序通常都有一个请求队列。然而,

一些设备驱动程序只有一个请求队列，其中包括了由这个驱动器处理的所有物理设备的请求。这种方法简化了驱动程序的设计，但是损失了系统的整体性能，因为不能对队列强制使用简单排序的策略。

正在得到服务的请求地址以及另外几个相关的信息都存放在和每个块设备驱动程序相关的描述符中。这个描述符是一个blk_dev_struct类型的数据结构，其域如表13-4所示。所有块设备的描述符都存放在blk_dev表中，该表的索引就是块设备的主号。

表13-4 块设备驱动程序描述符的域

类型	域	说明
void *(*)(void)	request_fn	策略程序
void *	data	驱动程序的私有数据通用队列
struct request	plug	空插入请求
struct request *	current_request	单独通用队列中的当前请求
struct request **(*)(kdev_t)	queue	从队列中获得一个请求的方法
struct tq_struct	plug_tq	插入任务队列元素

对于所有的物理块设备来说，如果块设备驱动程序只有一个唯一的请求队列，那么queue域就为空，current_request域就指向队列中正在得到服务的请求的描述符。如果这个队列为空，那么current_request就是空。

反之，如果这个块设备驱动程序维护了多个队列，那么queue域就指向驱动程序的一个自定义方法，这个方法接收块设备文件标志符，并根据设备号来选择其中的一个队列，然后如果某个请求正在得到服务，那么就返回指向这个请求描述符的指针的地址。在这种情况下，current_request域指向正在得到服务的请求描述符。(最多一次只能有一个请求，因为同一个设备驱动程序不允许并发处理多个请求，即使这些请求涉及不同的磁盘。)

request_fn()域包含驱动程序的策略程序的地址，策略程序是低级块设备驱动程序的关键函数，为了开始传送队列中的一个请求所指定的数据，它与物理块设备(通常是磁盘控制器)真正打交道。

ll_rw_block()函数

ll_rw_block()函数产生块设备请求。正如我们在本章前面已经看到的一样，内核和设备驱动程序的很多地方都会调用这个函数。该函数接收以下参数：

- 操作类型rw，其值可以是READ、WRITE、READA或者WRITEA。后两种操作类型和前两种操作类型之间的区别在于，当没有可用的请求描述符时后两个函数不会阻塞。
- 要传送的块数nr。
- 一个bh数组，有nr个指针，指向说明块的缓冲区首部（这些块的大小必须相同，而且必须处于同一个块设备）。

缓冲区首部在前面已经被初始化，因此每个缓冲区首部都指定了块号、块大小和虚拟设备标志符（请参看前面的“缓冲区首部”一节）。所有的块都必须属于同一虚拟设备。

函数ll_rw_block()进入一个循环，处理bh数组的所有非空元素。对于每个缓冲区首部来说，它执行以下操作：

1. 检查块大小b_size是否和虚拟设备b_dev的大小匹配。
2. 设置实际设备标志符（通常只是把b_rdev设置成b_dev）。
3. 根据块号和块大小设置扇区号b_rsector。
4. 如果操作是WRITE或WRITEA，就检查确认这个块设备不是只读的。
5. 设置缓冲区首部的BH_Req标志来说明请求这个块的其他内核控制路径。
6. 调用make_request()函数，向它传递实际设备的主号、I/O操作的类型以及缓冲区首部的地址。

make_request()函数又要执行以下操作：

1. 设置缓冲区首部的BH_Lock标志。
2. 检查确认b_rsector没有超过块设备的扇区数。
3. 如果一定要读取这个块，就检查确认它已经不是有效的块（也就是说，

BH_Uptodate标志没有置位)。如果一定要写这个块，就检查确认该块确实是脏数据(也就是说，BH_Dirty标志已经置位了)。如果这两个条件都不成立，就直接返回而不用请求发送数据，因为这些数据已经没有什么用处了。

4. 禁止局部中断并获得io_request_lock自旋锁(请参看第十一章的"自旋锁"一节)。

5. 如果定义了queue方法，就调用它，或者读取块设备描述符中的current_request域来取得实际设备的请求队列地址。

6. 执行以下子步骤中的一步：

   a. 如果请求队列为空，就向其中插入一个新的请求描述符，并在稍后调度激活的策略程序。

   b. 如果请求队列不为空，就向其中插入一个新的请求描述符，并尽力将其和队列中的其他请求组织在一起。正如我们很快就会看到的一样，这种情况下没有必要调度激活的策略程序。

让我们来详细研究一下最后两个子步骤。

调度激活的策略程序

正如我们在前面已经看到的一样，推迟策略程序的激活有利于把相邻块的请求进行集中。这种延时是通过一种所谓设备插入(device plugging)和设备拔出(device unplugging)的技术来实现的。

如果实际的设备请求队列为空，而且设备还没有被插入，那么make_request()函数就执行设备插入。该函数把块设备驱动程序描述符的current_request域设置成一个哑请求描述符地址，也就是设置成同一个块设备描述符的plug域。然后该函数就分配一个新的请求描述符，并使用从缓冲区首部中读取出的信息对其进行初始化。接下来，make_request()把这个新请求描述符插入适当的实际设备的请求队列中。如果只有一个队列，就把这个请求插入队列.位置处于那个包含块设备描述符的plug域的哑元素之后。最后，make_request()把plug_tq任务队列描述符(静态包含在块设备驱动程序描述符中)插入tq_task任务队列中(请参看第四章的"下半部分"一节)，以使这个设备的策略程序稍后能被激活。实际上，任务队列元素指向的unplug_device()函数执行设备的策略程序。

内核要周期性地检查tq_disk任务队列是否包含了plug_tq任务队列元素。这种情况或者在诸如kswapd和bdflush之类的内核线程中发生，或者在内核等待一些有关块设备驱动程序的资源（例如缓冲区或请求描述符）时发生。在tq_disk检查的过程中，内核要删除队列中的所有元素并执行相应的unplug_device()函数。这种操作被认为是拔出设备。

扩充请求队列

如果请求队列不为空，那么低级块设备驱动程序就连续对请求进行处理，直到队列为空为止（请参看下一节），因此make_request()函数就不必调度激活的策略程序。

在这种情况下，make_request()对请求队列的修改或通过增加一个新元素，或者把新的请求与现有的元素进行合并。第二种情况称为块聚簇（block clustering）。

块聚簇的实现只适合特定块设备中的块，如EIDE和SCSI硬盘，软盘，还有一些其他设备。此外，只有满足以下所有条件时才可以在一个请求中包含一个块：

- 要插入的块和请求中的其他块属于同一块设备，而且它们彼此之间是相邻的，这个块或者位于请求的第一个块之前，或者位于请求的最后一个块之后。
- 请求的块和要插入的块具有相同的I/O操作类型（READ或WRITE）。
- 扩展的请求不超过允许的最大扇区数。最大值存放在max_sectors表中，该表是使用块设备的主号和次号进行索引的。缺省值是244扇区。
- 低级设备驱动程序目前没有对这个请求正进行处理。

make_request()函数对这个队列中的所有请求进行扫描。如果有一个请求能满足刚才所提到的条件，就把这个缓冲区首部插入请求队列，并更新这个request数据结构的域。如果这个块位于最后一个请求之后，那么该函数还要试图把这个请求和队列中的下一个元素合并。然后就不需要处理其他内容了，make_request()就释放io_request_lock自旋锁并结束。

反之，如果现有的请求不能包括这个块，那么make_request()就分配一个新的请求描述符（注6），并用从缓冲区首部中读出的信息对其进行适当的初始化。

注6：如果没有空闲的请求描述符，就挂起当前进程，直到有请求描述符被释放为止。

最后，make_request()调用add_request()函数，该函数根据新请求的初始扇区号把新请求插入请求队列的适当位置。然后释放io_request_lock自旋锁并结束。

低级请求处理

现在我们已经到了Linux块设备处理体系结构的最低层。该层是由策略程序实现的，策略程序与物理块设备之间相互作用以满足将队列中的请求聚集在一起的要求。

如前所述，在把新的请求插入到空的请求队列之后，策略程序通常才被启动。只要低级块设备驱动程序被激活，就应该对队列中的所有请求都进行处理，直到队列为空才结束。

策略程序理想的执行过程如下：对于队列中的每个元素，与块设备控制器相互作用共同为请求服务，等待直到数据传送完成，然后把已经服务过的请求从队列中删除，继续处理下一个请求。

这种处理过程效率并不高。即使假设可以使用DMA传送数据，策略程序在等待I/O操作完成的过程中也必须自行挂起，因此毫不相关的用户进程将受到严重的处罚。（策略程序不必代表请求I/O操作的进程执行，而是在随后随机执行，因为它是由tq_disk任务队列激活的。）

因此很多低级设备驱动程序都采用如下模式：

- 策略程序处理队列中的当前请求并设置块设备控制器，以便在数据传送完成时可以产生一个中断。然后策略程序就终止。
- 当块设备控制器产生中断时，中断处理程序就激活一个下半部分。这个下半部分的处理程序把这个请求从队列中删除并重新执行这个策略程序来处理队列中的下一个请求。

通常，这些低级块设备驱动程序都可以进一步被划分成以下两类：

- 为请求的每个块单独服务的驱动程序。
- 为请求的几个块一起服务的驱动程序。

交换高速缓存是由页高速缓存数据结构实现的，实现过程在第十四章中的“页高速缓存”一节中已经介绍过。回想一下页高速缓存中包含的页是与正规文件相关的，并且散列表允许算法快速从索引节点对象的地址和文件内部偏移量中导出页描述符的地址。交换高速缓存中的页在页高速缓存中以其他页的形式存放，并对其进行以下特殊的处理：

- 页描述符的inode域存放的是包含在swapper_inode变量中的虚拟索引节点对象地址。
- offset 域存放的是与该页相关的被换出页的标识符。
- 在 flags 域中的 PG_swap_cache 标志被设置。

还有，当该页被放入交换高速缓存时，该页描述符的count域和页插槽引用计数器都被增加，因为交换高速缓存既要使用页框，也要使用页插槽。

内核使用几个函数来处理交换高速缓存，这些函数主要是基于第十四章中的“页高速缓存”一节中的讨论。稍后我们将说明这些相对低层的函数是如何被高层函数调用来根据需要换入换出页的。

处理交换高速缓存的函数有：

in_swap_cache()

检查页的PG_swap_cache标志来确定该页是否属于交换高速缓存；如果属于，就返回存放在offset域中的被换出页的标识符。

lookup_swap_cache()

对作为参数传递的被换出页标识符进行操作并返回该页的地址，如果该页不在这个高速缓存中就返回0。该函数调用find_page()函数，把swapper_inode的虚索引节点对象和被换出页标识符的地址作为参数传递来查找所需要的页。如果该页在交换高速缓存中，lookup_swap_cache()就检测该页是否被锁定；如果被锁定，就调用wait_on_page()把当前进程挂起，直到该页被取消锁定为止。

add_to_swap_cache()

把页插入交换高速缓存中。该页描述符的inode和offset域分别被设置成swapper_inode虚索引节点对象的地址和这个被换出页标识符的地址。然后

件，那么就要增加请求描述符的sector域而减少nr_sectors域来记录要传送的块。

块设备DMA数据传送的结束与一个中断处理程序有关，而这个中断处理程序应该调用（直接或者通过一个下半部分）end_request()函数。如果数据传送成功，该函数接收的参数值为1；如果发生了错误，接收的参数值为0。end_request()执行以下操作：

1. 如果有错误发生（参数值为0），就修改sector和nr_sectors域，从而忽略块中剩余的扇区。在步骤3a中，缓冲区内容也被标记为未更新。
2. 把已被传送块的缓冲区首部从请求队列链表中删除。
3. 调用缓冲区首部的b_end_io方法。当getblk()函数分配缓冲区首部时，它用end_buffer_io_sync()函数的地址来装载缓冲区首部域，该函数执行两个操作：
   a. 根据数据传送的成功还是失败，把缓冲区首部的BH_Lock标志设置成1或0
   b. 清除缓冲区首部的BH_Lock标志，并唤醒缓冲区首部b_wait域所指向的等待队列中的睡眠进程
4. 如果请求队列中有另外一个缓冲区首部，就执行以下操作：
   a. 把请求描述符的current_nr_sectors域设置成新块的扇区数
   b. 把buffer域设置成新的缓冲区地址（从新缓冲区首部的b_data域中获得）
5. 否则，如果请求队列为空，就说明所有的块都已经处理完了。因此就执行以下操作：
   a. 把当前的请求指针设置成请求队列中的下一个元素
   b. 把处理过请求的rq_status域设置成RQ_INACTIVE
   c. 唤醒在wait_for_request等待队列中所有睡眠的进程

在调用end_request()之后，低级块设备驱动程序要检查这个块设备描述符的current_request域的值，如果该值不是NULL，那么就说明请求队列不为空，就

要再次执行这个策略程序。注意ered_request()实际上要执行两个嵌套循环：外部循环处理请求队列的元素，内部循环处理每个请求的缓冲区首部链表中的元素。因此，就要为请求队列中的每个块都调用策略程序。

页I/O操作

块设备一次传送一个块的信息，而进程地址空间（更确切地说，应该是为进程分配的线性区）被定义为页的集合。使用页I/O操作可以在某种程度上屏蔽这种不匹配（请参看“块设备的处理”一节）。在以下这些情况下可以激活页I/O操作：

- 进程对正规文件执行read()或write()系统调用（请参看第十五章的“读写正规文件”一节）。
- 进程读取在内存中对文件进行映射的一个页中的某个单元（请参看第十五章的“内存映射”一节）。
- 内核把与文件内存映射相关的一些脏页刷新到磁盘（请参看第十五章的“把内存映射的脏页刷新到磁盘”一节）。
- 在换进或换出时，内核从磁盘把几个页框的内容装载到内存或者把几个页框的内容保存到磁盘（请参看第十六章）。

在本章剩余的部分我们来介绍以下这些操作是如何执行的。

启动页I/O操作

页I/O操作是由brw_page()函数激活的，该函数接收以下参数：

rw
: I/O操作的类型（READ或WRITE）

page
: 页描述符的地址

dev
: 块设备号

b

逻辑块号的数组

size

块大小

bmap

说明b中的块号是否是使用索引节点操作的bmap方法来计算的（参看第十二章的“索引节点对象”一节）

页描述符指向页I/O操作中所涉及的页。在调用brw_page()之前，它必须已被加锁（PG_locked标志置位），以使其他内核控制路径不能访问它。页被划分成4096/size个缓冲区，页中的第*i*个缓冲区是与设备dev的块b[i]相关的。

brw_page()函数执行以下操作：

1. 调用create_buffers()来为页中所包含的所有缓冲区分配临时缓冲区首部（这种缓冲区首部被称为异步缓冲区首部；在第十四章的“缓冲区首部数据结构”一节中会对其进行介绍）。该函数返回第一个缓冲区首部的地址，而每个缓冲区首部的b_this_page域都指向该页中的下一个缓冲区的缓冲区首部。
2. 对于该页中的所有缓冲区首部，执行以下的子步骤：
   a. 对缓冲区首部的域进行初始化。因为这是一个异步缓冲区首部，因此就把b_end_io方法设置成end_bufer_io_async()。
   b. 如果bmap参数标志不为空，就检查缓冲区首部是否指向一个块号是0的块。这是因为索引节点操作的bmap方法使用块号0来表示一个文件洞（请参看第十七章）。在这种情况下，要用0填充缓冲区，将缓冲区首部的BH_Uptodate标志置位，然后继续处理下一个异步缓冲区首部。
   c. 调用find_buffer()来检查与缓冲区首部关联的块是否早已在内存（请参看第十四章的“缓冲区高速缓存”一节）。如果的确如此，就执行以下子步骤：
      1. 增加在高速缓存中找到的缓冲区首部的引用计数器。
      2. 如果这个I/O操作是READ，且高速缓存中的缓冲区不是最新的，那么就调用ll_rw_block()来发出一个READ请求，然后调用wait_on_

buffer()等待I/O操作完成。注意ll_rw_block()作用于缓冲区高速缓存中的缓冲区首部，也因此就触发一个缓冲区I/O操作。

3. 如果这个I/O操作是READ，就把数据从高速缓存中的缓冲区拷贝到页缓冲区。

4. 如果这个I/O操作是WRITE，就把数据从页缓冲区拷贝到高速缓存中的缓冲区，并调用mark_buffer_dirty()来对高速缓存中的缓冲区的BH_Dirty标志进行置位。

5. 将这个异步缓冲区首部的BH_Jpdate域置位，对高速缓存中的缓冲区首部的引用计数器减1，继续处理下一个异步缓冲区首部。

d. 所请求的块不在高速缓存中。因此，如果I/O操作是READ，就清除各个异步缓冲区首部的BH_Update标志；如果I/O操作是WRITE，就设置BH_Dirty标志。

e. 把指向这个异步缓冲区首部的指针插入一个局部数组中，然后继续处理下一个异步缓冲区首部。

到现在为止，所有的异步缓冲区首部都已被处理。

3. 如果这个局部异步缓冲区首部指针数组为空，就说明所有请求的块都在缓冲区高速缓存中，因此就没有必要执行I/O操作。在这种情况下，要执行以下子步骤：

a. 清除页描述符的PG_locked标志，从而将该页框解锁。

b. 将页描述符的PG_uptodate标志置位。

c. 唤醒在该页描述符等待队列中睡眠的所有进程。

d. 调用free_async_buffers()来释放异步缓冲区首部。

e. 调用after_unlock_page()函数（请参看第十六章）。如果这个页描述符的PG_free_after标志置位，那么该函数就释放这个页框。

f. 返回0。

4. 如果程序流程执行到此处，就说明异步缓冲区首部局部指针数组不为空，因此页I/O操作的执行是必须的。调用ll_rw_block()函数向这个局部数组中所包含的所有缓冲区首部发出一个rw请求，然后立即返回0。

终止页I/O操作

ll_rw_block()函数激活正要访问的块设备的设备驱动程序（请参看“ll_rw_block()函数”）。正如我们在前面“低级请求处理”中介绍的一样，这个设备驱动程序执行实际的数据传送，然后调用已被传送的所有异步缓冲区首部的b_end_io方法。b_end_io域指向end_buffer_io_async()函数，该函数执行以下操作：

1. 调用mark_buffer_uptodate()函数，该函数又要执行以下的子步骤：
   a. 根据I/O操作的结果来设置这个异步缓冲区首部的BH_Uptodate标志。
   b. 如果这个BH_Uptodate标志置位，就检查该页中的其他异步缓冲区首部是否都是最新的。如果是，就将这个页描述符的PG_uptodate标志置位。
2. 清除这个异步缓冲区首部的BH_Lock标志。
3. 如果BH_Uptodate标志没有设置，就将这个页描述符的PG_error标志置位，因为在传送这个块的过程中有错误发生。
4. 把这个异步缓冲区首部的引用计数器减1（该值变成0）。
5. 检查这个页中所有异步缓冲区首部的引用计数器是否都为0。如果是，就说明页中所有缓冲区的数据传送都已经完成，因此执行以下子步骤：
   a. 调用free_async_buffers()函数释放所有的异步缓冲区首部。
   b. 清除这个页描述符的PG_locked位，从而将该页框解锁。
   c. 唤醒这个页描述符等待队列中所有睡眠的进程。
   d. 调用after_unlock_page()函数（请参看第十六章）。如果这个页描述符的PG_free_after标志置位，那么该函数就释放该页框。

对Linux 2.4的展望

Linux 2.4对于I/O设备驱动程序的处理方式进行了相当大的修改。主要的改进包括使用了一个新的资源管理子系统来分配IRQ线、DMA通道、I/O端口等等。有了这个新子系统的帮助，Linux现在就可以完全支持热插拔即插即用硬件设备、USB总线以及PCMCIA卡了。

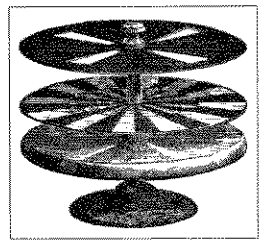
Linux 2.4 对块设备驱动程序层重新进行了组织，并加入对逻辑卷管理（Logical Volume Manager）的支持。逻辑卷管理允许文件系统跨越几个磁盘分区，还可以动态地改变大小。这种特性使 Linux 更接近于企业级的操作系统。

新内核引入了一种称为原始I/O设备（raw I/O device）的字符设备。这些设备允许如 DBMS 一类的应用程序不使用内核高速缓存而直接访问磁盘。

新增加的另一个显著特性是内核支持智能输入/输出（I2O）硬件。这种新硬件起源于PCI体系结构，其目的是可以为某些诸如磁盘、SCSI设备和网卡之类的设备编写独立于操作系统的设备驱动程序。

最后，Linux 2.4 包含的 devfs 虚拟文件系统代替了原来的静态 */dev* 设备文件目录。虚拟文件只有在对应的设备驱动程序存在于内核时才会出现。设备文件名也已被改变。例如，所有的磁盘设备都被放在*/dev/discs*目录下：*/dev/had*可能变成*/dev/discs/disc0*，而*/dev/hdb*可能变成了*/dev/discs/disc1*，依此类推。但是用户仍然可以通过正确地配置设备管理守护进程来使用原来的命名模式。

第十四章
磁盘高速缓存



本章涉及磁盘高速缓存，主要阐明Linux如何使用复杂的技术尽可能地减少磁盘访问来提高系统性能。

正如在第十二章的“通用文件模式”一节中提到的那样，磁盘高速缓存是一种软件机制，它允许系统把通常存放在磁盘上的一些数据保留在RAM中，以便对那些数据的进一步访问不用再访问磁盘而能尽快得到满足。

在Linux中，目录项高速缓存是VFS用来把文件路径名加速转换成对应的索引节点，除此之外，还有两种主要的磁盘高速缓存——缓冲区高速缓存和页高速缓存。本章大部分内容描述缓冲区高速缓存，最后的一小部分介绍页高速缓存。

在第十三章的“扇区、块和缓冲区”一节中，我们已了解到缓冲区是包含磁盘数据块的一个内存区。每个块的内容都是磁盘表面上的物理相邻字节，块大小依赖于块所在文件系统的类型。顾名思义，缓冲区高速缓存（buffer cache）是存放缓冲区的磁盘高速缓存。

相反，页高速缓存（page cache）是存放页框的磁盘高速缓存，而页框中的数据属于正规文件。页高速缓存和缓冲区高速缓存有本质的区别，因为页高速缓存中的页框不必对应物理上相邻的磁盘块。

缓冲区I/O操作（请参看第十三章的“块设备的处理”一节）只使用缓冲区高速缓存。页I/O操作使用页高速缓存，也可以选用缓冲区高速缓存。我们将在后面看到，这两种高速缓存的实现都使用适当的数据结构，这些数据结构存放着指向缓冲区首部的指针和页描述符的指针。

表14-1说明了一些广泛使用的I/O操作是如何使用缓冲区高速缓存和页高速缓存的。

表14-1 缓冲区高速缓存和页高速缓存的使用

I/O操作	高速缓存	系统调用	内核函数
读块设备文件[a]	缓冲区	read()	block_read()
写块设备文件[a]	缓冲区	write()	block_write()
读Ext2目录[b]	缓冲区	getdents()	ext2_bread()
读Ext2正规文件[b]	页	read()	generic_file_read()
写Ext2正规文件[b]	页、缓冲区	write()	ext2_file_write()
访问内存映射文件[c]	页	无	file_map_nopage()
访问交换出去的页[d]	页	无	do_swap_page()

a 请参看第十三章的“block_read()和block_write()函数”一节。

b 请参看第十七章。

c 请参看第十五章的“内存映射”一节。

d 请参看第十六章。

对于每类I/O操作，该表也列出了执行它所需要的系统调用（如果需要）及处理它所对应的主要内核函数。

你在表中会注意到，对于内存映射文件和交换出去页的访问不需要系统调用，它们对程序员是透明的。一旦文件内存映射已被建立，一旦交换已被激活，那么应用程序就可以访问映射的文件或换出的页，就好像它们在内存一样。内核的责任就是延迟进程，直到所需要的页已被确定在磁盘上并装入RAM中。

缓冲区高速缓存

缓冲区高速缓存所隐含的主要思想是把进程从存取慢速磁盘数据的等待中解放出来。因此，如果同时写入太多数据，结果反而是欲速则不达。反之，可以把数据划分成小片并周期性地写入磁盘，以使I/O操作对用户进程速度的影响最小，用户所感受的响应时间最少。

内核为每个缓冲区维护很多信息以有助于缓和写操作，这些信息包括一个“脏(dirty)”位，表示内存中的缓冲区已被修改，必须写到磁盘；还包括一个时间标志，

表示缓冲区被刷新到磁盘之前已经在内存中停留了多长时间。因为缓冲区的有关信息被保存在缓冲区首部(前一章已经介绍),所以,这些数据结构连同用户数据本身的缓冲区都需要维护。

缓冲区高速缓存的大小可以变化。当需要新缓冲区而现在又没有可用的缓冲区时,页框按需被分配。当空闲内存变得不足时,例如我们在第十六章中会看到的情况,就释放缓冲区并反复使用相应的页框。

缓冲区高速缓存包括两种数据结构:

- 描述高速缓存中缓冲区的一组缓冲区首部(请参看第十三章中的"缓冲区首部"一节)。
- 有助于内核快速找到缓冲区首部的散列表,而缓冲区首部描述了与一对给定的设备号和块号相关的缓冲区。

缓冲区首部数据结构

正如在第十三章的"缓冲区首部"一节中所提到的,每个缓冲区首部都存放在一个buffer_head类型的数据结构中,这些数据结构都有自己的叫做bh_cachep的slab分配器高速缓存,不要把它与缓冲区高速缓存本身弄混淆。slab分配器高速缓存是缓冲区首部对象使用的内存高速缓存(请参看第三章的"标识一个进程"一节),也就是说它不与磁盘打交道,而仅仅是有效管理内存的一种方式。

与之相对照,缓冲区高速缓存是缓冲区中的数据所使用的磁盘高速缓存。已分配的缓冲区首部的数目(也就是从slab分配器中所获得的对象的个数)存放在nr_buffer_heads变量中。

块设备驱动程序使用的每个缓冲区都必须有相应的缓冲区首部,以描述缓冲区的当前状态。反之则不成立,因为某个缓冲区首部可能没有被使用,也就是说它没有被限定到任何缓冲区上。内核要保留一定数目的未用缓冲区首部,从而避免频繁分配/回收内存而引起的过载。

通常,缓冲区首部可能为以下状态之一:

未用的缓冲区首部

该对象是可用的，但其域中的值没有什么意义。

空闲缓冲区的缓冲区首部

其b_data域指向一个空闲缓冲区，而b_dev域的值是B_FREE(0xffff)。注意这个缓冲区是可用的，而缓冲区首部本身是不可用的。

已用缓冲区的缓冲区首部

其b_data域指向一个存放在缓冲区高速缓存中的缓冲区。

异步缓冲区首部

其b_data域指向一个用来实现页I/O操作的临时缓冲区（请参看第十三章中的“页I/O操作”一节）。

严格说来，缓冲区高速缓存数据结构中只包括指向已用缓冲区的缓冲区首部指针。为了完整起见，我们将介绍内核处理各种类型的缓冲区首部所使用的数据结构和方法，而不仅仅介绍缓冲区高速缓存的缓冲区首部。

未用的缓冲区首部链表

所有未用的缓冲区首部都集中在一个简单的链表中，链表中第一个元素的地址存放在unused_list变量中。每个缓冲区首部都把下一个链表元素的地址存放在b_next_free域中。链表中当前元素的个数存放在nr_unuserd_buffer_heads变量中。

对缓冲区首部对象来说，未用缓冲区首部的链表就起主内存高速缓存的作用，而bh_cachep的slab分配器高速缓存就是辅内存高速缓冲。当缓冲区首部不再使用时，就将其插入未用缓冲区首部链表中。只有在未用缓冲区首部链表元素的个数超过MAX_UNUSED_BUFFERS（通常是36个元素）时，才会把一些缓冲区首部释放给slab分配器。换言之，可以把该链表中的缓冲区首部看作slab分配器已分配的一个对象，也可以看作缓冲区高速缓存还没有使用的一个数据结构。

该链表的NR_RESERVED（通常为16）个元素的子集保留给页I/O操作。内核使用这个子集来避免由于缺乏空闲缓冲区首部而产生的死锁。正如我们将在第十六章中所看到的，如果空闲内存不充足时，内核就试图把某一页交换到磁盘来释放一个页框。为了到达这个目的，至少还需要一个额外的缓冲区首部来执行页I/O文件操作。

如果交换算法不能成功获得缓冲区首部，那么它只能一直等待，并让文件的写入操作继续执行从而释放缓冲区，因为只要正在执行的文件操作一完成，至少就释放NR_RESERVED个缓冲区首部。

调用get_unused_buffer_head()函数来获得一个新的缓冲区首部。该函数实际上要执行以下操作：

1. 调用recover_reusable_buffer_heads()函数（在后面会详细介绍）。
2. 如果未用缓冲区首部链表的元素个数超过了NR_RESERVED，那么就从这个链表中移走一个元素并返回它的地址。
3. 否则，调用kmem_cache_alloc()函数来分配一个新的缓冲区首部。如果操作成功，就返回它的地址。
4. 没有空闲内存可以使用。这时，如果有一个缓冲区I/O操作已经对缓冲区首部发出请求，就返回NULL（失败）。
5. 如果程序流程执行到此，那么，一个页I/O操作肯定已发出对缓冲区首部的请求。如果未用缓冲区首部链表不为空，就移走其中的一个元素并返回它的地址。

put_unused_buffer_head()函数执行逆向操作，释放缓冲区首部。如果队列的元素个数少于MAX_UNUSED_BUFFERS，该函数就把释放的缓冲区首部插入未用缓冲区首部的链表中；否则，它就把释放的缓冲区首部释放给slab分配器。

空闲缓冲区的缓冲区首部链表

由于Linux有几种大小的块（请参看第十三章的“扇区、块和缓冲区”一节），所以它使用几个循环链表（每种缓冲区大小形成一个链表）来集中放置空闲缓冲区的缓冲区首部。这种链表的作用相当于内存高速缓存。正是由于这些链表，可以在需要时快速获取给定大小的空闲缓冲区，而不用依赖那个相当费时的伙伴系统。

系统为空闲缓冲区定义了7个链表，相应的缓冲区大小是512、1024、2048、4096、8192、16384和32768字节，但是，一个块的大小不能超过一个页框的大小，因此，在PC体系结构中实际上只能使用前面4个链表。

free_list数组指向这7个链表。对于每个链表，数组中就有一个元素来存放链表第一个元素的地址。BUFSIZE_INDEX宏接收一个块大小作为输入参数，并从中导出数组中的对应索引。例如，大小为512字节的缓冲区映射到free_list[0]，大小为1024的缓冲区映射到free_list[1]，依此类推。这些链表都是双向链表，通过每个缓冲区首部的b_next_free和b_prev_free域链接起来。

已用缓冲区的缓冲区首部链表

当一个缓冲区属于缓冲区高速缓存时，相应缓冲区首部的标志就描述它的当前状态（请参看第十三章中的“缓冲区首部”一节）。例如，当不在高速缓存中的块必须从磁盘中读入时，就分配一个新的缓冲区，并清除这个缓冲区首部的BH_Update标志，这是因为这个缓冲区的内容已经毫无意义。当从磁盘读取数据填充这个缓冲区时，又要对BH_Update标志进行置位来防止这个缓冲区被系统回收。如果读操作成功执行，那么BH_Update标志就被置位，而BH_Lock标志就被清除。如果这个块必须写入磁盘，就需要修改缓冲区的内容并设置BH_Dirty标志，该标志只有在缓冲区成功写入磁盘之后才被清除。

与用户缓冲区相关的任一缓冲区首部都处于一个双向链表中，这个双向链表是使用b_next_free和b_prev_free域实现的。一共有3个不同的链表，分别使用一个定义成宏的索引进行标识（BUF_CLEAN、BUF_DIRTY和BUF_LOCKED）。我们稍后介绍这3个链表。

这3个链表的引入是为了加速把脏缓冲区的内容刷新到磁盘的操作（请参看本章的“把脏缓冲区写入磁盘”一节）。出于效率的考虑，当缓冲区首部的状态发生变化时，并不是把缓冲区首部直接从一个链表中移到另一个链表中，这使得下面的介绍有些晦涩难懂。

BUF_CLEAN

这个链表集中存放干净缓冲区的缓冲区首部（没有置BH_Dirty标志）。注意这个链表中的缓冲区未必是最新的，也就是说，这些缓冲区未必包含有效数据。如果缓冲区不是最新的，那么就可能被加锁（置BH_Lock标志），并且在自己还处于这个链表时被选做从物理设备中读取数据。要保证该链表的缓冲区首部必须是干净的——换而言之，把脏数据刷新到磁盘上的操作根本就不会考虑这些缓冲区首部对应的缓冲区。

BUF_DIRTY

这个链表主要集中存放脏缓冲区的缓冲区首部，但这些缓冲区还没有被选中要把其中的数据写入物理设备，也就是说，块设备驱动程序的块请求中还没有包括这些脏缓冲区（BH_Dirty置位，而BH_Lock没有置位）。但是，这个链表也可以包括干净的缓冲区，因为在有些情况下，没有把脏缓冲区的内容刷新到磁盘上，也没有把相应缓冲区首部从这个链表中删除就把BH_Dirty标志清除（例如，没有执行卸载操作就把软盘从驱动器中拿走了——当然，这种事件可能会导致数据的丢失）。

BUF_LOCKED

这个链表主要集中存放脏缓冲区的缓冲区首部，但这些缓冲区已经被选定要把其中的数据写入块设备（BH_Lock置位。但BH_Dirty被清除，由于在一个块请求中包含这个缓冲区首部之前，add_request()函数重置了BH_Dirty标志）。但是，在某些加锁缓冲区的写操作完成时，低级块设备处理程序清除BH_Lock标志而不用把这个缓冲区首部从队列中删除（请参看第十三章的"低级请求处理"一节）。要保证这个链表中的缓冲区首部必须是干净的，否则，对应的脏缓冲区会被选定写入磁盘。

对于任意一个已用缓冲区的缓冲区首部来说，缓冲区首部的b_list域存放缓冲区所在链表的索引。lru_list数组（注1）存放每个链表中的第一个元素的地址，而nr_buffers_type数组则存放每个链表中元素的个数。

mark_buffer_dirty()和mark_buffer_clean()函数分别设置/清除一个缓冲区首部的BH_Dirty标志。这两个函数还要调用refile_buffer()函数，后者根据BH_Dirty和BH_Lock标志的值把缓冲区首部移动到适当的链表中。

已用缓冲区首部的散列表

属于缓冲区高速缓存的缓冲区首部地址被插入一个较大的散列表中。只要给定一个设备标识符和一个块号，内核就可以使用散列表快速取得相应缓冲区首部的地址（如果存在的话）。散列表能显著地提高内核的性能，因为对于缓冲区首部的检测是相当频繁的。在开始执行缓冲区I/O操作之前，内核必须检查所需要的块是否已经

注1：该数组的名字来自Least Recently Used的缩写。在早期的Linux版本中，这些链表是根据每个缓冲区最近被访问的时间排序的。

交换高速缓存是由页高速缓存数据结构实现的，实现过程在第十四章中的“页高速缓存”一节中已经介绍过。回想一下页高速缓存中包含的页是与正规文件相关的，并且散列表允许算法快速从索引节点对象的地址和文件内部偏移量中导出页描述符的地址。交换高速缓存中的页在页高速缓存中以其他页的形式存放，并对其进行以下特殊的处理：

- 页描述符的inode域存放的是包含在swapper_inode变量中的虚拟索引节点对象地址。
- offset 域存放的是与该页相关的被换出页的标识符。
- 在 flags 域中的 PG_swap_cache 标志被设置。

还有，当该页被放入交换高速缓存时，该页描述符的count域和页插槽引用计数器都被增加，因为交换高速缓存既要使用页框，也要使用页插槽。

内核使用几个函数来处理交换高速缓存，这些函数主要是基于第十四章中的“页高速缓存”一节中的讨论。稍后我们将说明这些相对低层的函数是如何被高层函数调用来根据需要换入换出页的。

处理交换高速缓存的函数有：

in_swap_cache()

检查页的PG_swap_cache标志来确定该页是否属于交换高速缓存；如果属于，就返回存放在offset域中的被换出页的标识符。

lookup_swap_cache()

对作为参数传递的被换出页标识符进行操作并返回该页的地址，如果该页不在这个高速缓存中就返回0。该函数调用find_page()函数，把swapper_inode的虚索引节点对象和被换出页标识符的地址作为参数传递来查找所需要的页。如果该页在交换高速缓存中，lookup_swap_cache()就检测该页是否被锁定；如果被锁定，就调用wait_on_page()把当前进程挂起，直到该页被取消锁定为止。

add_to_swap_cache()

把页插入交换高速缓存中。该页描述符的inode和offset域分别被设置成swapper_inode虚索引节点对象的地址和这个被换出页标识符的地址。然后

并调用该请求中所包括的所有缓冲区首部的b_end_io方法。当页I/O操作（而不是缓冲区I/O操作）中涉及到一个缓冲区时，end_request()域指向end_buffer_io_async()函数，后者把这个缓冲区首部的引用计数器减1，并检查该页中的所有缓冲区的引用计数器是否都为0。如果实际情况是这些缓冲区首部都不使用，那么这个函数就调用free_async_buffers()函数来释放这些异步缓冲区首部。注意异步缓冲区首部的引用计数器是被用作一个标志来说明缓冲区数据是否已经传送完。

但是，free_async_buffers()函数不能把这些异步缓冲区首部直接插入未用的链表，因为可能还有一个专用的块设备驱动程序的end_request()函数以后还需要访问这些缓冲区首部。因此，free_async_buffers()会把这些缓冲区首部插入一个叫作重用链表（reuse list）的特殊链表中，这个链表是使用b_next_free域来实现的。reuse_list变量指向该链表的第一个元素。在从未用链表取得一个缓冲区首部之前，recover_reusable_buffer_heads()函数就把这个重用链表中的元素移动到未用链表中。但是在end_request()执行完之前，这种情况是不会发生的，因此就不用担心对于这个重用链表的访问会产生竞争条件。

getblk()函数

getblk()函数是缓冲区高速缓存的主要服务例程。当内核需要读写物理设备上某个块的内容时，内核必须检查所请求缓冲区的缓冲区首部是否已包括在这个缓冲区高速缓存中。如果这个缓冲区还不在高速缓存中，那么内核必须在高速缓存中建立一个新项。为了做到这点，内核调用getblk()函数，其参数为设备标识符、块号以及块大小。该函数返回与这个缓冲区相关的缓冲区首部的地址。

回想一下，高速缓存中有缓冲区首部并不意味着这个缓冲区中的数据就是有效的。（例如，这个缓冲区有待从磁盘中读进来）。对块进行读取的函数[如block_read()]必须检查用getblk()函数所取得的缓冲区是否是最新的。如果不是，它就必须在使用这个缓冲区之前先把这个块从磁盘中读进来。

getblk()函数执行以下操作：

1. 调用find_buffer()，该函数使用散列表检查所需要的缓冲区首部是否已经在高速缓存中。

2. 如果在高速缓存中找到这个缓冲区首部，就增加它的引用计数器（b_count域）的值并返回这个缓冲区首部的地址。下一节解释这个域的用途。

3. 如果这个缓冲区首部不在高速缓存中，就必须分配一个新缓冲区和一个新缓冲区首部。从块大小得出free_list数组的一个索引，并检查free_list中相应的空闲链表是否为空。

4. 如果这个空闲链表不为空，就执行以下操作：

   a. 从这个链表中移走第一个缓冲区首部

   b. 用设备标识符、块号和块大小对缓冲区首部进行初始化；把一个指向end_buffer_io_sync()函数的指针保存在b_end_io域中（注2）；并把b_count引用计数器设置为1

   c. 调用insert_into_queues()，把这个缓冲区首部插入散列表和lru_list[BUF_CLEAN]链表中

   d. 返回这个缓冲区首部的地址

5. 如果这个空闲链表为空，就调用refill_freelist()函数对其进行补充（请参看下一节“缓冲区分配”）。

6. 调用find_buffer()，再次检查当内核控制路径等待上一步完成的过程中是否已经有其他进程在高速缓存中放入缓冲区。如果是，转到第2步；否则，转到第3步。

缓冲区引用计数器

缓冲区首部的b_count域是对应缓冲区所使用的一个引用计数器。在对缓冲区操作之前计数器加1，操作之后计数器减1。它主要起安全锁的作用，因为只要引用计数器不为0，内核就不会撤消这个缓冲区（或它的内容）。相反，或周期性地或当空闲内存不足时检测已用缓冲区，只有那些引用计数器为0的缓冲区才能被撤消（请参看第十六章）。换而言之，虽然引用计数器为0的一个缓冲区可以属于缓冲区高速缓存，但是，不能确定这个缓冲区在高速缓存中可以存在多长时间。

注2：这个缓冲区高速缓存是保留给缓冲区I/O操作的，缓冲区I/O操作需要把b_end_io方法指向end_buffer_io_sync()函数。异步缓冲区首部不会进入这个缓冲区高速缓存。

当内核控制路径想要访问一个缓冲区时，首先应该增加其引用计数器的值。这个任务是由getblk()函数来执行的，对这个函数的调用通常是为了查找缓冲区的位置，所以这个增加操作就不必由高层函数显式地执行。当内核控制路径停止访问缓冲区时，就可以调用brelse()或bforget()函数来减少相应引用计数器的值。

brelse()函数接受的参数为缓冲区首部的地址。该函数检查缓冲区是否为脏，如果是，就把这个缓冲区应该刷新到磁盘上的时间写进缓冲区首部的b_flushtime域（请参看下一节“把脏缓冲区写入磁盘”）。如果必要，该函数还要调用refile_buffer()把这个缓冲区首部移到适当的链表。最后，设置这个缓冲区所在页框的PG_referenced标志（见第十六章），并减少b_count域的值。

bforget()函数和brelse()函数类似，二者之间唯一的区别是如果引用计数器变成0而缓冲区没有加锁（BH_Lock标志被清除）时，bforget()把缓冲区首部从缓冲区高速缓存中删除并将其插入适当的空闲缓冲区链表中。换而言之，这个缓冲区中所包含的数据以及缓冲区与物理设备特定块之间的联系都被丢弃。

缓冲区的分配

出于效率的考虑，缓冲区并不是作为单个内存对象分配的。相反，缓冲区存放在称为缓冲区页（buffer page）的专用页中。在一个单独缓冲区页中的所有缓冲区必须有相同的大小。在PC体系结构中，根据块大小的不同，一个缓冲区页可以包括8、4、2甚至只有1个缓冲区。缓冲区首部的b_this_page域把一个单独缓冲区页中所包含的所有缓冲区链成一个循环链表。

如果页描述符指向一个缓冲区页，那么它的buffers域就指向该页中所包含的第一个缓冲区的缓冲区首部；否则，该域就被设置成NULL。图14-1说明了一个包含4个缓冲区以及相应的缓冲区首部的页。

缓冲区页的个数决不能变得太少，否则缓冲区I/O操作就会由于缺乏缓冲区而被延时。缓冲区页在所有页中所占的百分比存放在buffer_mem表的min_percent域中（注3），可以使用*proc/sys/vm/buffermem*或者使用sysctl()系统调用对其进行访问。

注3：该表还包括另外两个域：borrow_percent和max_percent，这两个域在Linux 2.2中没有使用。

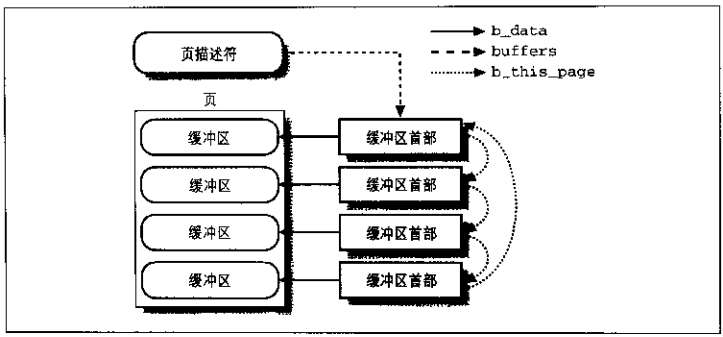


图14-1 包括4个缓冲区及其缓冲区首部的页

当getblk()函数需要一个空闲的缓冲区时，它就尽力从指向适当大小的空闲缓冲区链表中获得一个元素。如果该链表为空，那么内核必须分配另外的页框，并创建一个所请求大小的新缓冲区。这个任务是由refill_freelist()函数来执行的，该函数接收的参数为要分配缓冲区的块大小。实际上，该函数只是调用grow_buffers()，后者才实际分配新的缓冲区，如果成功就返回1，否则就返回0。如果grow_buffers()由于可用内存紧张而不能成功获得新缓冲区，那么refill_freelist()就唤醒bdflush内核线程（请参看下一节）。然后它通过设置current的SCHED_YIELD标志并调用schedule()而放弃CPU，这样就可以让bdflush在CPU上运行。getblk()函数会重复调用refill_freelist()，直到成功为止。

grow_buffers()函数接收的参数为要分配缓冲区的大小，它执行以下操作：

1. 以优先级GFP_BUFFER调用__get_free_page()以从伙伴系统中获得一个新页框。GFP_BUFFER优先级说明在执行这个函数时应该把当前进程挂起。
2. 如果没有可用页框，就返回0。
3. 如果有可用页框，就调用create_buffers()函数，该函数顺次执行以下操作：
   a. 通过重复调用get_unused_buffer_head()函数尽力为该页中的所有缓冲区分配缓冲区首部。

b. 如果获得了需要的所有缓冲区首部，就对这些缓冲区首部进行适当的初始化，特别是要把b_dev域设置成B_FREE，b_size域设置成缓冲区的大小，把b_data域设置成该页中缓冲区的起始地址，然后使用b_this_page域把这些缓冲区首部链接在一起。最后，返回该页中第一个缓冲区的缓冲区首部地址。

c. 如果没有获得全部的缓冲区首部，就通过重复调用put_unused_buffer_head()函数把已经获得的缓冲区首部释放掉。

d. 如果一个缓冲区I/O操作已经对这个缓冲区发出请求，就返回NULL（失败）。这将引起grow_buffers()函数返回0。

e. 如果我们执行到达此处，那么get_unused_buffer_head()函数就失败了，说明已经有一个页I/O操作请求了这个缓冲区。在这种情况下，未使用的链表就为空，链表中的所有NR_RESERVED异步缓冲区首部都正在用于其他的页I/O操作。执行tq_disk任务队列中的函数［请参看第十三章的"ll_rw_block()函数"一节］，在buffer_wait等待队列上睡眠，直到某个异步缓冲区首部变成空闲为止。

f. 返回步骤a，再次试图为一个页分配所有的缓冲区首部。

4. 如果create_buffers()返回NULL，释放该页框并返回0。

5. 否则，所有需要的缓冲区首部现在都可用。把与新缓冲区对应的缓冲区首部插入适当的空闲链表中。

6. 把新创建的缓冲区的个数加到nr_buffers，该域总是存放现有的缓冲区总数。

7. 把这个页描述符的buffers域设置成该页中的第一个缓冲区首部的地址。

8. 修改buffermem变量，该变量存放缓冲区页中字节的总数。

9. 返回1（成功）。

把脏缓冲区写入磁盘

Unix系统允许把脏缓冲区写入块设备的操作延迟执行，因为这种策略可以显著地提高系统的性能。对相应的磁盘块物理上只更新一次就可以满足对一个缓冲区所进行的几次写操作。此外，写操作没有读操作那么紧迫，因为进程通常是不会由于延迟

写而挂起，而大部分情况都因为延迟读而挂起。正是由于延迟写，任一物理块设备平均为读请求提供的服务将多于写请求。

一个脏缓冲区可能直到最后一刻（即直到系统关闭时）都一直逗留在主存。然而，从延迟写策略的局限性来看，它有两个主要的缺点：

- 如果发生了硬件错误或电源掉电的情况，那么就无法再获得RAM的内容，因此，从系统启动以来所对文件进行的很多修改就被丢失。
- 缓冲区高速缓存的大小（因此存放它所需RAM的大小）必须很大——至少要与所访问块设备的大小相同。

因此，在以下这些条件下把脏缓冲区刷新（写入）到磁盘：

- 缓冲区高速缓存变得太满，但还需要更多的缓冲区，或者脏缓冲区的数量越来越多。当这些条件之一发生时，就会激活*bdflush*内核线程。
- 自从缓冲区变成脏缓冲区已过去太长时间。*kupdate*内核线程周期性地刷新"年长（old）"的缓冲区。
- 进程请求刷新块设备的所有缓冲区或者特定文件的所有缓冲区。通过调用sync()、fsync()或fdatasync()系统调用来实现。

bdflush内核线程

*bdflush*内核线程（也称为kflushd）是在系统初始化的过程中创建的。它执行bdflush()函数，该函数选择一些脏缓冲区并强行对物理块设备上的相应块进行更新。

一些系统参数可以控制*bdflush*的行为，这些参数存放在bdf_prm表的b_un域中，对它们的访问或者通过*/proc/sys/vm/bdflush*文件，或者通过调用bdflush()系统调用。每个参数都有一个缺省的标准值，但是其取值在一个最小值和最大值之间变化，而最小值和最大值分别存放在bdflush_min和bdflush_max表中。表14-2给出了这些参数。我们知道一个时钟节拍大约是10毫秒（注4）。

注4：bdf_prm表还包括其他几个没有使用的域。

表14-2 缓冲区高速缓存的调整参数

参数	缺省值	最小值	最大值	说明
age_buffer	3000	100	60,000	正在写入磁盘的普通脏缓冲区超时的节拍数
age_super	500	100	60,000	正在写入磁盘的超级块脏缓冲区超时的节拍数
interval	500	0	6000	两次激活*kupdate*之间延迟的节拍数
ndirty	500	10	5000	在激活*bdflush*过程中写入磁盘的脏缓冲区的最大数
nfract	40	0	100	为唤醒*bdflush*，脏缓冲区的百分比上限

为了有效地实现写延迟技术，如果想一次写入很多数据反而会欲速则不达，这会降低系统的响应时间，还不如缓冲区一脏就把它写入磁盘。因此，不要在一次激活*bdflush*时把所有的脏缓冲区都写入磁盘。在每次激活*bdflush*时刷新的脏缓冲区的最大个数存放在bdf_prm的ndirty参数中。

在以下几种特殊情况下会唤醒这个内核线程：

- 当一个缓冲区首部被插入BUF_DIRTY链表并且该链表中的元素个数超过：

  ```
  nr_buffers * bdf_prm.b_un.nfract / 100
  ```

  时，也就是说，脏缓冲区的百分比超过了nfract系统参数所表示的上限。

- 正如前面“缓冲区的分配”一节中所介绍的，当refill_freelist()调用grow_buffers()函数失败，不能补充一个空闲缓冲区链表时。

- 当内核试图通过释放缓冲区高速缓存中的一些缓冲区来获得某些空闲页时(请参看第十六章)。

- 当用户在控制台上按下一些特殊的组合键（通常是ALT+SysRq+U和ALT+SysRq+S）时。只有以Magic SysRq Key选项编译Linux内核时才启用这些组合键，这些组合键允许Linux高手清楚地控制内核的行为。

为了唤醒*bdflush*，内核调用wakeup_bdflush()函数。它接收的参数是一个wait标志，这个标志表示正在调用的内核控制路径是否希望一直等待，直到某些缓冲区已经成功刷新到磁盘为止。该函数执行以下操作：

1. 调用wake_up()来唤醒在bdflush_wait任务队列中被挂起的进程。在这个等待队列中只有一个进程，就是*bdflush*自己。
2. 如果这个wait参数表示正在调用的进程希望等待，就调用sleep_on()来把当前进程插入一个名为bdflush_done的特定等待队列。

每次激活*bdflush*内核线程时，bdflush()函数执行以下操作：

1. 把局部变量ndirty初始化成0。该变量说明了在一次激活bdflush()的过程中写入磁盘的脏缓冲区的个数。
2. 扫描BUF_DIRTY和BUF_LOCKED的缓冲区首部链表。
3. 如果ndirty小于bdf_prm.b_un.ndirty，且没有其他要检查的缓冲区首部，则返回到第2步。
4. 调用run_task_queue()执行tq_disk 任务队列中的函数，由此开始执行有效的低级块设备驱动程序。
5. 调用wake_up()唤醒在bdflush_done等待队列被挂起的所有进程。
6. 如果在这次循环中有些缓冲区已被刷新，且脏缓冲区的百分比大于bdf_prm.b_un.nfract，则回到第1步并开始新一轮的循环，说明缓冲区高速缓存还包含太多的脏缓冲区。
7. 否则，按如下方式挂起*bdflush*内核线程：调用flush_signals()刷新*bdflush*所有挂起的信号，并调用interruptible_sleep_on()把*bdflush*插入bdflush_wait等待队列。当这个内核线程被唤醒时，它将从第1步重新恢复它的执行。

kupdate内核线程

通常只有在脏缓冲区太多或者需要太多缓冲区而可用内存很紧张时，*bdflush*内核线

程才会被激活，所以脏缓冲区在被刷新到磁盘上之前，可能已在RAM中停留了任意长的时间。由此引入的*kupdate*内核线程就是刷新年长的脏缓冲区（注5）。

内核把磁盘超级块所使用的缓冲区与其他缓冲区加以区分。超级块包括很重要的信息，它的崩溃可能引起严重的问题，事实上，整个分区都可能变为不可读。如表14-2中所示，有两个超时参数：age_buffer是在*kupdate*把这些正常缓冲区写入磁盘之前它们已经存在的时间（通常是30秒），而age_super是超级块的相应时间（通常是5秒）。

bdf_prm表的interval域存放两次激活*kupdate*内核线程之间的时间间隔的节拍数（通常是5秒）。如果该域为空，这个内核线程通常被停止，只有接收到一个SIGCONT信号时它才会被激活。

当内核修改某些缓冲区内容时，就把相应缓冲区首部的b_flushtime域设置成这个缓冲区稍后应该被刷新到磁盘上的时间（以jiffies为基础）。*kupdate*内核线程只选择那些b_flushtime域小于jiffies当前值的那些脏缓冲区。

*kupdate*内核线程由kupdate()函数组成，该函数执行下面的死循环：

```
for (;;) {
    if (bdf_prm.b_un.interval) {
        tsk->state = TASK_INTERRUPTIBLE;
        schedule_timeout(bdf_prm.b_un.interval);
    } else {
        tsk->state = TASK_STOPPED;
        schedule(); /* 等待 SIGCONT */
    }
    sync_old_buffers();
}
```

如果bdf_prm.b_un.interval不为空，那么该线程把自己挂起指定的节拍数（请参看第五章中的“动态定时器的应用”一节）；否则，该线程自己停止，直到接收到一个SIGCONT信号为止（请参看第九章中的“信号的作用”一节）。

注5：在早期的Linux 2.2版本中，相同的任务是使用bdflush()系统调用来实现的。在系统启动时装入系统的一个用户态系统进程，每5秒钟就会调用一次这个系统调用，这个系统调用执行的是*/sbin/update*程序。在最近的内核版本中，bdflush()系统调用只用来允许用户修改bdf_prm表中的系统参数。

kupdate()函数的核心由sync_old_buffers()函数构成。对于Unix所使用的标准文件系统来说，要执行的操作非常简单，函数所要做的全部事情就是把脏缓冲区写入磁盘。但是，有些外部文件系统却引入了一些复杂性，因为这些文件系统以复杂的方式存放自己的超级块或索引节点信息。sync_old_buffers()函数执行以下操作：

1. 调用sync_supers()，访问super_blocks数组来扫描当前已装载的所有文件系统中的超级块（请参看第十二章中的“文件系统安装”一节）。然后如果定义了超级块，就为每个超级块都调用相应的write_super超级块操作（请参看第十二章中的“超级块对象”一节）。
2. 调用sync_inodes()，扫描当前已装载的所有文件系统的超级块，并扫描每个超级块对象的s_dirty域所指向的脏索引节点链表。如果定义了write_superblock超级块方法，那么这个函数对链表中的每个元素都调用这个方法。
3. 扫描BUF_DIRTY和BUF_LOCKED链表，并把所有的脏缓冲区（也就是那些缓冲区首部的b_flushtime域小于或等于jiffies的缓冲区）写入磁盘。执行这个步骤的代码基本上和bdflush()所使用的代码是相同的，但是sync_old_buffers()不会把年轻的缓冲区刷新到磁盘上，而且它没有限制每次激活时检查的缓冲区个数。
4. 执行tq_disk任务队列中的函数，从而启动（拔出）把数据块写入磁盘所需要的任一低级块设备驱动程序。

sync()、fsync()和fdatasync()系统调用

用户应用程序可以使用三个不同的系统调用把脏缓冲区刷新到磁盘上：

sync()
: 通常是在关机之前执行，因为该函数把所有的脏缓冲区都刷新到磁盘上

fsync()
: 允许进程把属于一个特定打开文件的所有块都刷新到磁盘上

fdatasync()
: 和fsync()非常类似，只是不刷新该文件的索引节点块

sync()系统调用的核心是fsync_dev()函数，该函数执行以下操作：

1. 调用sync_buffers()，该函数扫描BUF_DIRTY和BUF_LOCKED链表，并通过ll_rw_block()向这两个链表中包含的没有锁定的所有脏缓冲区发出一个WRITE请求
2. 如果必要，使用write_super方法调用sync_supers()把脏超级块写入磁盘（请参看本节前面的内容）
3. 如果必要，使用write_inode方法调用sync_inodes()把脏索引节点写入磁盘（请参看本节前面的内容）
4. 再次调用sync_buffers()，因为sync_supers()和sync_inodes()可能把其他缓冲区已标记成脏

fsync()系统调用强制内核把fd文件描述符所指文件的所有脏缓冲区(如果需要，也包括索引节点所在的缓冲区)写入磁盘。系统服务例程获得这个文件对象的地址，然后调用它的fsync方法。这个方法是依赖于文件系统的，因为它必须知道文件在磁盘上是如何存放的，以便能够识别与一个给定文件相关的脏缓冲区。文件和缓冲区之间的对应关系一旦建立，其余的工作就可以全权委托给ll_rw_block()。fsync方法把调用进程挂起，直到该文件的所有脏缓冲区都已经写入磁盘为止。为了实现这个功能，它要扫描BUF_DIRTY和BUF_LOCKED链表，并对每个所找到的锁定缓冲区调用wait_on_buffer()。

fdatasync()系统调用与fsync()非常类似，但是它只会把包含文件数据的缓冲区写入磁盘，而不会把包含索引节点信息的那些缓冲区写入磁盘。由于Linux 2.2没有针对fdatasync()的一个特定文件方法，所以这个系统调用就使用fsync方法，因而也就与fsync()系统调用相同。

页高速缓存

页高速缓存比缓冲区高速缓存简单得多，它是页I/O操作访问数据所使用的磁盘高速缓存。正如我们在第十五章中会看到的一样，read()、write()和mmap()系统调用对正规文件的访问都是通过页高速缓存来完成的。当然，高速缓存中所保留的信息单元是一个整页，因为页I/O操作要传输整页数据。一个页未必是包含物理上相邻的磁盘块，因此就不能使用设备号和块号来标识页。相反，页高速缓存中一个页的标识是通过文件的索引节点和文件中的偏移量达到的。

与页高速缓存有关的操作主要有三种：当访问的文件部分不在高速缓存中时增加一页，当高速缓存变得太大时删除一页，以及查找一个给定文件偏移量所在的页。

页高速缓存的数据结构

页高速缓存使用两个主要的数据结构：

页散列表（page hash table）

对于与指定的索引节点和文件偏移量相关的页，让内核快速获得该页的页描述符的地址

索引节点队列（inode queue）

一个特定文件的数据页所对应的页描述符链表（由唯一的索引节点来区分）

页高速缓存的操作涉及对这些数据结构增加和删除元素，还要对已在高速缓存中的文件对应的所有索引节点对象的域进行更新。

页散列表

当进程读一个大型文件时，就可能用与该文件相关的页来填充页高速缓存。在这种情况下，就得扫描适当的索引节点队列以查找对所需要的文件部分进行映射的页，这个过程可能变为一个费时的操作。

由于这个原因，Linux使用一个叫做page_hash_table的散列表页描述符指针。该表的大小依赖于可用RAM的数量，例如，对于有64MB的RAM系统来说，page_hash_table被存放在16个页框中，包括16384个页描述符指针。

page_hash()函数从一个索引节点对象的地址和一个偏移量中导出相应元素在这个散列表中的地址。系统照样引入一个链表来处理那些产生冲突的项：那些散列值相同的项使用页描述符的next_hash和pprev_hash域实现一个双向链表。page_cache_size变量说明了页散列表（因此也在页高速缓存中）的冲突链表中所包含的页描述符的个数。

add_page_to_hash_queue()和remove_page_from_hash_queue()函数分别用来向散列表中加入一个元素和从散列表中删除一个元素。

索引节点队列

在内核内存中每个索引节点对象都有一个页队列。每个索引节点对象的i_pages域存放的是它的索引节点队列中第一个页描述符的地址，而i_nrpages域存放该队列的长度。

add_page_to_inode_queue()和remove_page_from_inode_queue()函数分别被用来把一个页描述符插入一个索引节点队列和把一个页描述符从一个索引节点队列中删除。

与页高速缓存有关的页描述符域

当一个页框包含在页高速缓存中时，相应页描述符的一些域就具有特殊的意义：

inode

包含一个文件的索引节点对象的地址，该页中所包含的数据就属于这个文件。如果该页不属于页高速缓存，那么这个域就是NULL（注6）。

offset

指定文件内数据的相对地址。

next

指向索引节点队列的下一个元素。

prev

指向索引节点队列的上一个元素。

next_hash

指向页散列表中的下一个冲突页描述符。

pprev_hash

指向页散列表中的上一个冲突页描述符。

另外，当一个页框被插入页高速缓存时，相应页描述符的引用计数器（count域）就被递增。如果count域正好是1，那么该页框属于这个高速缓存，但还没有被任

注6：正如我们在第十六章中会看到的一样，当该页包含交换分区中的数据时，inode域指向一个假想的索引节点对象。实际上，该页属于一个名为“交换高速缓存”的页高速缓存的子集。在本章中，我们就不考虑这种特例了。

交换高速缓存是由页高速缓存数据结构实现的，实现过程在第十四章中的“页高速缓存”一节中已经介绍过。回想一下页高速缓存中包含的页是与正规文件相关的，并且散列表允许算法快速从索引节点对象的地址和文件内部偏移量中导出页描述符的地址。交换高速缓存中的页在页高速缓存中以其他页的形式存放，并对其进行以下特殊的处理：

- 页描述符的inode域存放的是包含在swapper_inode变量中的虚拟索引节点对象地址。
- offset 域存放的是与该页相关的被换出页的标识符。
- 在 flags 域中的 PG_swap_cache 标志被设置。

还有，当该页被放入交换高速缓存时，该页描述符的count域和页插槽引用计数器都被增加，因为交换高速缓存既要使用页框，也要使用页插槽。

内核使用几个函数来处理交换高速缓存，这些函数主要是基于第十四章中的“页高速缓存”一节中的讨论。稍后我们将说明这些相对低层的函数是如何被高层函数调用来根据需要换入换出页的。

处理交换高速缓存的函数有：

in_swap_cache()

检查页的PG_swap_cache标志来确定该页是否属于交换高速缓存；如果属于，就返回存放在offset域中的被换出页的标识符。

lookup_swap_cache()

对作为参数传递的被换出页标识符进行操作并返回该页的地址，如果该页不在这个高速缓存中就返回0。该函数调用find_page()函数，把swapper_inode的虚索引节点对象和被换出页标识符的地址作为参数传递来查找所需要的页。如果该页在交换高速缓存中，lookup_swap_cache()就检测该页是否被锁定；如果被锁定，就调用wait_on_page()把当前进程挂起，直到该页被取消锁定为止。

add_to_swap_cache()

把页插入交换高速缓存中。该页描述符的inode和offset域分别被设置成swapper_inode虚索引节点对象的地址和这个被换出页标识符的地址。然后

们会在第十六章中看到的一样，当空闲内存紧张时，内核就通过释放最老的未用页来裁剪页高速缓存。

但是，页高速缓存的大小一定不能小于预定义的下限，否则系统性能就会急剧下降。页高速缓存大小的下限可以使用存放在page_cache表(注7)中的min_percent参数进行调整，该参数定义了在属于页高速缓存的所有页框中这些页所占的最小百分比。该参数的缺省值是2%，可以通过调用sysctl()系统调用或者通过访问*/proc/sys/vm/pagecache*文件读取或者修改它。

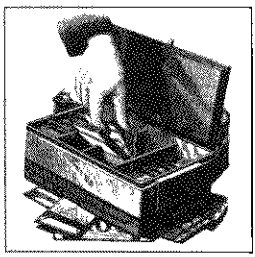
对 Linux 2.4 的展望

对于页高速缓存已经进行了很多工作。首先，页高速缓存使用高端内存中更适宜的页框。此外，Linux 2.4引入一种用来表示文件地址空间的新对象，该对象涉及一个索引节点的给定块(或者一个块设备)，并且包含几个指针，分别指向对该文件进行映射的线性区描述符和包含该文件数据的页。这个索引节点对象现在包含一个指向新地址空间对象的指针，并且对页高速缓存的索引是通过把这个地址空间对象的基地址与文件内部的偏移量结合起来获得的。

这个地址空间对象包含读写一个完整的数据页的方法。这些方法考虑索引节点对象的管理(例如更新文件访问时间)、页高速缓存处理以及临时缓冲区的分配。这种方法使页高速缓存和缓冲区高速缓存之间可以很好地结合。大部分文件系统由此可以使用generic_file_write()函数(正如你将在第十五章中看到的一样，Linux 2.2只为网络文件系统使用这个函数)。缓冲区高速缓存和页高速缓存之间的同步问题已经解决，因为对于正规文件的读写操作会使用相同的页高速缓存，因此就没有必要同步这两个高速缓存中现有的数据。

注意缓冲区高速缓存和页高速缓存都继续用于不同的目的。前者用于磁盘块，后者用于在磁盘上有文件映象的页。

注7：page_cache表还包括borrow_percent和max_percent参数，这两个参数现在已经不再使用了。



第十五章
访问正规文件

访问正规文件是一种复杂的行为，即要涉及VFS抽象层（第十二章，块设备的处理（第十三章），也涉及磁盘高速缓存的使用（第十四章）。本章介绍内核是如何使用这些技术实现文件的读及写的。本章所涉及的主题适用于正规文件，这些文件或者存放在基于磁盘的文件系统中，或者用于网络文件系统（如NFS或Samba）。

在对一个特定文件适当的读或写方法进行调用之后（正如第十二章中所介绍的），我们就开始本章内容的介绍。在这里，我们会说明每个读操作最终是如何把所需要的数据传递给用户态进程的，以及每个写操作最终又是如何把这些数据标志为准备好以准备传送到磁盘上的。其他传送过程是使用第十二章和第十四章中的技术来处理的。

尤其是，在“读写正规文件”一节中，我们会介绍如何使用`read()`和`write()`系统调用来访问正规文件。当进程从正规文件中读取数据时，数据首先被从磁盘移动到内核地址空间的一组缓冲区中。这组缓冲区包含在页高速缓存的一组页中（请参看第十三章中的“页I/O操作”一节）。接下来，这些页被拷贝到进程的用户地址空间。本章只涉及从内核空间到用户地址空间的移动。写操作的过程则完全相反，但是有些步骤和读操作有很大的区别。

我们在“内存映射”中还会讨论内核是如何允许进程直接把一个正规文件映射到自己的地址空间中的，因为在内核内存中对页进行处理时也要执行相同的操作。

读写正规文件

在第十三章的“block_read()和block_write()函数”一节中已经说明了read()和write()系统调用是如何实现的。相应的服务例程最终会调用文件对象的read和write方法，这两个方法也许是依赖于文件系统的。对于基于磁盘的文件系统来说，这些方法能够确定正被访问的数据所在物理块的位置，并激活块设备驱动程序开始执行数据传送。但是，在Linux中读写操作的执行是不同的。

对正规文件的读操作是基于页的，内核总是一次传送完整的数据页。如果进程发布read()系统调用来读取几个字节，而这些数据还不在RAM中，那么，内核就要分配一个新页框，并使用这个正规文件的适当部分来填充这个页，把该页加入页高速缓存，最后把所请求的字节拷贝到进程地址空间中。对于大部分文件系统来说，从一个正规文件中读取一个数据页就等同于在磁盘上查找所请求的数据存放在哪些块上。只要这个过程完成了，内核就可以使用一个或多个页I/O操作来填充这些页。

对基于磁盘的文件系统来说，写操作的处理相当复杂，因为文件大小可以改变，因此内核可能会分配或释放磁盘上的一些物理块。当然，这个过程到底如何实现要依赖于文件系统的类型。

实际上，大部分文件系统的read方法是由一个名为generic_file_read()的通用函数来实现的。但是，所有基于磁盘的文件系统都有一个专用的write方法。由于第二扩展文件系统（Ext2）是现在Linux可以使用的效率最高、功能最强大的一种文件系统，又由于Ext2是大部分Linux系统的标准文件系统，因此我们会在第十七章中介绍诸如write之类的方法是如何实现的。只有NFS和Samba才会使用generic_file_write()函数，因为这两种文件系统是网络文件系统，内核并不关心数据在异地磁盘上如何物理地存放。

从正规文件读取数据

让我们讨论一下generic_file_read()函数，该函数实现了大部分文件系统的正规文件的read方法。该函数使用以下参数：

file

文件对象的地址

buf

用户态内存区的线性地址，从文件中读出的数据必须存放在这里

count

要读取的字符个数

ppos

指向一个变量的指针，存放着距离读操作开始处的文件偏移量

该函数要调用access_ok()来验证这些参数是否正确（请参看第八章中的“验证参数”一节），然后调用do_generic_file_read()，后者执行以下步骤：

1. 从*ppos确定距离读操作开始处的文件偏移量是在该文件的预读窗口之内还是之外（请参看下一节）。

2. 开始执行一个循环来读取所请求的count个字符所在的所有页，把pos局部变量初始化成*ppos的值。在一次单独的循环中，该函数通过执行以下子步骤来读取一页的数据：

   a. 如果pos超过文件大小，就退出循环，跳到第3步。

   b. 调用find_page()来检查该页是否已经在页高速缓存中。

   c. 如果该页尚未在页高速缓存中，就分配一个新页框，将其加入页高速缓存中，并调用索引节点对象的readpage方法来填充该页。虽然其实现要依赖于文件系统，但是大部分磁盘文件系统都依赖于一个通用的generic_readpage()函数，后者执行以下操作：

      1. 把该页的PG_locked标志置位，这样其他内核控制路径就不能访问该页的内容了。

      2. 增加该页描述符的引用计数器。这是一种安全失效机制，如果要读取该页内容的进程在睡眠时被杀死，那么就确保该页框不会释放给伙伴系统。

      3. 把PG_free_after标志置位，这样就可以保证在这个页I/O操作结束时减少页描述符的引用计数器（请参看第十三章中的“页I/O操作”一节）。这是一个必要的标志，因为设备驱动程序不用首先增加引用计数器的值就可以调用brw_page()函数。但是现在没有设备驱动程序是这样处理的。

4. 计算填充该页所需要的块数，并从相对于该页的文件偏移量中导出该页中第一个块的文件块号。

5. 对每个文件块号，调用索引节点操作表的bmap方法来获得相应的逻辑块号。

6. 调用brw_page()来传送页中的块（请参看第十三章中的“启动页I/O操作”一节）。

d. 调用generic_file_readahead()函数（请参看下一节）。

e. 如果该页被加锁了，就调用wait_on_page()等待I/O操作完成。

f. 把该页（或者该页的一部分）拷贝到进程地址空间中，把pos修改为指向文件中下一次读取的位置，并跳到步骤2a继续处理下一个请求的页。

3. 把pos的当前值赋给*ppos，这样就保存了下一次调用该函数时应该读取的位置。

4. 把该文件描述符的f_reada域设置成1（请参看下一节）。

5. 调用update_atime()把当前时间存放在这个文件索引节点的i_atime域中，并把这个索引节点标记成脏。

对正规文件进行预读

在第十三章的“预读的作用”一节中，我们已经讨论了磁盘的顺序访问受益于预读块设备的几个相邻块。同样的考虑也适用于正规文件的顺序读操作。但是，正规文件的预读需要的算法比物理块的预读需要的算法更复杂，这是由于以下几个原因：

- 由于数据是逐页进行读取的，因此预读算法不必考虑页内偏移量，只要考虑所访问的页在文件内部的位置就可以了。如果相关的页彼此之间比较邻近，那么可以认为对同一文件中的页进行访问的次序是顺序的。我们很快就会更确切地定义“邻近（close）”这个词。

- 当前的访问与上一次访问不是顺序的时（随机访问），预读就必须从头开始重新执行。

- 当进程一直反复地访问同一页时（该文件只有很少的一部分被使用），应该减慢预读的速度甚至停止执行。

- 如果需要，预读算法就必须激活低级I/O设备驱动程序来确保新页会被读取。

现在我们试图大概描绘一下Linux是如何实现预读的。然而，我们不会详细介绍所使用的算法，因为其背后的动机看起来有点经验主义。

预读算法把文件中的一个邻近部分所对应的一组页标识成预读窗口（read-ahead window）。如果进程产生的下一个读操作在正好落在这组页中，那么内核就认为这次文件访问和上一次文件访问是顺序的。预读窗口由进程所请求的页组成，或者由内核预读的页以及页高速缓存中的页组成。预读窗口通常包括最后一次预读操作所请求的页，我们把这些页称之为预读组（read-ahead group）。预读窗口或预读组中的所有页没必要都是最新的。如果从磁盘上传送这些页还没有完成，这些页就是无效的（即其PG_uptodate标志被清除）。

文件对象包括以下与预读有关的域：

f_raend
: 在预读组和预读窗口之后的第一个字节位置

f_rawin
: 当前预读窗口的长度，以字节为单位

f_ralen
: 当前预读组的长度，以字节为单位

f_ramax
: 下一个预读操作的最大字符数

f_reada
: 一个标志，说明文件是否已被顺序访问［只有在访问块设备文件时才会使用；请参看第十三章中的“block_read()和block_write()函数”一节］

图15-1举例说明如何使用一些域来划分预读窗口和预读组。generic_file_readahead()函数实现了预读算法，该函数的整体模式在后面的图15-3中给出。

预读操作

内核通过多次调用一个名为try_to_read_ahead()的函数来执行预读操作（read-ahead operation），一次预读一页。

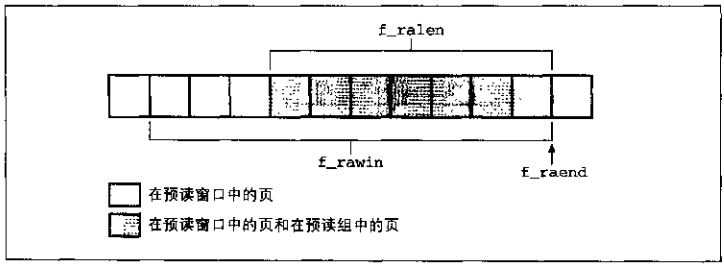


图 15-1 预读窗口和预读组

要预读的字节数存放在文件对象的 f_ramax 域中。这个数字最初被设置成 MIN_READAHEAD(通常是3页),该值不能超过MAX_READAHEAD(通常是31页)。

try_to_read_ahead()函数检查所考虑的页是否早就包含在页高速缓存中。如果没有,就调用相应索引节点对象的 readpage 方法传送该页。然后,这个函数就把 f_ramax域的值翻倍,这样下一次预读操作就更有针对性(访问看起来是真正的顺序的)。正如前面已经介绍的一样,该域不允许大于 MAX_READAHEAD。

接下来,generic_file_readahead()函数就更新文件的预读窗口和预读组。正如我们后面会看到的一样,该函数可以设置一个短预读窗口,或者一个长预读窗口:前者只包括最近的那个预读组,而后者则包括两个最近预读组(见图 15-2)。

最后,generic_file_readahead()函数可以通过执行 tq_disk 任务队列来激活低级块设备驱动程序(请参看图 15-3)。

第一次执行预读操作时会发生一个特例,此时上一个预读窗口和预读组都为空(所有的相关域都被设置成0)。第一次预读操作要预读的字节数等于MIN_READAHEAD 或者进程在 read()系统调用中所请求的字节数,后者的值更大。

非顺序访问(在预读窗口之外)

当进程通过 read()系统调用发出一个读请求时,内核要检查所请求的数据的第一页是否包含在相应文件对象的当前预读窗口中。

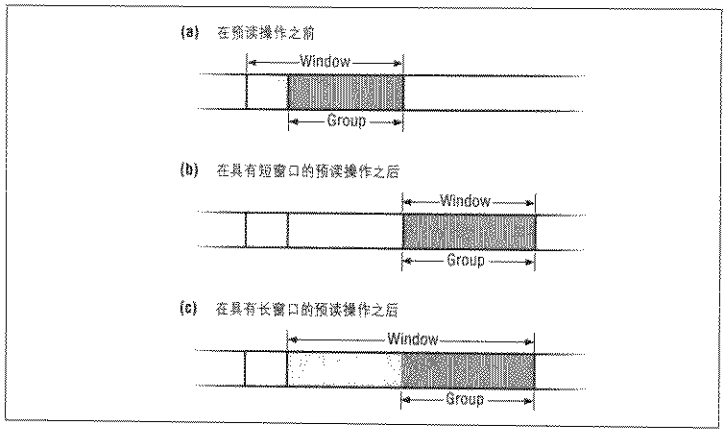


图 15-2 预读组和预读窗口

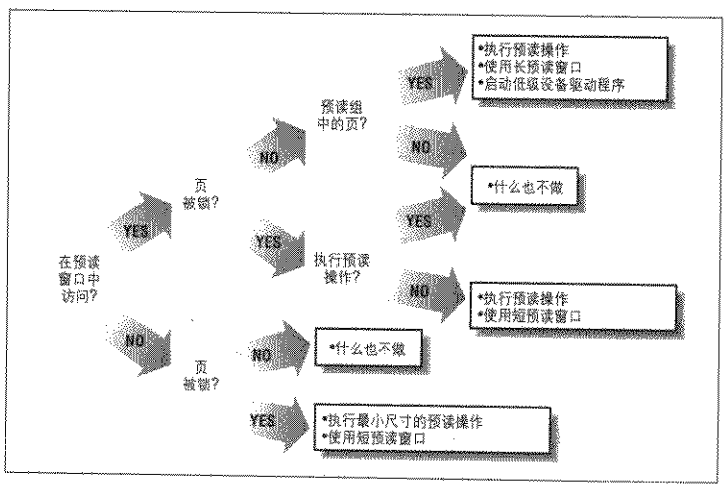


图 15-3 预读模式

假设第一页不在预读窗口中，这可能是因为进程从来没有访问过这个文件，也可能是因为进程发出了一个lseek()系统调用让当前的文件指针重新定位。内核会依次考虑进程请求的每个页。对于每个请求的页，内核都要调用generic_file_readahead()函数，该函数确定是否要执行预读操作。

通常可能发生两种情况（请参看图15-3）：或者该页被加锁了，也就是说对于该页的实际数据传送还没有完成；或者该页没有加锁，也就是该页是最新的。（为了简单起见，我们在本章中就不考虑在传送页的过程中所发生的I/O错误。）

如果该页没有被加锁，就不会发生预读操作。这条规则确保对数据全部包含在页高速缓存中的文件不执行预读操作。在这种情况下，任何预读处理都是对CPU时间的浪费。反之，如果该页被加锁了，那么内核就开始执行预读操作并为下一次预读操作准备一个短预读窗口。

顺序访问（在预读窗口之内）

现在假设进程使用read()系统调用所访问的第一页位于上一次预读操作的预读窗口之内。内核会依次考虑进程所请求的所有页，对于其中的每个页来说，内核都要调用generic_file_readahead()函数。根据该页是否被锁定以及该页是否包含在一个预读组中，可能会发生四种情况（请参看图15-3）：

- 如果该页没有被锁定，但是包含在一个预读组中，那么进程已经继续执行到它正在访问的页（这个页是最后一次预读操作从磁盘传送来的），这样进程在预读页结束的地方就有些危险了。在这种情况下，要执行另外的预读操作并使用一个长预读窗口，还要执行tq_disk任务队列中的函数来确保低级块设备驱动程序会被激活。
- 如果该页既没有被锁定，也不包含在一个预读组中，那么进程正在访问的页就是在倒数第二次预读操作中从磁盘上传送的页。在这种情况下，不会执行预读操作，因为进程与预读无关。这是最好的一种情况，说明内核有足够数量的预读页正由进程顺序地读取，完全如我们所愿。
- 如果该页既被锁定，也包含在一个预读组中，那么进程正在访问的页就是最后一次预读操作中所请求的页，但是该页可能还没有从磁盘上传送完。在这种情况下，开始执行另一个预读操作没有什么意义。

- 最后，如果该页被锁定，但是并不包含在一个预读组中，那么进程正在访问的页就是在倒数第二次预读操作中所请求的页。在这种情况下，内核要执行另外一个预读操作，因为该页可能会被写入磁盘，另外执行一个预读操作没有什么坏处。

写正规文件

回想一下write()系统调用会涉及把数据从调用进程的用户态地址空间中移动到内核数据结构中，然后再移动到磁盘上。文件对象的write方法允许每种文件类型都定义一个专用的写操作。在Linux 2.2中，每个基于磁盘的文件系统的write方法都是一个过程，该过程主要识别写操作所涉及的磁盘块，把数据从用户态地址空间拷贝到相应的缓冲区，然后把这些缓冲区标记成脏的。该过程和文件系统的类型有关，我们将在第十七章的“读写Ext2正规文件”一节中给出一个例子。

基于磁盘的文件系统不直接使用页高速缓存来写正规文件。这是从早期Linux版本中继承而来的，在早期的Linux版本中，唯一的磁盘高速缓存就是缓冲区高速缓存。但是，基于网络的文件系统通常都使用页高速缓存来写正规文件。

在Linux 2.2所使用的方法中，绕过了页高速缓存的问题，却产生了一个同步问题。当写操作发生时，有效数据是在缓冲区高速缓存中，而不是在页高速缓存中，更确切地说，当write方法修改了文件的任何部分时，与这些部分对应的页高速缓存中的所有页都不再包含有效数据。举一个例子，一个进程可能认为自己正在读取正确的数据，但是却未能看到其他进程对这些数据所做的修改。

为了解决这个问题，所有基于磁盘的文件系统的write方法都要调用update_vm_cache()函数来修改读操作所使用的页高速缓存。该函数使用以下参数：

inode
: 指向写操作所针对的文件的索引节点对象

pos
: 写操作发生时在文件内的偏移量

buf
: 从这个地址获得要写入文件的字符

count

要写入的字符数

通过执行以下操作，该函数逐页地更新与要写入的文件有关的页高速缓存部分：

1. 从pos中计算出要写入的第一个字符在一个页中的偏移量。
2. 调用find_page()在页高速缓存中查找文件偏移量pos处的字符所在的页框。
3. 如果找到了这个页，就执行以下操作：
   a. 调用wait_on_page()等待，直到该页被解锁为止（万一涉及一个I/O数据传送）。
   b. 从前面计算出的页偏移量pos处开始，用进程地址空间中的数据来填充该页。这是由文件系统的专用write方法早已写入缓冲区高速缓存中的数据。
   c. 调用free_page()来减少该页的引用计数器（该计数器是由find_page()函数增加的）。
4. 如果用户态地址空间中的一些数据还要被拷贝，就修改pos的值，把页偏移量设置成0，并返回步骤2。

现在我们要简要介绍一下通过页高速缓存对正规文件执行的写操作。回想一下这个操作只能用于网络文件系统。在这种情况下，文件对象的write方法是使用generic_file_write()函数实现的，该函数使用以下参数：

file

文件对象指针

buf

必须获取的要写入文件的字符的地址

count

要写入的字符个数

ppos

一个变量的地址，这个变量存放写入操作开始执行处的文件偏移量

该函数执行以下操作：

1. 如果file->flags的O_APPEND标志被置位，就把*ppos设置成文件的末尾，这样就可以把新数据追加到文件末尾。

2. 开始执行一个循环来更新这个写操作所涉及的所有页。在每次循环中，执行以下子步骤：

   a. 试图在页高速缓存中查找一个页。如果该页不在高速缓存中，就分配一个空闲页并将其加入页高速缓存。

   b. 调用wait_on_page()函数，然后把该页描述符的PG_locked标志置位，这样就可以获得对页内容的互斥访问。

   c. 调用 copy_from_user()，使用从进程地址空间中获得的数据填充该页。

   d. 调用索引节点操作的updatepage方法。这个方法是专用于正在使用的一种特殊网络文件系统的，在本书中没有介绍。它应该确保以新的所写数据适当地更新异地文件。

   e. 解除该页的锁定，调用wake_up()来唤醒在该页的等待队列中挂起的进程，并调用 free_page()来减少该页的引用计数器的值［该计数器是由find_page()来增加的］。

3. 修改*ppos的值，使它指向写入的最后一个字符之后的位置。

内存映射

正如我们在第七章的“线性区”一节中已经介绍过的一样，一个线性区可以和基于磁盘的文件系统中的一个文件（或者文件的一部分）相关联。这就是说，内核会把线性区中对一个页中字节的访问转换成对正规文件中相应字节的操作。这种技术称为内存映射（memory mapping）。

有两种类型的内存映射：

共享的

对线性区中页的任何写操作都会修改磁盘上的文件。而且，如果进程对共享内存映射中的一个页进行写，那么这种修改对于其他映射了相同文件的所有进程来说都是可见的。

私有的

当进程创建的映射只是为读文件，而不是写文件时才会使用。出于这种目的，私有映射的效率要比共享映射的效率更高。但是对私有映射页的任何写操作都会使内核不再映射该文件中的页。因此，一个写操作既不会修改磁盘上的文件，对访问相同文件的其他进程来说这种改变也是不可见的。

进程可以发出一个mmap()系统调用来创建一个新内存映射(请参看本章后面的“创建内存映射”一节)。程序员必须指定一个MAP_SHARED标志或MAP_PRIVATE标志作为这个系统调用的参数。正如你可以猜到的一样，前一种情况下，映射是共享的，而后一种情况下，映射是私有的。一旦创建了这种映射，进程就可以从这个新线性区的内存单元读取数据，也就等价于读取了文件中存放的数据。如果这个内存映射是共享的，那么进程可以通过对相同的内存单元进行写而达到修改相应文件的目的。为了撤消或者缩小一个内存映射，进程可以使用munmap()系统调用(请参看后面的“撤消内存映射”一节)。

作为一条通用规则，如果一个内存映射是共享的，相应的线性区就设置了VM_SHARED标志；如果一个内存映射是私有的，那么相应的线性区就清除了VM_SHARED标志。正如我们在后面会看到的一样，对于只读共享内存映射来说，有一个不符合本规则的特例。

内存映射的数据结构

以下数据结构结合起来来表示内存映射：

- 与所映射的文件相关的索引节点对象
- 一个文件对象，不同的进程对这个文件所执行的映射是不同的
- 对文件进行每一不同映射所使用的vm_area_struct描述符
- 对文件进行映射的线性区所分配的每个页框所对应的页指述符

图15-4说明了这些数据结构是如何链接在一起的。在左上角给出了索引节点。每个索引节点对象的i_mmap域指向一个双向链表的第一个元素，该链表中包含了当前对该文件进行映射的所有线性区。如果i_mmap为NULL，那么就没有任何线性区映射该文件。这个链表包含了表示线性区的vm_area_struct描述符，它是使用vm_next_share和vm_pprev_share域实现的。

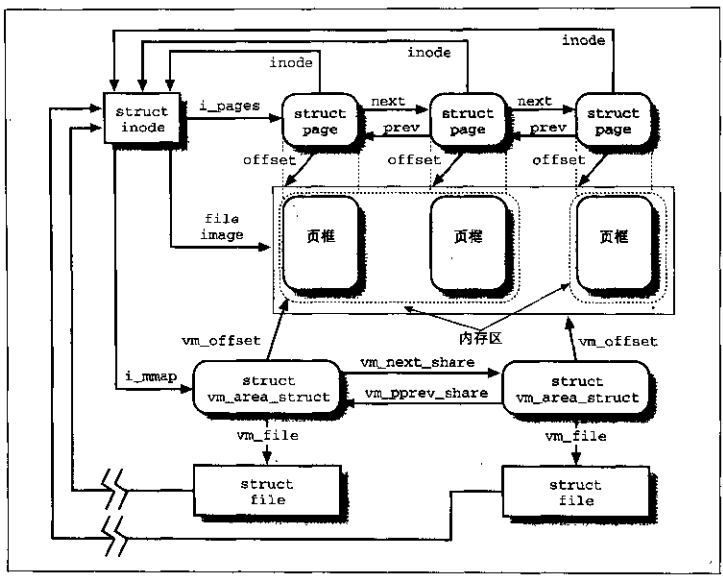


图 15-4 与内存映射有关的数据结构

每个线性区描述符的vm_file域都包含被映射文件的文件对象的地址。如果该域为空，那么在内存映射中就没有使用这个线性区。文件对象还包含几个域，这些域允许内核既标识拥有内存映射的进程，也标识被映射的文件。

第一个所映射的位置存放在线性区描述符的vm_offset域中。被映射的文件部分的长度就是这个线性区的大小，这样该值就可以从vm_start和vm_end域中计算出来。

共享内存映射的页通常都包含在页高速缓存中；私有内存映射的页只要还没有被修改，也都包含在页高速缓存中。当进程试图修改一个私有内存映射的页时，内核就把该页框进行复制，并在进程页表中用复制的页来替换原来的页框，这是第七章中介绍的写时复制机制的应用之一。虽然原来的页框还仍然在页高速缓存中，但不再属于这个内存映射，这是由于被复制的页框替换了原来的页框。依次类推，这个复制的页框不会被插入到页高速缓存中，因为其中所包含的数据不再是磁盘文件的有效数据。

交换高速缓存是由页高速缓存数据结构实现的，实现过程在第十四章中的“页高速缓存”一节中已经介绍过。回想一下页高速缓存中包含的页是与正规文件相关的，并且散列表允许算法快速从索引节点对象的地址和文件内部偏移量中导出页描述符的地址。交换高速缓存中的页在页高速缓存中以其他页的形式存放，并对其进行以下特殊的处理：

- 页描述符的inode域存放的是包含在swapper_inode变量中的虚拟索引节点对象地址。
- offset 域存放的是与该页相关的被换出页的标识符。
- 在 flags 域中的 PG_swap_cache 标志被设置。

还有，当该页被放入交换高速缓存时，该页描述符的count域和页插槽引用计数器都被增加，因为交换高速缓存既要使用页框，也要使用页插槽。

内核使用几个函数来处理交换高速缓存，这些函数主要是基于第十四章中的“页高速缓存”一节中的讨论。稍后我们将说明这些相对低层的函数是如何被高层函数调用来根据需要换入换出页的。

处理交换高速缓存的函数有：

in_swap_cache()

检查页的PG_swap_cache标志来确定该页是否属于交换高速缓存；如果属于，就返回存放在offset域中的被换出页的标识符。

lookup_swap_cache()

对作为参数传递的被换出页标识符进行操作并返回该页的地址，如果该页不在这个高速缓存中就返回0。该函数调用find_page()函数，把swapper_inode的虚索引节点对象和被换出页标识符的地址作为参数传递来查找所需要的页。如果该页在交换高速缓存中，lookup_swap_cache()就检测该页是否被锁定；如果被锁定，就调用wait_on_page()把当前进程挂起，直到该页被取消锁定为止。

add_to_swap_cache()

把页插入交换高速缓存中。该页描述符的inode和offset域分别被设置成swapper_inode虚索引节点对象的地址和这个被换出页标识符的地址。然后

通常，当一个方法的值为NULL时，内核就调用缺省的函数或者根本就不调用任何函数。如果nopage方法值为NULL，那么这个线性区就是匿名的，也就是说，它不会映射磁盘上的任何文件。这种用法在第七章中的“请求调页”一节中已经讨论过了。

表15-2 file_shared_mmap和file_private_mmap所使用的方法

方法	file_shared_mmap	file_private_mmap
open	NULL	NULL
close	NULL	NULL
unmap	filemap_unmap	NULL
protect	NULL	NULL
sync	filemap_sync	NULL
advise	NULL	NULL
nopage	filemap_nopage	filemap_nopage
wppage	NULL	NULL
swapout	filemap_swapout	NULL
swapin	NULL	NULL

创建内存映射

要创建一个新的内存映射，进程就要发出一个mmap()系统调用，并向该函数传递以下参数：

- 文件描述符，标识要映射的文件。
- 文件内的偏移量，指定要映射的文件部分的第一个字符。
- 要映射的文件部分的长度。
- 一组标志。进程必须显式地设置MAP_SHARED标志或MAP_PRIVATE标志来指定所请求的内存映射的种类（注1）。

注1：进程可以设置MAP_ANONYMOUS标志来指定这个新线性区是匿名的，也就是说，和任何文件都无关（请参看第七章中的“请求调页”一节）。但是，这个标志是一个Linux扩展，而不是由POSIX标准定义的。

- 一组权限，指定一种或者多种对线性区进行访问的权限：读访问（PROT_READ）、写访问（PROT_WRITE）或执行访问（PROT_EXEC）。
- 一个可选的线性地址，内核把该地址作为一个线索，说明新线性区应该从哪里开始。如果指定了MAP_FIXED标志，且内核不能从指定的线性地址开始分配这个新线性区，那么这个系统调用失败。

mmap()系统调用返回新线性区中第一个位置的线性地址。其服务例程是使用old_mmap()函数实现的，该函数实际上调用在第七章的“分配线性地址区间”一节中介绍过的do_mmap()函数。我们现在就详细介绍当创建对文件进行映射的线性区时才会执行的步骤。

1. 检查是否为要映射的文件定义了mmap文件操作。如果没有，就返回一个错误代码。文件操作表中的mmap值为NULL说明相应的文件不能被映射（例如，因为这是一个目录）。
2. 除了进行正常的一致性检查之外，还要对所请求的内存映射的种类与在打开文件时所指定的标志进行比较。mmap()系统调用作为参数传递的这些标志指定需要映射的种类，而文件对象的f_mode域的值说明这个文件的打开方式。根据这两个消息源，执行以下的检查：
   a. 如果一个共享可写的内存映射被请求，就检查那个文件是为写入而打开的，而不是以追加模式打开的（open()系统调用使用O_APPEND标志）。
   b. 如果一个共享内存映射被请求，就检查那个文件上没有强制锁（请参看第十二章中的“文件加锁”一节）。
   c. 对于任何种类的内存映射，都要检查文件是为读操作而打开的。

   如果以上这些条件都不能满足，就返回一个错误代码。
3. 当对这个新线性区描述符的vm_flags域的值进行初始化时，要根据访问权限来设置该文件的VM_READ、VM_WRITE、VM_EXEC、VM_SHARED、VM_MAYREAD、VM_MAYWRITE、VM_MAYEXEC和VM_MAYSHARE标志以及所请求的内存映射的种类（请参看第七章中的“线性区存取权限”一节）。最佳的情况是，对于非写入共享内存映射，VM_SHARED标志被清0。可以这样处理是因为不允许进程写入这个线性区的页，因此，这种映射的处理就与私有映射的处理相同。但是，内核实际上允许共享该文件的其他进程访问这个线性区中的页。

4. 为正被映射的文件调用mmap方法，给其传递的参数为这个文件对象的地址和线性区描述符的地址。对于大部分文件系统来说，这个方法是使用generic_file_mmap()函数实现的，该函数执行以下操作：

   a. 初始化线性区描述符的vm_ops域。如果VM_SHARED标志被置位，就把该域设置成file_shared_mmap，否则就把该域设置成file_private_mmap（请参看表15-2）。从某种意义上说，这个步骤所做的事情类似于打开一个文件并初始化文件对象的方法。

   b. 从索引节点的i_mode域检查要映射的文件是否是一个正规文件。如果是其他类型的文件（例如目录或套接字），就返回一个错误代码。

   c. 从索引节点的i_op域中检查是否定义了readpage()的索引节点操作。如果没有定义，就返回一个错误代码。

   d. 调用update_atime()函数把当前时间存放在该文件索引节点的i_atime域中，并将这个索引节点标记成脏。

5. 用文件对象的地址初始化这个线性区描述符的vm_file域，并增加该文件的引用计数器的值。

6. 回想一下第七章中的“分配线性地址区间”一节中的内容，do_mmap()会调用insert_vm_struct()函数。该函数在执行过程中，把这个线性区描述符插入到索引节点的i_mmap域所指向的链表。

撤消内存映射

当进程准备撤消一个内存映射时，就调用munmap()系统调用，要给它传递以下参数：

- 要删除的线性地址区间中第一个位置的地址
- 要删除的线性地址区间的长度

注意munmap()系统调用可以用来删除或者减少任何种类的线性区的大小。实际上，这个系统调用的服务例程sys_munmap()本质上要调用在第七章中的“释放线性地址区间”一节中已经介绍过的do_munmap()函数。但是，如果这个线性区映射了一个文件，那么对要释放的线性地址区间所在的每个线性区都执行以下额外的步骤：

1. 检查这个线性区是否有unmap方法；如果有，就调用这个方法。通常，私有内存映射没有这种方法，因为文件没有被修改。共享内存映射使用fileap_unmap()函数，该函数又依次调用filemap_sync()函数（请参看后面的“把内存映射的脏页刷新到磁盘”一节）。
2. 调用remove_shared_vm_struct()函数把这个线性区描述符从i_mmap域所指向的索引节点链表中删除。

对内存映射进行请求调页

出于效率的原因，内存映射被创建之后并没有立即把页框分配给它，而是尽可能向后推迟到不能再推迟——也就是说，当进程试图对其中的一页进行寻址时，就产生一个“缺页”异常。

我们在第七章中的“缺页异常处理程序”一节中已经看到，内核是如何验证所缺少的地址是否包含在某个进程的线性区中。如果在，那么内核就检查所缺的地址所对应的页表项，如果该项为空就调用do_no_page()函数（请参看第七章中的“请求调页”一节）。

do_no_page()函数执行对请求调页的所有类型都通用的操作，例如分配页框和修改页表。它还检查这个线性区是否定义了nopage方法。在第七章中的“请求调页”一节中，我们已经介绍了这个方法没有定义的情况（匿名线性区）。现在我们讨论当这个方法被定义时，以该函数所执行的操作来完成对这个函数的介绍：

1. 调用nopage方法，它返回所请求页所在的页框地址。
2. 增加该进程内存描述符的rss域来表示已经给该进程分配了一个新页。
3. 用nopage方法所返回的页框地址以及这个线性区的vm_page_prot域中所包含的页访问权，来建立缺少的地址所对应的页表表项。进一步的操作依赖于访问的类型：
   - 如果进程正在试图写入页，就强制把这个页表项的Read/Write和Dirty位设置成1。在这种情况下，或者这个页框被专门分配给这个进程，或者这个页框是共享的，在这两种情况下，都应该允许写入。（这就避免了由写时复制机制第二次产生无用的“缺页”异常。）

— 如果这个进程正试图从该页中读取数据，线性区的VM_SHARED标志也没有被设置，而且该页的引用计数器大于1，那么就强制把该页表项的Read/Write设置成0。这是因为该页的引用计数器说明其他进程正在共享该页；因为该页不属于共享线性区，因此就必须通过写时复制机制来处理。

请求调页算法的核心是线性区的nopage方法。一般来说，该方法必须返回进程所访问页所在的页框地址。其实现依赖于页所在线性区的种类。

在处理对磁盘文件进行映射的线性区时，nopage方法必须首先在页高速缓存中查找所请求的页。如果没有找到相应的页，这个方法就必须将其从磁盘上读入。大部分文件系统都是使用filemap_nopage()函数来实现nopage方法的，该函数使用三个参数：

area
所请求页所在线性区的描述符地址。

address
所请求页的线性地址。

no_share
一个标志，指定该函数所返回的页框不一定在很多进程之间共享。只有进程试图写入该页且VM_SHARED标志没有置位时，do_no_page()函数才会设置这个标志。

filemap_nopage()函数执行以下步骤：

1. 从area->vm_file域中获得文件对象的地址；从这个文件对象的f_dentry域所指向的目录项对象的d_inode域中，获得索引节点对象的地址。
2. 用area的vm_start和vm_offset域来确定，从address开始的页对应的数据在文件中的偏移量。
3. 检查这个文件偏移量是否大于文件大小。如果是，而且VM_SHARED标志置位，就返回0，这样就会引起向进程发送一个SIGBUS信号。（私用内存映射的行为有些不同，它给进程分配一个用0填充的新页框。）
4. 调用find_page()在页高速缓存中查找由索引节点对象和文件偏移量所指定的页。如果该页不在高速缓存中，就调用try_to_read_ahead()来分配一个新

页框，将其加入页高速缓存，并用从磁盘上读取的数据来填充该页的内容。实际上，内核还会试图预读下一个page_cluster页（请参看第十六章）。

5. 调用wait_on_page()一直等待，直到所请求的页变为解锁为止（也就是说，直到页的当前I/O数据传送完成为止）。

6. 如果no_share标志为0，那么该页框就可以共享：返回该页框的地址。

7. 如果no_share标志为1，那么该进程就试图把数据写入一个私有内存映射页（或者更确切地说，把数据写入一个VM_SHARED标志没有置位的线性区）。这样，就执行以下操作分配一个新页：

   a. 调用__get_free_page()函数

   b. 把页高速缓存中所包含的该页内容拷贝到这个新页框中

   c. 减少页高速缓存中该页的引用计数器，从而取消find_page()对这个引用计数器所做的增加

   d. 返回新页框的地址

把内存映射的脏页刷新到磁盘

进程可以使用msync()系统调用把属于共享内存映射的脏页刷新到磁盘。这个系统调用所接收的参数为：一个线性地址区间的起始地址、区间的长度以及具有下列含义的一组标志。

MS_SYNC

要求这个系统调用挂起进程，直到这个I/O操作完成为止。在这种方式中，调用进程就可以假设，当系统调用完成时，这个内存映射中的所有页都已经被刷新到磁盘。

MS_ASYNC

要求系统调用立即返回，而不用挂起调用进程。

MS_INVALIDATE

要求系统调用从进程地址空间删除内存映射中的所有页。

对这个线性地址区间中所包含的每个线性区，sys_msync()服务例程都调用msync_

interval()(请参看第七章中的“分配线性地址区间”一节)。后者依次执行以下操作:

1. 如果线性区描述符的vm_file域为NULL,就返回0(说明这个线性区没有映射一个文件)。
2. 调用线性区操作的sync方法。在大部分文件系统中,这个方法是使用filemap_sync()函数实现的(很快就会介绍)。
3. 如果MS_SYNC标志被置位,就调用file_fsync()函数把所有相关文件信息都刷新到磁盘上,包括该文件的索引节点、该文件系统的超级块以及该文件的所有脏缓冲区[通过sync_buffers()]。

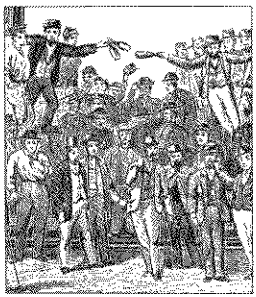
filemap_sync()函数把这个线性区中所包含的数据都拷贝到磁盘上。它首先扫描与这个线性地址区间对应的页表表项。对于每个找到的页框,都要执行以下步骤:

1. 调用flush_tlb_page()来刷新转换后援缓冲器(TLB)。
2. 如果MS_INVALIDATE标志没有设置,就增加这个页描述符的引用计数器。
3. 如果MS_INVALIDATE标志被设置,就把相应的页表项设置成0,从而说明该页不再存在了。
4. 调用filemap_write_page()函数,该函数顺次执行以下子步骤:
   a. 增加与该文件相关联的文件对象的引用计数器。这是一种故障保险(fail-safe)机制,因此如果在I/O数据传送仍在进行时该进程就结束了,那么这个文件对象也不会被释放。
   b. 调用do_write_page(),该函数实际上会执行该文件操作的write方法,从而对文件模拟执行一个write()系统调用。当然,在这种情况下,要写入磁盘的数据不是从用户态缓冲区中取得的,而是从正在被刷新的页中取得的。
   c. 调用fput()来减少该文件对象的引用计数器。从而抵消了步骤4a中对该计数器所做的增加。
5. 调用free_page()来减少该页描述符的引用计数器。如果MS_INVALIDATE标志没有置位,该步操作就抵消了步骤2中对该计数器所做的增加;否则,如果

MS_INVALIDATE标志被置位，filemap_sync()的全局作用是释放该页框（如果这个计数器变成0，就把该页框归还给伙伴系统）。

对Linux 2.4的展望

以上这些方法基本上保持不变。然而，如果一个线性区被认为是“连续读取”，那么在从磁盘上读出页时就要执行预读操作。还有，正如我们在第十三章最后所介绍的一样，在Linux 2.4中写正规文件要简单得多，因为写操作可以简单地通过页I/O操作来实现。



第十六章

交换：释放内存的方法

上一章所介绍的磁盘缓存把RAM用作磁盘的一个扩充，目的是提高系统的响应时间，这种解决方法减少了磁盘访问的次数。在本章我们引入一种称为交换（swapping）的相反的方法：这里内核把磁盘上的一些空间用作RAM的一个扩展。交换对于程序员来说是透明的，只要交换空间正确地被安装并激活，进程就可以假定自己已经拥有了所有可以寻址的可用物理内存，而不用知道那些页存放在何处，在需要时又从何处去检索。

磁盘高速缓存以占用空闲RAM为代价来提高系统的性能，而交换则以损失访问速度为代价来扩大可寻址内存的数量。因此，磁盘高速缓存是一种比较好而理想的方法，而交换应该被认为是在空闲RAM变得严重不足时万不得已采取的一种措施。

本章开始我们首先在“什么是交换？”一节中对交换进行定义。然后我们在“交换区”一节中介绍Linux用来实现交换所引入的主要数据结构。我们在本章要讨论交换高速缓存及在RAM和交换区之间传送页的低级函数。本章中最关键的两节是“页换出”和“页换入”，在“页换出”一节中我们介绍选择一个页交换到磁盘所使用的过程，在“页换入”一节中我们介绍交换区中所存放的页是如何在需要时读回到RAM中的。

本章有力地论证了内存管理的理论。最后还有一个主题需要讨论，也就是页框的回收，这将在最后一节进行介绍，它只是与交换的部分内容相关。由于使用如此之多的磁盘高速缓存（包括交换高速缓存），最终所有可用的RAM都可能消耗在这些高速缓存中而没有多少空闲的RAM所剩余。我们将看到内核如何防止这种情况的发

生，这是通过一方面监控空闲RAM的数量，另一方面按需从高速缓存或进程的地址空间中释放RAM来达到的。

什么是交换？

交换主要用于两个目的：

- 扩展进程可以有效使用的地址空间
- 扩充动态RAM的数量（也就是内核代码和静态数据结构被初始化之后剩余的RAM）来装载进程

让我们给出几个例子来说明用户是如何得益于交换的。最简单的一个例子是程序的数据结构占用的空间超过了可用RAM的大小，此时交换区会允许毫无问题地装载这个程序。一个更恰当的例子是用户执行几个命令试图同时运行几个需要很多内存的大型应用程序。如果此时没有激活交换区，那么系统就可能拒绝开始执行新的应用程序；如果此时激活了交换区，那么交换区就允许内核来启动新的应用程序，因为不用杀死现有的进程就可以把现有进程的部分内存释放出来。

这两个例子都不仅说明了交换的优点，而且也说明了交换的缺点。从性能方面来说，模拟的RAM毕竟不如RAM。进程对目前已被换出页的每次访问都把进程的执行时间增加几个数量级。简而言之，如果我们最大程度地追求性能，那么交换就应该是我们万不得已时的一种选择方案；增加RAM芯片仍就是处理日益增加的计算需求的最好方法。但是公正地说，在某些情况下，交换对整个系统来说还是有益的。长时间运行的进程通常只会访问所获得页框的一半。即使当还有一些RAM可用时，把未用的页交换出去并使用RAM作为磁盘高速缓存也可以提高整个系统的性能。

交换技术已经使用了很多年。第一个Unix系统内核就监控空闲内存的数量。当空闲内存数量小于一个固定的极限值时，就执行换出操作。换出操作包括把进程的整个地址空间拷贝到磁盘上。反之，当调度算法选择一个换出进程时，整个进程又被从磁盘中交换进来。

现代的Unix（包括Linux）内核已经摒弃了这种方法，主要是因为当进行换入换出时，上下文切换的代价相当高。为了补偿这种交换操作的负担，调度算法必须相当复杂，它必须在考虑换出进程时优先考虑RAM中的进程。

在Linux中，交换目前是在页级别而不是在进程地址空间级别上执行的。正是由于CPU的硬件分页单元，页级交换已经得以实施。我们回想一下第二章中的“常规分页”一节的内容，每个页表项都包含一个Present标志，内核可以利用这个标志发出信号通知硬件，进程地址空间中的一个页已经被交换出去了。除了这个标志之外，Linux还利用页表项中的其余位来存放要交换到磁盘上的页的位置。当发生“缺页”异常时，相应的异常处理程序就可以检测到这个页不在RAM中，并调用函数把所缺的页从磁盘中交换进来。

算法的复杂性大部分是与换出有关。具体说来，必须考虑四个主要问题：

- 哪种页要换出
- 如何在交换区中分布各个页
- 如何选择被交换出的页
- 何时执行页换出操作

在介绍与交换有关的数据结构和函数之前，让我们先来简要地看一下Linux是如何处理这四个主要问题的。

哪种页要换出

交换仅仅适用于以下类型的页：

- 属于进程的匿名线性区（例如，用户态堆栈）的页
- 对属于进程的私有内存映射修改过的页
- 属于IPC共享线性区的页（请参看第十八章中的“IPC共享内存”一节）

其他类型的页可以供内核使用，或者用来映射磁盘上的文件。在第一种情况下，交换会忽略这些页，因为这样可以简化内核的设计；在第二种情况下，这些页使用的最好的交换区是文件本身。

如何在交换区中分布页

每个交换区都被组织成插槽（slot），每个插槽都正好包含一页。当换出时，内核尽

力把换出的页存放在相邻的插槽中，从而减少在访问交换区时磁盘的寻道时间，这是一个高效交换算法的一个重要因素。

如果系统使用了多个交换区，事情就变得更加复杂了。快速交换区（也就是存放在快速磁盘中的交换区）可以获得比较高的优先级。当查找一个空闲插槽时，要从优先级最高的交换区中开始搜索。如果优先级最高的交换区不止一个，为了避免超负荷地使用其中一个，应该循环选择相同优先级的交换区。如果在优先级最高的交换区中没有找到空闲插槽，就在优先级次高的交换区中继续进行搜索，依此类推。

如何选择要换出的页

如果缺页异常属于IPC共享内存（我们将在第十八章中讨论），换出的一般规则是从RAM中页数最多的进程中挪用几页。但是对这个进程中的哪些页应当换出必须做出选择，根据某一标准对这些页进行排列是一种好的方法。在某些内核中已经提出并采用了几种最近最少使用（LRU）的置换算法。其主要思想是给RAM中的每个页设置一个计数器来存放该页的年龄，也就是说，从最后一次访问该页以来所经过的时间。这样就可以把该进程的最老页交换出去。

有些计算机平台为LRU算法提供了相当好的支持。例如，有些大型主机的CPU可以自动更新在每个页表项中包含的计数器，这个计数器就是用来定义相应页的年龄的。但是Intel 80x86处理器并没有提供这种硬件特性，因此Linux不能使用真正的LRU算法。然而，在选择要换出的页时，Linux利用了每个页中包含的`Accessed`标志。正如我们会在后面看到的一样，这个标志的设置和清除是以相当简单的方法进行的，以此来防止过渡频繁地换入换出页。

何时执行页换出操作

当内核由于内存紧缺而变得不稳定时换出是十分有用的。实际上，内核保留了少量的空闲页框供最关键的函数使用。实验证明这对于防止系统崩溃有关键作用，在对空闲资源进行操作的内核例程不能获得自己完成任务所需要的内存区时就会发生系统崩溃。为了保护这种空闲页框的保留机制，Linux要在以下情况发生时执行换出操作：

- 当空闲页框少于预定义的极限时，每秒激活一次名为*kswapd*的内核线程

- 由于空闲页框低于预定义的极限，当对伙伴系统（请参看第六章中的“伙伴系统算法”一节）的内存请求不能满足时

有几个函数和交换有关，图16-1说明了其中最重要的一些函数，在后面的章节中我们会陆续对这些函数进行讨论。

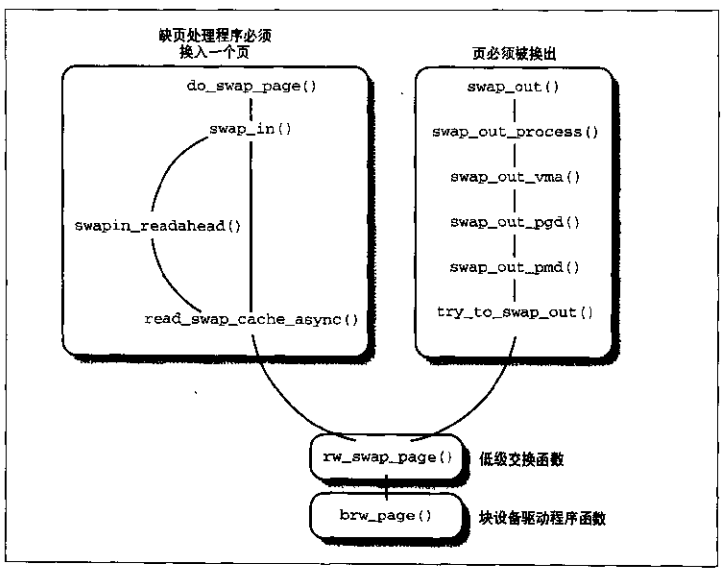


图16-1 与交换有关的主要函数

交换区

从内存中换出的页被存放在交换区（swap area）中，交换区可以使用自己的磁盘分区进行实现，也可以使用大型分区中所包含的文件来实现。可以定义几种不同的交换区，最大个数由MAX_SWAPFILES宏（通常被设置成8）确定。

交换高速缓存是由页高速缓存数据结构实现的，实现过程在第十四章中的“页高速缓存”一节中已经介绍过。回想一下页高速缓存中包含的页是与正规文件相关的，并且散列表允许算法快速从索引节点对象的地址和文件内部偏移量中导出页描述符的地址。交换高速缓存中的页在页高速缓存中以其他页的形式存放，并对其进行以下特殊的处理：

- 页描述符的inode域存放的是包含在swapper_inode变量中的虚拟索引节点对象地址。
- offset 域存放的是与该页相关的被换出页的标识符。
- 在 flags 域中的 PG_swap_cache 标志被设置。

还有，当该页被放入交换高速缓存时，该页描述符的count域和页插槽引用计数器都被增加，因为交换高速缓存既要使用页框，也要使用页插槽。

内核使用几个函数来处理交换高速缓存，这些函数主要是基于第十四章中的“页高速缓存”一节中的讨论。稍后我们将说明这些相对低层的函数是如何被高层函数调用来根据需要换入换出页的。

处理交换高速缓存的函数有：

in_swap_cache()

检查页的PG_swap_cache标志来确定该页是否属于交换高速缓存；如果属于，就返回存放在offset域中的被换出页的标识符。

lookup_swap_cache()

对作为参数传递的被换出页标识符进行操作并返回该页的地址，如果该页不在这个高速缓存中就返回0。该函数调用find_page()函数，把swapper_inode的虚索引节点对象和被换出页标识符的地址作为参数传递来查找所需要的页。如果该页在交换高速缓存中，lookup_swap_cache()就检测该页是否被锁定；如果被锁定，就调用wait_on_page()把当前进程挂起，直到该页被取消锁定为止。

add_to_swap_cache()

把页插入交换高速缓存中。该页描述符的inode和offset域分别被设置成swapper_inode虚索引节点对象的地址和这个被换出页标识符的地址。然后

被激活，也可以在系统运行之后动态激活。在交换区可以有效地表示成系统RAM的扩展时，这个初始化过的交换区就被认为是激活的了（请参看本章后面的“激活和使交换区无效”一节）。

交换区描述符

每个活动的交换区在内存中都有自己的swap_infc_struct描述符，其域如表16-1所示。

表16-1 交换区描述符的域

类型	域	说明
unsigned int	flags	交换区标志
kdev_t	swap_device	交换设备的设备号
struct dentry *	swap_file	文件或设备文件的目录项
unsigned short *	swap_map	指向计数器数组的指针，每个交换区页插槽对应一个数组元素
unsigned char *	swap_lockmap	指向位锁数组的指针，每个交换区页插槽对应一个数组元素
unsigned int	lowest_bit	在搜索一个空闲页插槽时要扫描的第一个页插槽
unsigned int	highest_bit	在搜索一个空闲页插槽时要扫描的最后一个页插槽
unsigned int	cluster_next	在搜索一个空闲页插槽时要扫描的下一个页插槽
unsigned int	cluster_nr	在重新从头开始执行之前空闲页插槽分配的个数
int	prio	交换区优先级
int	pages	可用页插槽的个数
unsigned long	max	交换区的大小，以页为单位
int	next	指向下一个交换区描述符的指针

flags域包括两个重叠的子域：

SWP_USED

如果交换区是活动的，该值就是1；如果交换区不是活动的，该值就是0。

SWP_WRITEOK

如果不能写入交换区，这个2位的域就被设置成3，否则被设置成0。由于该域的最低一位就是用来实现SWP_USED的位，所以只有交换区是活动的才可以被写入。当交换区正被激活或无效时，内核不能写入这个交换区。

swap_map域指向一个计数器数组，每个交换区页插槽对应一个元素。如果计数器值等于0，那么这个页插槽就是空闲的；如果计数器为正数，那么这个页插槽就被填充成一个换出页的内容（正数值的确切意义会在“交换高速缓存”一节中进行讨论）。如果计数器的值为SWAP_MAP_MAX（等于32767），那么存放在这个页插槽中的页就是“持久”的，并且不能从相应的插槽中删除。如果计数器的值是SWAP_MAP_BAD（等于32768），那么这个页插槽就被认为是有错的，也就是不可用的。

swap_lockmap域指向一个位数组，每个交换区页插槽对应一个数组元素。如果一个位被置位，就说明存放在页插槽中的页现在正在被换入或换出。这样该位就作为一个锁来使用，用来确保在一次I/O数据传送过程中对这个页插槽进行互斥访问。

prio域是一个有符号的整数，专门用来说明交换区的“良好程度”。在快速磁盘中实现的交换区应该有较高的优先级，以便将被首先使用。只有在高优先级的交换区全部被填充时交换算法才会考虑优先级更低的交换区。优先级相同的交换区被循环选中从而把换出页分布到各个交换区之间。正如我们在“激活和使交换区无效”一节中会看到的一样，优先级是在交换区被激活时分配的。

swap_info数组包括MAX_SWAPFILES个交换区描述符。当然，并不是所有这些交换区描述符都必须使用，只有那些设置了SWP_USED标志的交换区描述符才是必需的。图16-2说明了swap_info数组、一个交换区和相应的计数器数组的情况。

nr_swapfiles变量存放的是最后一个数组元素的索引，这个元素中包含或者已经包含了一个有效使用的交换区描述符。这个变量有些名不符实，它并没有包括活动交换区的个数。

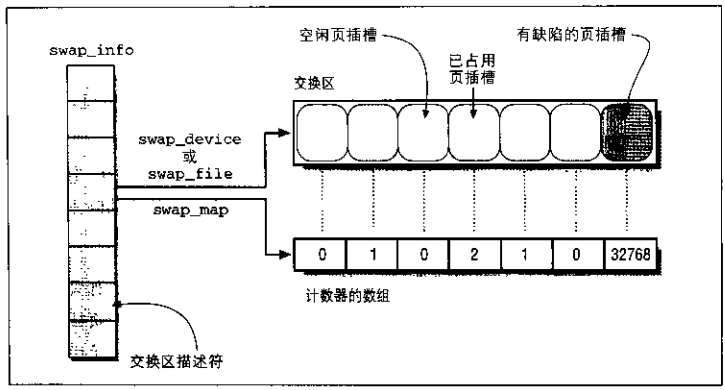


图 16-2 交换区数据结构

活动交换区描述符也被插入一个按照交换区优先级排序的列表中。该列表是通过交换区描述符的 next 域实现的，其中存放了 swap_info 数组中的下一个描述符。该域作为索引的这种用法与我们已经见过的名为 next 的域的用法有所不同，后者通常都是指针。

swap_list_t 类型的 swap_list 变量包括以下域：

head

第一个链表元素在 swap_info 数组中的索引。

next

为换出的页所选择的下一个交换区的描述符在swap_info数组中的索引。用该域在具有空闲插槽的优先级最大的交换区之间实现轮询算法。

max域存放以页为单位交换区的大小，而pages域存放可用页插槽的数目。这两个数字之所以不同是因为 pages 域并没有考虑第一个页插槽和有缺陷的页插槽。

最后，nr_swap_pages 变量包含所有活动交换区中空闲、无缺陷的页插槽的数目。

换出页标识符

可以很简单地而又唯一地标识一个换出页，这是通过在swap_info数组中指定交换区的索引和在交换区内指定页插槽的索引实现的。由于交换区的第一个页（索引为0）留给swap_header联合体（我们在前面已经讨论过），第一个可用页插槽的索引就为1。换出页标识符的格式如图16-3所示。

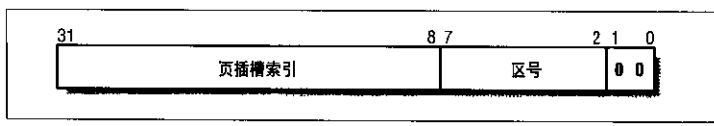


图16-3 换出页标识符

SWP_ENTRY(type, offset)宏负责从交换区索引type和页插槽索引offset中构造换出页标识符。SWP_TYPE和SWP_OFFSET宏正好相反，它们分别从换出页标识符中提取出交换区索引和页插槽索引。

当一个页被换出时，其标识符就被作为该页的表项插入页表中，这样在需要时就可以找到这个页了。要注意这种标识符的最低位与Present标志对应，通常被清除来说明该页目前不在RAM中这一事实。但是，最高30位中至少有一位被置位，因为插槽0中没有存放任何页。这样就可以从一个页表项的值中区分三种不同的情况：

- 空项：该页不属于进程的地址空间。
- 前30个最高位不全等于0，最后2位等于0：该页现在被换出了。
- 最低位等于1：该页包含在RAM中。

由于一个页可以属于多个进程的地址空间（请参看后面的“交换高速缓存”一节），所以它可能从几个进程的地址空间中被换出，但是仍旧保留在主存中；因此可以把同一个页换出多次，当然，一个页在物理上只被换出并存储一次，但是后来每次请求换出该页都会增加swap_map计数器的值。

在试图换出一个已经换出的页时就会调用swap_duplicate()函数。该函数只是验证以参数传递的换出页标识符是否有效，并增加相应的swap_map计数器的值。更确切地说，该函数执行以下操作：

1. 使用SWP_TYPE和SWP_OFFSET宏从参数中提取出分区号type和页插槽索引offset。
2. 检查是否发生了以下错误条件之一：
   a. type大于nr_swapfiles。
   b. swap_info[type].flags中的SWP_USED标记被清除了，说明这个交换区不是活动的。
   c. offset大于swap_info[type].max。
   d. swap_info[type].swap_map[offset]为0，说明这个页插槽是空闲的。

   如果这些条件有一个发生，就返回0（无效的标识符）。
3. 如果前面的测试通过了，换出页标识符就定位在一个有效页上。因此，增加swap_info[type].swap_map[offset]的值；但是，如果计数器等于SWAP_MAP_MAX或SWAP_MAP_BAD，就不要修改这个值了。
4. 返回1（有效的标识符）。

激活和使交换区无效

一旦交换区被初始化，超级用户（或者更确切地说是任何具有CAP_SYS_ADMIN权限的用户，有关内容将在第十九章中的“进程的信任状和能力”一节中介绍）就可以分别使用*/sbin/swapon*和*/bin/swapoff*程序激活和使交换区无效。这两个程序分别使用了swapon()和swapoff()系统调用，我们将简要介绍相应的服务例程。

sys_swapon()服务例程接收如下参数：

specialfile
: 这个参数包括设备文件（或分区）路径名，或用来实现交换区的普通文件的路径名。

swap_flags
: 如果SWAP_FLAG_PREFER位被置位，那么最低15位就指定这个交换区的优先级。

sys_swapon()函数对在创建交换区时放入第一个插槽中的swap_header联合域进行检查。其执行的主要步骤有：

1. 检查当前进程是否具有CAP_SYS_ADMIN能力。
2. 在swap_info中查找第一个SWP_USED标志被清除的标识符，也就是说对应的交换区不是活动的。如果没有找到匹配的项，就返回一个错误代码，因为此时已经有MAX_SWAPFILES个活动交换区了。
3. 查找到了交换区描述符；设置SWP_USED标志。如果这个描述符的索引大于nr_swapfiles，还要更新这个变量。
4. 设置这个描述符的prio域。如果swap_flags参数没有指定优先级，就在所有活动交换区中把优先级最低的域减少1后赋给这个域（这样就假设最后一个被激活的交换区在最慢的块设备上）。如果没有其他交换区是活动的，就把该域设置成-1。
5. 把描述符的swap_file域初始化成与specialfile相关的目录项对象的地址，并由namei()函数返回（请参看第十二章中的“路径名的查找”一节）。
6. 如果specialfile参数指定了一个块设备文件，就把设备号存放在这个描述符的swap_device域中，把blksize_size数组中适当的项的块大小设置成PAGE_SIZE，并调用blkdev_open()函数。后面这个函数设置一个与这个设备文件有关的新文件对象的f_ops域，并初始化这个硬件设备（请参看第十三章中的“VFS对设备文件的处理”一节）。还要通过在swap_info查找其他描述符的swap_device域来检查这个交换区还没有被激活。如果已经被激活了，就返回一个错误代码。
7. 另一方面，如果specialfile参数指定的是一个正规文件，就通过在swap_info中查找其他描述符的swap_file->d_inode域来检查这个交换区还没有被激活。如果已经被激活，就返回一个错误码。
8. 分配一个页框并调用rw_swap_page_nocache()函数（请参看本章后面的“传送交换页”一节）来读取交换区的第一页，该页中存放着swap_header联合体。
9. 检查第一页的最后10个字符中的magic字符串等于SWAP-SPACE或SWAPSPACE2（有两个稍微不同版本的交换算法）。如果不是，specialfile参数就没有指定一个已经初始化过的交换区，因此就返回一个错误代码。

10. 根据存放在swap_header联合体中的info.last_page域中的交换区的大小初始化这个交换区描述符的lowest_bit、highest_bit和max域。

11. 调用vmalloc()来创建和这个新交换区有关的计数器数组，并把它的地址存放在这个交换描述符的swap_map域中。还要根据swap_header联合体的info.bad_pages域中存放的有缺陷的插槽链表把这个数组的元素初始化成0或SWAP_MAP_BAD。

12. 再次调用vmalloc()函数来创建锁数组，并将其地址存放在交换描述符的swap_lockmap域中；把所有的位都初始化成0。

13. 通过访问第一个页插槽中的info.last_page和info.nr_badpages域计算可用页插槽的个数。

14. 把交换描述符的flags域设置成SWP_WRITEOK，把pages域设置成可用页插槽的个数，并更新nr_swap_pages变量。

15. 把这个新交换区描述符插入swap_list变量所指向的链表中。

16. 释放交换区的第一页数据所在的页框，并返回0（成功）。

sys_swapoff()服务例程使specialfile参数所指定的交换区无效。sys_swapoff()比sys_swapon()复杂得多，也更加耗时，因为要使之无效的这个分区现在可能仍然还包含几个进程的页。因此，强制该函数扫描交换区并把所有现有的页都换入，由于每个换入操作都需要一个新的页框，因此如果现在没有空闲页框，这个操作就可能失败。在这种情况下，该函数就返回一个错误码。所有这些操作都是通过执行以下步骤实现的：

1. 对specialfile调用namei()函数来获得一个与这个交换区有关的目录项对象的指针。

2. 扫描swap_list所指向的链表，并定位一个描述符，这个描述符的swap_file域指向与specialfile相关的目录项对象。如果不存在这样的描述符，就给该函数传递一个无效参数；如果存在这样的描述符，但是在设置SWP_USED时清除了它的SWP_WRTIEOK标志的最重要的位，那么这个交换区就正在被禁用的过程中。这两种情况只要出现一种，就返回一个错误码。

3. 把这个描述符从链表中删除，并将其flags域设置成SWP_USED，这样在该函数使这个交换区无效之前，内核就不会再把其他页存放到这个交换区了。

4. 调用try_to_unuse()函数接连强制把这个交换区中剩余的页移到RAM中，并相应地修改使用这些页的进程的页表的内容。对于每个页插槽来说，该函数要执行以下子步骤：

   a. 如果与页插槽相关的计数器等于0（这里没有存放页）或为SWAP_MAP_BAD，就不执行任何操作（继续处理下一个页插槽）。

   b. 否则，如果需要，就调用read_swap_cache()函数（请参看本章后面的“传送交换页”一节）来分配一个新页框并使用这个页插槽中存放的数据填充这个页框。

   c. 对进程链表中的每个进程都调用unuse_process()函数。这个费时的函数要扫描进程的所有页表项，并使用这些页框的物理地址来替换所有被换入的页标识符。为了反映这个替换的结果，该函数还要减少swap_map数组中的页插槽计数器的值并增加页框的引用计数器。

   d. 调用shm_unuse()来检查被换出的页是否用于IPC共享内存资源并用来正确地处理这种情况（请参看第十八章中的“IPC共享内存”一节）。

   e. 如果必要，就把这个页框从交换区高速缓存中删除（请参看本章后面的“交换高速缓存”一节）。

5. 如果try_to_unuse()函数在分配所有请求的页框时失败，那么就不能使这个交换区无效。在这种情况下要把这个交换区描述符重新插入swap_list链表中，将其flags域再次设置成SWP_WRITEOK并返回一个错误码。

6. 否则，所有已用的页插槽都已经被成功传送到RAM中。最后释放与这个交换区相关的目录项对象和索引节点对象、释放用来存放swap_map和swap_lockmap数组的线性区、修改nr_swap_pages变量并返回0（成功）。

查找空闲的页插槽

正如我们将在后面看到的一样，在释放内存时，内核要在很短的时间内把很多页都交换出去。因此尽力把这些页存放在相邻的插槽中非常重要，这样就减少了在访问交换区时磁盘的寻道时间。

搜索空闲插槽的第一种方法可以选择两种策略之一，这两种策略都非常简单，但是非常严格：

- 总是从交换区的开头开始。这种方法在换出操作过程中可能会增加平均定位时间，因为空闲页插槽可能已经被弄得零乱不堪。
- 总是从最后一个分配的页插槽开始。如果交换区的大部分空间都是空闲的（这是最通常的情况），那么这种方法在换入操作过程中会增加平均定位时间，因为所占用的为数不多的页插槽可能零零散散地存放。

Linux采用了一种混合的方法。除非发生以下这些条件之一，否则Linux总是从最后一个分配的页插槽开始查找。

- 已经到达该交换区的末尾。
- 上次从该交换区的开头重新执行之后，已经分配了SWAPFILE_CLUSTER（通常是256）个空闲页插槽。

swap_info_struct描述符的cluster_nr域存放了已经分配的页插槽的个数。该域在函数从交换区的开头重新分配时被重置成0。cluster_next域存放了在下一次分配时要检查的第一个页插槽的索引（注1）。

为了加速对空闲页插槽的搜索过程，内核要保证每个交换区描述符的lowest_bit和highest_bit域是最新的。这两个域定义了第一个和最后一个可能是空闲的页插槽，换而言之，所有低于lowest_bit和高于highest_bit的页插槽都被认为已经分配过了。

scan_swap_map()函数被用来查找一个空闲页插槽。该函数只对一个参数进行操作，该参数指向一个交换区描述符并返回一个空闲页插槽的索引。如果这个交换区并不含有任何空闲插槽，该函数就返回0。该函数执行以下操作：

1. 如果交换区描述符的cluster_nr域是整数，就从cluster_next索引处的元素开始，对计数器的swap_map数组进行扫描，查找一个空项。如果找到一个空项，就减少cluster_nr域的值并转到步骤3。
2. 如果执行到这儿，那么，或者cluster_nr域为空，或者从cluster_next开始的搜索在swap_map数组中没有找到空项。现在就应该开始第二阶段的混合

注1：正如你可能已经注意到的一样，Linux数据结构的名字都不太恰当。在这种情况下，内核并不会真正“聚簇”交换区的页插槽。

查找了。把cluster_nr重新初始化成SWAPFILE_CLUSTER，并重新从lowest_bit索引处开始扫描这个数组。如果没有找到空项，就返回0（该交换区满）。

3. 已经找到了空项。把值1放在这个项中，减少nr_swap_pages的值，如果需要就修改lowest_bit和highest_bit域，并把cluster_next域设置成刚才分配的页插槽的索引。

4. 返回刚才分配的页插槽的索引。

分配和释放页插槽

get_swap_page()函数返回一个新近分配的页插槽，如果所有的交换区都填充满了就返回0。该函数要考虑活动交换区的不同优先级。

该函数需要经过两阶段。第一阶段是局部的，只适用于那些优先级相同的交换区。该函数以轮询的方式在这种交换区中查找一个空闲插槽。如果没有找到空闲插槽，就从交换区列表中开始进入第二阶段。在第二阶段中要对所有的交换区都进行检查。更确切地说，该函数执行以下步骤：

1. 如果nr_swap_pages为空，就返回0。

2. 首先考虑swap_list.next所指向的交换区（回想一下交换区列表是按照优先级从高到低的顺序排列的）。

3. 如果这个交换区是活动的，现在没有被禁用，就调用scan_swap_map()来释放一个空闲插槽。如果scan_swap_map()函数返回一个页插槽索引，该函数的任务基本上就完成了，但是它还要准备下一次被调用。因此，如果下一个交换区的优先级和这个交换区的优先级相同（即轮询使用这些交换区），该函数就把swap_list_next修改成指向交换区列表中的下一个交换区。如果下一个交换区的优先级和现在交换区的优先级不同，该函数就把swap_list_next设置成交换区列表中的第一个交换区（这样下一次搜索操作就会从优先级最高的交换区开始执行）。该函数最终返回和刚才分配的页插槽相对应的标识符。

4. 或者这个交换区是不可写的，或者该交换区中没有空闲页插槽了。如果交换区列表中的下一个交换区的优先级和当前交换区的优先级相同，就把下一个交换区设置成当前交换区并跳转到步骤3。

5. 此时，交换区列表中的下一个交换区的优先级小于前一个交换区的优先级。下一步操作和该函数正在执行哪一阶段有关。

   a. 如果该函数正执行第一阶段(局部阶段)，就考虑该列表中的第一个交换区并跳转到步骤3，这样就开始执行第二阶段。

   b. 否则，就检查交换区列表中是否有下一个元素。如果有，就考虑这个元素并跳转到步骤3。

6. 此时，第二阶段已经扫描完了交换区列表，并没有发现空闲插槽，返回0。

当换入页时，调用swap-free函数以减少相应的swap-map计数器（参看表16-1）。当这个计数器达到0时，由于页插槽的标识符不再包含在任何页表项中，因此它就变成空闲的。该函数要对一个换出页标识符的entry参数进行操作，执行以下步骤：

1. 使用SWP_TYPE和SWP_OFFSET宏来从entry中提取出交换区索引和页插槽索引。

2. 检查这个交换区是否是活动的。如果不是，就直接返回。

3. 如果这个交换区的优先级大于swap_list.next所指向的交换区的优先级，就把swap_list.next设置成swap_list.head，这样下一次对空闲页插槽的搜索就从优先级最高的交换区开始执行。通过这种方式，在其他页插槽从较低优先级的交换区中被分配之前，正在被释放的这个页插槽就被再分配。

4. 如果与正在释放的页插槽对应的swap_map计数器小于SWAP_MAP_MAX，就减少这个计数器的值。回想一下值为SWAP_MAP_MAX的项都被认为是持久的(不可删除的)。

5. 如果swap_map计数器变成0，就增加nr_swap_pages的值，如果需要就修改这个交换区描述符的lowest_bit和highest_bit域。

交换高速缓存

在Linux中，以下情况下可以在几个进程间共享页框：

- 页框与一个共享内存映射(注2)有关(请参看第十五章中的“内存映射”一节)。

注2：私有内存映射使用的页框是通过写时复制机制来处理的，这是后一种情况。

- 页框使用写时复制的方式进行处理，可能是因为已经产生了一个新进程（请参看第七章中的“写时复制”一节）。
- 页框被分配给一个IPC共享内存资源（请参看第十八章中的“IPC共享内存”一节）。

正如我们将在本章中看到的一样，共享内存映射使用的页框永远都不会被交换出去。相反，这些页框是由另外一个内核函数处理的，该函数把这些页框的数据写入适当的文件中并将其丢弃。但是，另外两种共享内存页框必须使用交换算法谨慎处理。

由于内核要单独处理每个进程，因此由两个进程（A和B）共享的一页可能已经被从A进程的地址空间中交换出去了，但却仍然保留在B进程的地址空间中。Linux使用交换高速缓存（swap cache）来处理这种特殊情况，交换高速缓存搜集了已经被拷贝到交换区中的所有共享页框。交换高速缓存并不是作为自己的一个数据结构存在的，而是如果特定的域被设置，那么就会认为正规页高速缓存中的页在交换高速缓存中。

读者在这里也许会提出这样的问题：为什么交换算法不把共享页同时从所有进程的地址空间中交换出去呢，这样就不需要交换高速缓存了。答案是因为没有什么快速方法可以从该页框中获得拥有它的进程的列表。扫描所有进程的页表项来查找给定物理地址项的操作将相当耗时。

共享页交换的工作原理如下。考虑在两个进程（A和B）之间共享的页P。假设交换算法扫描进程A的页框并选择把P换出。分配一个新页插槽并把P中存放的数据拷贝到这个新页插槽中。然后把被换出页标识符放到进程A的相应页表项中。最后调用`__free_page()`释放这个页框。但是，该页的引用计数器并没有变成0，因为P仍然被进程B使用。于是，交换算法就成功地把该页传送到交换空间中，但没有回收相应的页框。

现在假设交换算法稍后又扫描进程B的页框并选择把P换出。内核必须注意到P已经交换到交换区中，不用再次交换出去了。而且，内核还必须能够导出被换出页的标识符，这样就可以增加该页插槽的引用计数器。

图16-4说明了内核对一个共享页所执行的操作，这个共享页是由多个进程在不同时间交换出去的。交换区内的数字和P内的数字分别表示页插槽引用计数器和页引用

计数器。注意任一引用计数计算的是正在使用该页或页插槽的进程个数，如果该页包含在交换高速缓存中，计数还要加上这个交换高速缓存。四个步骤如下所示：

1. 在步骤(a)中，P位于进程A和进程B的页表中。
2. 在步骤(b)中，P已经被从进程A的地址空间中换出。
3. 在步骤(c)中，P已经被从进程A和进程B的地址空间中换出，但是仍然包括在交换高速缓存中。
4. 最后在步骤(d)中，P已经被释放给伙伴系统。

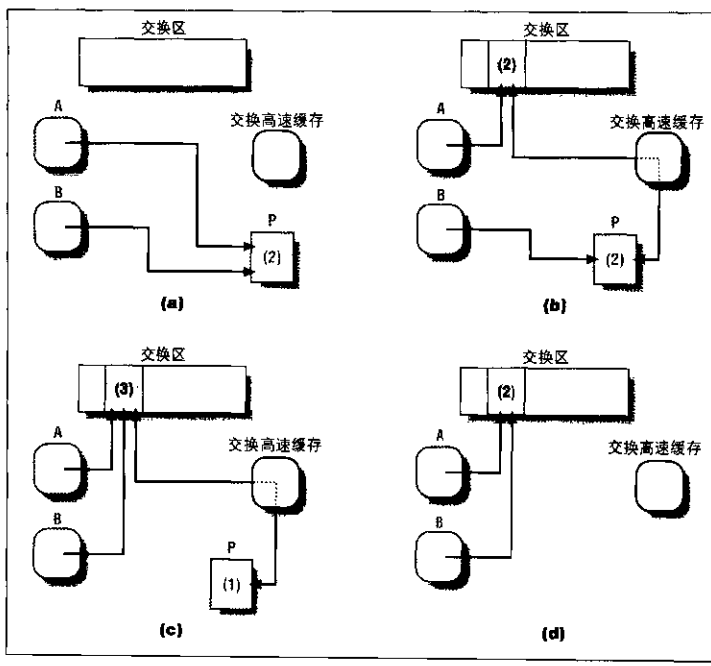


图16-4 交换高速缓存的作用

交换高速缓存是由页高速缓存数据结构实现的，实现过程在第十四章中的“页高速缓存”一节中已经介绍过。回想一下页高速缓存中包含的页是与正规文件相关的，并且散列表允许算法快速从索引节点对象的地址和文件内部偏移量中导出页描述符的地址。交换高速缓存中的页在页高速缓存中以其他页的形式存放，并对其进行以下特殊的处理：

- 页描述符的inode域存放的是包含在swapper_inode变量中的虚拟索引节点对象地址。
- offset 域存放的是与该页相关的被换出页的标识符。
- 在 flags 域中的 PG_swap_cache 标志被设置。

还有，当该页被放入交换高速缓存时，该页描述符的count域和页插槽引用计数器都被增加，因为交换高速缓存既要使用页框，也要使用页插槽。

内核使用几个函数来处理交换高速缓存，这些函数主要是基于第十四章中的“页高速缓存”一节中的讨论。稍后我们将说明这些相对低层的函数是如何被高层函数调用来根据需要换入换出页的。

处理交换高速缓存的函数有：

in_swap_cache()

检查页的PG_swap_cache标志来确定该页是否属于交换高速缓存；如果属于，就返回存放在offset域中的被换出页的标识符。

lookup_swap_cache()

对作为参数传递的被换出页标识符进行操作并返回该页的地址，如果该页不在这个高速缓存中就返回0。该函数调用find_page()函数，把swapper_inode的虚索引节点对象和被换出页标识符的地址作为参数传递来查找所需要的页。如果该页在交换高速缓存中，lookup_swap_cache()就检测该页是否被锁定；如果被锁定，就调用wait_on_page()把当前进程挂起，直到该页被取消锁定为止。

add_to_swap_cache()

把页插入交换高速缓存中。该页描述符的inode和offset域分别被设置成swapper_inode虚索引节点对象的地址和这个被换出页标识符的地址。然后

该函数调用add_page_to_hash_queue()函数和add_page_to_inode_queue()函数。页引用计数器也要增加。

is_page_shared()

如果该页在多个进程之间共享就返回1（true），否则就返回0。虽然该函数并不是专用于交换算法的，但是它必须考虑页引用计数器的值，考虑该页是否在交换区高速缓存中，并且如果必要，还要考虑相应页插槽的引用计数器。此外，该函数还要考虑该页框现在是否参与和交换有关的页I/O操作，因为在这种情况下引用计数器是使用故障保险机制增加的（请参看第十五章中的“从正规文件读取数据”一节）。

delete_from_swap_cache()

清除PG_swap_cache标志并调用remove_inode_page()函数，从而把页从交换高速缓存中移走，还要调用swap_free()来释放相应的页插槽。

free_page_and_swap_cache()

通过调用__free_page()来释放页。该函数还要检测该页是否在交换高速缓存中（PG_swap_cache标志被设置），是否只有一个进程拥有该页［is_page_shared()返回0］。在这种情况下，该函数就调用delete_from_swap_cache()，从而把该页从交换高速缓存中移走并释放相应的页插槽。

传送交换页

如果没有这么多竞争条件和其他潜在危险，传送交换页就不会这么复杂。以下是需要常规检测的一些内容：

- 可能有太多异步交换操作正在执行。当这种情况发生时，只有启动同步I/O操作［请参看本章后面的“rw_swap_page()函数”一节］。
- 拥有一个页的进程在该页正被换入或换出时可能终止了。
- 在当前页试图换出时其他进程可能正在换入该页，反之亦然。

在以下各节中我们使用从上到下的方法进行介绍。首先我们描述一种同步机制，这种机制防止对同一页框或页插槽同时执行I/O操作所引起的数据崩溃。然后我们介绍几个函数，用来执行被交换页的数据传送。

锁定页框和页插槽

与其他磁盘访问类型相同，交换页的I/O数据传送都是阻塞操作。因此，内核必须要谨慎地避免同时传送同一页框、页插槽，或者是同时传送二者。

使用第十三章中讨论的机制可以防止页框的竞争条件。具体说来，在开始对页框执行I/O操作之前，内核调用wait_on_page()函数进行等待，直到PG_locked标志被关掉为止。当该函数返回时，就已经获得了页框锁，因此在这个I/O操作的过程中其他内核控制路径就不能访问该页框的内容。

但是还必须跟踪页插槽的状态。页描述符的PG_locked标志被再次使用来确保对I/O数据传送过程中所涉及的页插槽进行互斥访问。在对交换页开始执行I/O操作之前，内核通常要检测交换高速缓存中是否包含了所涉及的页框；如果没有，就把该页框加入交换高速缓存中。让我们假设某一进程试图换入某一页，但这一页目前正被传送。在执行任何有关换入操作之前，内核都要在交换高速缓存中查找一个与给定换出页标识符相关的页框。找到这个页框之后，内核就知道不必再分配一个新页框，只需使用高速缓存中的页框就可以。此外，由于设置了PG_locked标志，内核要挂起这个内核控制路径直到该位变成0，这样该页框的内容和交换区中的页插槽的内容就一直被保留到这次I/O操作终止时为止。

在一些特殊的情况下，PG_locked标志和交换高速缓存不足以避免竞争条件。例如，让我们假设内核对某些页开始执行一个换出操作。这样，内核就增加该页引用计数器，分配页插槽，然后开始执行I/O数据传送操作。进一步假设在这个操作期间，拥有该页的进程死亡了。内核回收该进程的内存，也就是说，释放该进程使用的所有页框和页插槽。由于页引用计数器是在开始执行I/O操作之前被增加的，交换I/O操作中所涉及的页框是不会被释放给伙伴系统的。但是，交换区描述符中的页插槽引用计数器可能变为0，因此，就可能在上一个I/O操作完成之前使用这个页插槽换出另外一页。如果确实是这样，那么一个内核控制路径就可能对这个页插槽执行写操作，同时另外一个内核控制路径也正在对相同的页插槽（也就是说同一个物理磁盘位置）执行另一个写操作。这样会导致不可预知（令人讨厌）的后果。

为了避免这种问题的出现，内核使用了一个位数组，其地址存放在交换区描述符的swap_lockmap域中。该数组中的每个位都是交换区中页插槽的一个锁。该位在激活相应页插槽的I/O数据传送之前被设置，在操作结束时被清除。如果页插槽的锁位被设置了，上一节“查找空闲的页插槽”中所介绍的scan_swap_map()函数就不会

认为页插槽是“空闲的”，即使其swap_map数组中的引用计数器是空的也是如此。lock_queue等待队列被用来挂起进程，直到swap_lockmap数组中的位被清除为止。

rw_swap_page()函数

正如图16-1所说明的一样，rw_swap_page()函数用来换入/换出页。该函数接收以下参数：

buffer

包含要换入或换出页的页框的起始地址。

entry

被换出页的标识符。该参数有些多余，因为可以从buffer页框的描述符中获得相同信息。

rw

说明数据传送方向的标志：换入是READ，换出是WRITE。

wait

说明内核控制路径是否必须一直阻塞到I/O操作完成为止的一个标志。

缺页所触发的换入操作通常都是同步的（wait等于1），因为进程应该被挂起到所请求的页已经从磁盘上传送过来为止；相反，换出操作通常都是异步的（wait等于0），因为不必把当前进程挂起到I/O操作完成为止。但是，为了防止滥用块设备驱动程序的请求队列，内核对当前要进行的异步交换操作的个数作了限制。这个限制被存放在pager_daemon变量的swap_cluster域中（请参看本章后面的“kswapd内核线程”一节）。如果达到了这个限制，rw_swap_page()就忽略wait参数的值，好象wait等于1一样继续处理。

为了对页进行交换，该函数执行以下操作：

1. 计算与页框对应的页描述符的地址。
2. 从entry中获得交换区索引和页插槽索引。
3. 测试并设置swap_lockmap数组中与该页插槽对应的锁位。如果该位已被设置，说明该页正处于被换入或换出的过程中，我们希望等待这个操作完成。因

此执行tq_disk任务队列中的函数，就可以拔出正在等待的任一块设备驱动程序，并在lock_queue等待队列中睡眠直到正在执行的交换操作完成为止。

4. 设置页的PG_swap_unlock_after标志，从而确保在第10步这个函数开始执行的交换操作的末尾清除swap_lockmap中的标志。(在本节后面将讨论这个标志的影响。)

5. 如果数据传送是为了换入操作（rw被设置成READ），就清除页框的PG_uptodate标志。只有在换入操作成功完成时该标志才会被再次设置。

6. 增加页引用计数器，这样即使拥有该页的进程死亡也不会把该页框释放给伙伴系统（上一节中讨论的故障保险机制）。还要设置PG_free_after标志，从而确保在这个页I/O操作完成时会减少页描述符的引用计数器（请参看第十三章中的“页I/O操作”一节）。该标志表示一种挂起（belt-and-suspenders）的警告情况，因为该函数可能没有首次增加引用计数器就调用了brw_page()函数。实际上，内核从不这样处理。

7. 如果交换区是一个磁盘分区，就定位和该页插槽相关的块（交换区有4096字节的块）。否则，如果交换区是一个正规文件，就调用相应索引节点对象的bmap方法（或调用针对基于扇区的文件系统的smap方法）来获得和该页插槽相关的块的逻辑号。

8. 如果nr_async_pages大于pager_daemon.swap_cluster，就强制把wait参数设置成1（正在进行的异步交换操作太多）。

9. 如果wait为0，就增加nr_async_pages。设置该页的PG_decr_after标志，从而确保该变量在下一步开始执行的交换操作结束时再次被减少。（与PG_swap_unlock_after类似，稍后就会讨论PG_decr_after标志。）

10. 调用brw_page()函数开始执行实际的I/O操作。正如在第十三章的“启动页I/O操作”一节中介绍的一样，该函数在数据传送完成之前返回。

11. 如果wait为0，不用等待数据传送完成就可以返回。

12. 否则（wait等于1），就调用wait_on_page()函数把当前进程挂起，直到页框变为解锁为止，也就是说直到I/O数据传送完成为止。

注意rw_swap_page()函数依赖于brw_page()函数来执行数据传送。正如在第十三章的“终止页I/O操作”一节中介绍的一样，块设备驱动程序不论何时终止页插槽

中的块数据传送，都要调用从相应异步缓冲区首部中获得的b_end_io方法。该方法是使用end_buffer_io_async()函数实现的，如果该页中的所有块都被传送，该函数又要调用after_unlock_page()。后面这个函数执行以下操作：

1. 如果该页的PG_decr_after标志被置位，就清除该标志并增加nr_async_pages变量。正如我们已经看到的一样，这个变量有助于把上限设为当前异步页交换的个数。

2. 如果该页的PG_swap_unlock_after标志被置位，就清除该标志并调用swap_after_unlock_page()。该函数清除和该页插槽有关的swap_lockmap数组中的锁位并唤醒正在lock_queue睡眠队列中睡眠的进程。因此，在该页插槽上正在等待I/O操作完成的进程就可以再次开始执行。

3. 如果该页的PG_free_after被置位，就清除该标志并调用__free_page()来释放这个页框，由此补偿由rw_swap_page()作为故障保险机制所执行的对该页引用计数器的增加。

read_swap_cache_async()函数

正如图16-1所说明的一样，read_swap_cache_async()函数被调用来从交换高速缓存或磁盘中换入页。该函数接收以下参数：

entry
　被换出页的标识符。

wait
　一个标志，说明内核是否允许把当前进程挂起直到该交换的I/O操作完成为止。

顾名思义，该函数的wait参数决定这个I/O交换操作必须是同步的还是异步的。通常用read_swap_cache宏来调用read_swap_cache_async()给wait参数传递值1。

该函数执行以下操作：

1. 对entry调用swap_duplicate()来检测页插槽是否有效，是否要增加页插槽引用计数器。(增加计数器是在查找交换高速缓存过程中所使用的故障保险机制，用来避免万一引起缺页的进程在结束换入操作之前死亡而产生的问题。)

2. 调用lookup_swap_cache()在交换高速缓存中搜索该页。如果找到了该页，就对entry调用swap_free()来减少页插槽引用计数器，并通过返回该页的地址而结束。

3. 该页没有包含在交换高速缓存中。调用__get_free_page()来分配一个新页框。然后再次调用lookup_swap_cache()，因为进程可能在等待新页框时已经被挂起，而且其他内核控制路径可能已经在这段时间内创建了该页。与上一步相同，如果所请求的页找到，就减少该页插槽引用计数器的值，释放刚才分配的页框，并返回所请求页的地址。

4. 调用add_to_swap_cache()来初始化这个新页框描述符的inode和offset域并将其插入交换高速缓存中。

5. 设置这个页框的PG_locked标志。由于该页框是新创建的，其他内核控制路径都不能访问该页框，因此就不必对前面的标志进行检查。

6. 调用rw_swap_page()来从交换区中读取该页的内容，向该函数传递READ参数、被换出页标识符、页框地址以及wait参数。最后，所请求的页就被拷贝到这个页框中。

7. 返回该页的地址。

rw_swap_page_nocache()函数

在少数情况下，内核希望不用把页放入交换高速缓存中就将其从交换区中读出。例如，在处理swapon()系统调用时就会发生这种情况。内核读取一个交换区的第一页，该交换区包含swap_header联合体，然后立即丢弃该页。IPC共享线性区的换出页也从不会包含在交换高速缓存中（请参看第十八章中的"IPC共享内存"一节）。

rw_swap_page_nocache()函数接收I/O操作类型（READ或WRITE）、换出页标识符以及页框地址作为参数。该函数执行以下操作：

1. 调用wait_on_page()，然后设置页框的PG_locked标志。

2. 把该页描述符的inode域初始化为swapper_inode的索引节点对象的地址，把offset域设置成被换出页标识符，并设置PG_swap_cache标志。但是要注意，该函数并不会把该页框插入交换高速缓存的数据结构中，PG_swap_cache

标志、页描述符的 inode 以及 offset 域的初始化仅仅是为了满足 rw_swap_page()函数的一致性检查。

3. 增加该页的引用计数器（故障保险机制）。
4. 调用 rw_swap_page()来开始执行 I/O 交换操作。
5. 减少该页引用计数器，清除 PG_swap_cache 标志，并把页描述符的 indoe 域设置为 0。

页换出

在后面的“释放页框”一节中解释了何时换出页。正如我们在本章开头说明的一样，释放内存也可采用其他策略，但作为一般策略的一部分，页换出（pape swap-out）是最后的一种手段。这一部分我们将说明内核如何执行换出操作。这是通过 swap_out()函数实现的，该函数对以下参数进行操作：

priority

从 0 到 6 范围内的一个整数值，该值说明内核在确定要交换页的位置时应该花费多少时间；该值越小，对应的搜索时间越长。我们在本章后面的“do_try_to_free_pages()函数”一节中将说明如何设置这个参数。

gfp_mask

如果该函数作为一个内存分配请求的结果已经被调用，那么该参数就是传递给分配器函数的 gfp_mask 参数的一个拷贝（请参看第六章的“请求和释放页框”一节）。该参数告诉内核如何处理这个页，特别是告诉内核这个请求是多么紧迫，以及是否可以挂起这个内核控制路径。

swap_out()函数扫描现有的进程，并试图把每个进程页表中所引用的页都交换出去。只要如下条件一发生，该函数就立即结束：

- 该函数成功换出一页。
- 该函数执行某个阻塞操作。它不会不厌其烦地执行扫描操作，因为在当前进程睡眠过程中正在被检查的进程就可能已经被撤消，这样就不需要进一步的页扫描操作。

- 在扫描一定数量的进程以后，函数还是不能换出一页。这是因为内核并不希望花费太多的时间在换出操作上。明确地说，swap_out()的每次调用最多要考虑nr_tasks/(priority+1)个进程。

内核如何选择要处罚的进程呢？在每次调用时，swap_out()函数扫描进程链表并找到进程描述符的swap_cnt域值最大的进程。如果所有进程的swap_cnt域都为0，该函数就再次扫描进程链表，并把swap_cnt域设置成分配给对应进程的页框个数（这个数字可以在进程描述符的mm->rss域中找到）。通过这种方法，具有很多页框的进程通常比具有较少页框的进程更容易受到处罚。

在选定进程之后，swap_out()调用swap_out_process()函数，向其传递进程描述符指针和gfp_mask参数（见图16-1）。如果后面这个函数返回1，swap_out()就结束执行，因为或者页框已经被换出，或者当前进程已经被挂起。否则，swap_out()试图选择另外一个进程，直到达到要检查的进程个数的最大值为止。

swap_out_process()扫描进程的所有线性区并对每个线性区调用swap_out_vma()函数。swap_out_process()所扫描的第一个线性区的地址存放在进程描述符的swap_address域中，因为该域标识了上一次调用该函数所最后扫描的一个线性区，该进程的所有线性区受到同等的处罚。（这似乎是最好的策略，因为内核并不了解每个线性区被访问的频率。）swap_out_process()继续调用swap_out_vma()直到该函数返回1，或者到达线性区链表的末尾为止。在后一种情况下，进程描述符中的swap_cnt域被设置成0，因此在swap_out()检查系统中所有要考虑的其他进程之前，该函数不会再次考虑这个进程。

swap_out_vma()函数检查这个线性区是否没有被加锁，也就是VM_LOCKED标志等于0。然后开始执行一系列操作，在这些操作过程中，该函数要考虑该进程的页全局目录中与这个线性区的线性地址相关的所有目录项。对于这样的每个目录项来说，该函数都要调用swap_out_pgd()函数，该函数又会考虑这个线性区中的地址区间在页中间目录中对应的所有目录项。对于这样的每个项来说，swap_out_pgd()都要调用swap_out_pmd()函数，后者考虑这个线性区中的页在页表中对应的所有项。对于这样的每个页来说，swap_out_pmd()都要调用try_to_swap_out()函数，后者最终确定该页是否可以被换出。如果try_to_swap_out()返回1（说明这个页框被释放，或者当前进程被挂起），那么swap_out_vma()、swap_out_pgd()、swap_out_pmd()和try_to_swap_out()函数的这一连串嵌套调用就结束。

try_to_swap_out()函数

try_to_swap_out()函数试图释放一个给定的页框，或者将其内容丢弃，或者将其内容换出。该函数的参数有：

tsk

进程描述符指针

vma

线性区描述符指针

address

页的初始线性地址

page_table

对address进行映射的tsk页表项的地址

gfp_mask

swap_out()函数的gfp_mask参数，该函数是和上一节末尾介绍的函数调用链一起传递的

如果正在成功换出页或已经执行了一个阻塞I/O操作，那么该函数就返回1。在第二种情况下，继续执行换出操作有很大的风险，因为该函数可能对一个实际上已经不再存在的进程进行操作。如果决定不再交换，该函数就返回0。

try_to_swap_out()函数必须能够识别要求不同响应的各种情况，但是所有响应都共享很多相同的基本操作。具体说来，该函数执行以下步骤：

1. 考虑page_table地址处的页表项。如果Present位是0，就不分配页框，因此就没有什么要交换的，该函数返回0。

2. 如果该页表项的Accessed标志被设置，那么该页框就很年轻。在这种情况下，该函数要清除Accessed标志，设置该页描述符的PG_referenced标志，调用flush_tlb_page()使与该页相关的TLB项无效，并返回0。只有那些Accessed标志为空的页才能被换出。因为该位是由CPU的分页单元在访问每页时自动设置的，只有在上一次对该页调用了try_to_swap_out()函数之后没有再被访问的情况下，该页才能被交换出去。如前所述，Accessed标志提供了一定程度的硬件支持，允许Linux利用最原始的LRU置换算法。

3. 如果该页描述符的PG_reserved或PG_locked标志被设置，就返回0（该页不能被换出）。

4. 如果PG_DMA标志等于0，而且gfp_mask参数指明所释放的页框应该用于ISA的DMA缓冲区，就返回0。

5. 如果该页属于交换高速缓存，就说明该页是和其他进程共享的，而且已经被存放在交换区中。在这种情况下，该页必须被标记成可以换出，但是不用执行内存传送工作。而要执行以下操作：

   a. 从页描述符的offset域中获得被换出页的标识符

   b. 调用swap_duplicate()来增加页插槽引用计数器

   c. 把这个换出页标识符写入页表项

   d. 减少该进程的mm->rss计数器

   e. 调用flush_tlb_page()使和该页有关的TLB项无效

   f. 调用__free_page()减少该页的引用计数器

   g. 返回0（还没有换出页）

6. 如果页表项的Dirty位为0，该页就是“干净”页，就没有必要将其回写到磁盘上，因为内核使用请求调页机制总能恢复它的内容。这样就执行以下子步骤把该页从进程的地址空间中删除：

   a. 把这个页表项设置为0

   b. 减少该进程的mm->rss计数器

   c. 调用flush_tlb_page()使和该页有关的TLB项无效

   d. 调用__free_page()减少该页的引用计数器

   e. 返回0（还没有换出页）

7. 该页为脏页，并且可以被换出。但是仍然要检查内核是否允许执行I/O操作（也就是说，gfp_mask参数中的__GFP_IO标志是否被置位）；如果不允许，就返回0。当内核控制路径不能被挂起时（例如，因为正在执行一个中断处理程序），__GFP_IO标志被清除。

8. 检查包含该页的vma线性区是否有自己的swapout方法。如果有，就执行以下子步骤：

   a. 把这个页表项设置为0

   b. 减少该进程的mm->rss计数器

   c. 调用flush_tlb_page()使和该页有关的TLB项无效

   d. 调用swapout方法。如果该函数返回一个错误码，发送SIGBUS信号给进程tsk

   e. 调用__free_page()减少该页的引用计数器

   f. 返回1（步骤8d所调用的swapout方法可能阻塞，所以swap_out函数的执行必须结束）

9. 该线性区没有定义swapout方法，因此就必须显式地换出该页。（这是最通常的情况。）执行以下子步骤：

   a. 调用get_swap_page()来分配一个新页插槽

   b. 减少该进程的mm->rss域并增加其nswap域（换出页的计数器）

   c. 把换出页标识符写入页表项中

   d. 调用flush_tlb_page()使和该页有关的TLB项无效

   e. 调用swap_duplicate()来增加页插槽引用计数器的值。该值现在为2，一次为进程增加，另外一次为交换高速缓存增加

   f. 调用add_to_swap_cache()把该页加入到交换高速缓存中

   g. 为下一步要开始执行的交换操做准备，设置PG_locked标志（我们并非必须测试这个标志，因为在第3步中我们早就对其测试过。从现在开始，其他内核控制路径再也不能设置这个标志，因为该函数不执行任何阻塞操作。）

   h. 调用rw_swap_page()开始执行一个异步交换操作来把该页写入交换区中

   i. 调用__free_page()来减少该页引用计数器

   j. 返回1（一页被交换出去）

从共享内存映射中换出页

正如我们在第十五章的“内存映射”一节中所看到的一样，磁盘上正规文件的部分与共享内存映射中的页相对应。由于这个原因，内核并不把这些页存放在交换区中，而是更新对应的文件。

共享内存映射段都定义了自己的swapout方法。正如在第十五章的表15-2中所说明的一样，这个方法是使用filemap_swapout()函数实现的，该函数只是调用filemap_write_page()函数强制把该页写入磁盘。

但是，在这种情况下，filemap_write_page()函数不显式地调用do_write_page()函数，后者在第十五章的“把内存映射的脏页刷新到磁盘”一节中已介绍过。其原因是运行该函数可能会引起下面这些讨厌的竞争条件：假设内核获得一个临界文件系统的锁，然后因内存分配请求而开始换出一些页。do_write_page()函数可能试图获得相同的锁，这样就引起死锁。

为了防止出现这种问题，允许把属于共享内存映射的页换出的内核唯一部分就是一个名叫*kpiod*的内核线程，该线程以特定的输入队列对所有的I/O请求进行服务。由于*kpiod*是从执行filemap_write_page()的进程中分离出来的一个内核线程，因此就不会产生死锁。即使在试图获得文件系统的锁时挂起了*kpiod*，进程对filemap_write_page()的执行还可以继续执行，最终将这个锁释放。

只要新请求被加入到*kpiod*的输入队列，*kpiod*内核线程就被唤醒。每个请求都涉及要写入磁盘的共享线性区中的一页。由于内核可能试图一次换出多个页（请参看本章后面的“释放页框”一节），在*kpiod*输入队列中可能堆积了多个请求。这个内核线程继续处理请求，直到队列为空为止。

该队列中的每个元素都是一个pio_request类型的描述符，该类型包括表16-2中所说明的域。pio_first和pio_last变量分别指向队列中的第一个元素和最后一个元素。pio_request描述符是由pio_request_cache的slab分配器高速缓存处理的。

表16-2 pio_request描述符的域

类型	域	说明
struct pio_request *	next	队列中的下一个元素
struct file *	file	文件对象指针

表 16-2 pio_request 描述符的域（续）

类型	域	说明
unsigned long	offset	文件偏移量
unsigned long	page	页起始地址

当被 filemap_swapout()调用时，filemap_write_page()函数调用 make_pio_request()把一个请求加入到 *kpiod* 的输入队列中，而不是调用通常的 do_write_page()来完成自己的数据传送过程。make_pio_request()函数执行以下操作：

1. 增加要写入页的引用计数器。
2. 分配一个新的pio_request描述符。如果没有可用内存，就试图删除一些磁盘高速缓存而不执行任何实际的 I/O 操作。该函数通过调用 try_to_free_pages()函数（参数中 __GFP_IO标志被清除）实现了这个功能［请参看本章后面的“try_to_free_pages()函数”一节］。然后 make_pio_request()就试图再次分配这个请求的 pio_request 描述符。
3. 初始化 pio_request 描述符的域。
4. 把 pio_request 描述符插入请求队列。
5. 唤醒 pio_wait 等待队列中的进程（实际上是 *kpiod* 内核线程）。

make_pio_request()函数不会触发任何 I/O 操作，它只是唤醒 *kpiod* 内核线程。内核执行 kpiod()函数，该函数要考虑输入队列中的所有请求并对每个请求调用 do_write_page()函数把相应的页写入磁盘。然后减少页计数器并把 pio_request 描述符释放给 slab 分配器。当队列中的所有请求都被处理完时，kpiod()就把自己插入 pio_wait 等待队列并转入睡眠状态。

kpiod()必须防止其他可能出现的错误。通常，当一个内核线程请求一些空闲的页框而空闲内存很少时，kpiod()就开始回收一些页。为了做到这点，它需要请求另外一些页框。但是，在这个新请求执行过程中，这个线程不能试图回收页，否则就会出现递归情况。由于这个原因，在每个进程中定义了 PF_MEMALLOC 标志。这个标志从本质上禁止递归调用try_to_free_pages()，因此内核通常都在调用该函数之前设置这个标志，在该函数返回时清除这个标志。具体地说，__get_free_

pages()会检查这个标志的值；如果这个标志置位，就永远不会调用try_to_free_pages()函数。*kpiod*内核线程通常以设置PF_MEMALLOC标志来运行。

页换入

当进程试图在自己的地址空间中对一个页进行寻址，而这个页已经被换出到磁盘上时，必然会发生页换入（page swap-in）。在以下条件发生时，“缺页”异常处理程序就会触发一个换入操作（请参看第七章中的“处理地址空间内的错误地址”一节）：

- 引起异常发生的地址所在的页是一个有效的页，也就是说，它属于当前进程的一个线性区。
- 该页不在内存中，也就是说，页表项中的Present标志被清除。
- 与该页有关的页表项不为0，这意味着它包含一个换出页标识符。

正如在第七章的“请求调页”一节中介绍的一样，do_page_fault()异常处理程序调用handle_pte_fault()函数，后者检查页表项是否是非空的。如果是非空的，就调用do_swap_page()，该函数对以下参数进行操作：

tsk
: 引起“缺页”异常的进程的进程描述符的地址

address
: 引起这个异常的线性地址

vma
: address所在的线性区的线性区描述符地址

page_table
: 对address进行映射的页表项的地址

entry
: 换出页的标识符

write_access
: 一个标志，表示试图执行的访问是读操作还是写操作

Linux 允许每个线性区都包含一个用来执行换入操作的专用函数。需要这种专用函数的线性区将指向函数的指针存放在自己描述符的 swapin 域中。直到最近，IPC 共享线性区都还有一个专用的 swapin 域。但是从 Linux 2.2 开始，任何线性区就没有专用方法了。如果提供了这种方法，do_swap_page()就会执行以下操作：

1. 调用 swapin 方法。返回一个页表项值，该值就包括要分配给进程的页框的地址。
2. 把从 swapin 方法中返回的值写入 page_table 所指向的页表项。
3. 如果页框引用计数器大于 1，而且这个线性区不是共享的，就清除这个页表项的 Read/Write 标志。
4. 增加该进程的 mm->rss 域和 tsk->maj_flt 域。
5. 释放 kernel_flag 全局内核锁，这个锁是在进入这个异常处理程序时获得的。
6. 返回 1。

反之，如果没有定义 swapin 方法，do_swap_page()就调用普通的 swap_in()函数。该函数使用的参数和 do_swap_page()的参数相同，执行以下操作：

1. 调用 lookup_swap_cache()来检查交换高速缓存是否已包含 entry 所指定的页。如果是，就跳到第 4 步。
2. 调用 swapin_readahead()函数从交换区中读取一组 2^n 个页，其中包括所请求的页。值 n 存放在 page_cluster 变量中，该变量通常被设置成 4，但是如果系统少于 32MB 内存，该值就会稍微小一点。每个页都是通过调用 read_swap_cache_async()方法进行读取的，同时还要指定一个非 0 的 wait 参数（异步交换操作）。
3. 万一 swapin_readahead()函数不能读取所请求的页时（例如，因为系统正在执行太多异步交换操作），才对 entry 参数调用 read_swap_cache()函数。回想一下 read_swap_cache()激活一个同步交换操作。结果是当前进程会被挂起，直到该页已经被从磁盘上读出为止。
4. 检查 page_table 所指向的项是否和 entry 不同。如果的确不同，就说明另外一个内核控制路径已经换入所请求的页。因此，就调用 free_page_and_swap_cache()来释放前面获得的页并返回。

5. 调用 swap_free()函数减少 entry 所对应的页插槽的引用计数器。
6. 增加该进程的 mm->rss 域和 min_flt 域。
7. 如果该页由几个进程共享，或者进程正在试图只对它进行读操作，那么该页就在交换高速缓存中。但是，必须修改进程的页表，这样进程才可以找到该页。因此，把所请求页的物理地址与在线性区的 vm_page_prot 域中所找到的保护位都写入 page_table 指向的页表项中。
8. 否则，如果该页不是共享的，而且进程试图对该页进行写入，那么就没有理由将该页保留在交换高速缓存中，因为该页是进程私有的。因此，就调用 delete_from_swap_cache()及上一步所描述的相同信息写入页表项中。但是还要把 Read/Write 和 Dirty 位设置成 1。

释放页框

可以用以下几种可能的方法对页框进行释放：

- 通过回收高速缓存中的一个未用页框。根据高速缓存的类型，要使用以下函数：

  shrink_mmap()
  用于页高速缓存、交换高速缓存和缓冲区高速缓存

  shrink_dcache_memory()
  用于目录项高速缓存

  kmem_cache_reap()
  用于 slab 高速缓存（请参看第六章中的“从高速缓存中释放 slab”一节）

- 通过换出属于进程的匿名线性区的一个页，或者换出属于私有内存映射的一个修改过的页。
- 通过换出一个属于 IPC 共享线性区的页。

正如我们很快就会看到的一样，从这些可能的方法中选择一种方法的工作完全是靠经验来进行的，有很少的理论支持。从某种程度上来说，这种情况类似于对确定进程动态优先级的因素进行评估。主要目的是获得一个可以达到良好系统性能的参数，所以就不用问为什么要这样做。

监视空闲内存

除了nr_free_pages变量（表示当前空闲页框数）之外，内核还依赖于两个表示上限和下限的值。这些值存放在一个名为freepages的结构中（在Linux 2.2中也有一个low域、但是没有使用）：

min

内核为执行临界操作（例如把页交换到磁盘上）保留的页框的最小个数。free_area_init()把该域初始化为2*n*，此处*n*表示以MB为单位的主存大小。结果值必须在10到256之间。

high

nr_free_pages的上限，说明内核可以使用的最大空闲内存。在这种情况下，不执行交换操作；free_area_init()把这个上限值初始化成3 × freepages.min。

min和high域的内容可以通过写入文件*/proc/sys/vm/freepages*进行修改。

从页高速缓存、交换高速缓存和缓冲区高速缓存中回收页

为了从磁盘高速缓存中回收页框，内核要使用shrink_mmap()函数。如果它成功地释放属于页高速缓存、交换高速缓存或缓冲区高速缓存的一个页框，该函数就返回1；否则就返回0。该函数作用于以下两个参数：

priority

在函数放弃并以返回0结束之前要检查的页框总数。该参数的值范围为从0(非常紧急：缩减一切内容）到6（不紧急：试图稍微缩减一点）。

gfp_mask

指定要释放的页框种类的标志。

该函数扫描mem_map数组并查找一个可以释放的页。为了能达到这个要求，这个页必须属于以上几种高速缓存之一、必须未被锁定、必须没有被任何进程使用，并且如果这个页框是为ISA的DMA所请求的，还必须设置了PG_DMA标志。此外，还必须最近没有被访问过。

类似于swap_out_process()在选择要检查的进程的第一个线性区时所遇到的问题，这里也存在一个公平的问题。当shrink_mmap()函数被调用时，它不能总是从头开始扫描mem_map数组，否则，物理地址低的页比物理地址高的页在磁盘高速缓存中的被选择的机会就少得多。clock（注3）静态局部变量与进程描述符的swap_address域起同样的作用，它指向mem_map数组中要检查的下一个页框。

该函数通过执行以下步骤扫描mem_map的页描述符：

1. 把局部变量count初始化成在该函数激活期间要检查的未锁定、非共享页框数$n/2^p$。此处n是存放在num_physpages变量中在系统中可以找到的页框的个数，p等于priority。

2. 如果count大于0，就增加clock并对mem_map[clock]中的页描述符执行以下子步骤：

   a. 如果该页被加锁，如果它的PG_DMA被清除而gfp_mask参数指定一个ISA DMA页，或者如果该页的计数器不等于1，就跳过该页，对下一页从第2步重新开始执行。

   b. 该页是未加锁而且是非共享的，因此减少count。

   c. 如果PG_swap_cache标志被置位，该页就属于交换高速缓存。如果以下条件之一成立，该页就被回收：

      - 其PG_referenced标志没有置位，这意味着从上次调用shrink_mmap()以来该页没有被访问过。（这个标志所起的作用与前面介绍的以简单方式来阻止交换的Accessed标志类似。）
      - 页插槽引用计数器是1（没有进程正在引用该页）。

      如果该页可以被回收，就调用delete_from_swap_cache()清除PG_referenced标志并返回1（一个页框已经被释放）。

   d. 如果该页的PG_referenced标志被置位，就说明该页最近被访问过，因此不能被回收，清除该标志并对下一页从第2步重新开始执行。

   e. 如果该页属于缓冲区高速缓存（也就是说页描述符的buffer域不为0），

注3：这个局部变量名源自于循环转动的时钟指针的思想。该函数当然和系统时钟毫无瓜葛。

而且缓冲区高速缓存的大小大于buffer_mem.min_percent系统参数所定义的上限，就调用try_to_free_buffers()来检查该页中的所有缓冲区是否都未被使用。具体地说，该函数执行以下操作：

1. 考虑该页中的所有缓冲区来确定这些缓冲区是否可以被释放。它们都必须是空闲的（也就是说其引用计数器必须是0）、未加锁的、干净的而且未加保护的。如果其中一个测试失败，就不会执行什么操作。调用wakeup_bdflush()（请参看第十四章中的“把脏缓冲区写入磁盘”一节）并返回0来声明该页没有被释放。
2. 所有缓冲区都未被使用。反复调用remove_from_queues()和put_unused_buffer_head()来释放相应的缓冲区首部。
3. 用该页中的缓冲区个数减去nr_buffers，并用4 KB减去buffermem。
4. 唤醒由于缺乏缓冲区首部而在buffer_wait等待队列中睡眠而被挂起的进程（参见第十四章中的“缓冲区的分配”一节）。
5. 调用__free_page()把该页框释放给伙伴系统，并返回1声明该页框已经被释放了。

如果try_to_free_buffers()返回0，该页就不能被释放：跳到第2步。否则返回1。

f. 如果该页属于页高速缓存（也就是说该页描述符的inode域是非0值）而且页高速缓存的大小大于page_cache.min_percent系统参数所定义的上限，就调用remove_inode_page()（请参看第十四章中的“页高速缓存处理函数”一节）把该页从页高速缓存中删除，并把该页释放给伙伴系统，然后返回1。

3. 如果程序流程执行到此处，还没有释放任何页框，就返回0。

从目录项高速缓存和索引节点高速缓存中回收页

目录项对象本身并不大，但是释放其中一个对象会产生一种级连的效果，即通过释放几个数据结构来最终释放很多内存。调用shrink_dcache_memory()函数把目录项对象从目录项高速缓存中删除。显然，只有任何进程未引用的目录项对象（在第十二章的“目录项对象”一节中定义为未用目录项对象）才可以被删除。

由于目录项高速缓存对象是通过slab分配器分配的，shrink_dcache_memory()函数就可以强制一些slab变成空闲的，这样有些页框就可以因此由kmem_cache_reap()回收（请参看第六章中的“从高速缓存中释放slab”一节）。此外，目录项高速缓存起索引节点高速缓存控制器的作用。因此，当一个目录项对象被释放时，存放相应索引节点对象的缓冲区就变为不可使用，从而shrink_mmap()函数就可以释放相应的缓冲区页。

shrink_dcache_memory()函数与shrink_mmap()函数接收的参数相同。它检查是否允许内核执行I/O操作（在gfp_mask参数中是否设置了__GFP_IO标志），如果允许，就调用prune_dcache()。

传递给后面这个函数的两个参数是：要释放的目录项对象的个数d_nr和要释放的索引节点对象的个数i_nr（因为删除一个目录项也会导致释放一个索引节点）。只要达到了两个目标之一，prune_dcache()就停止缩减目录项高速缓存，第一个参数d_nr的值与priority有关。如果该值为0，shrink_dcache_memory()就传递0给prune_dcache()，这意味着所有未用目录项对象都会被删除。否则，d_nr就被计算为n/priority，此处n是未用目录项对象的总数。shrink_dcache_memory()传递-1作为prune_dcache()的第二个参数，这意味着不用对所释放的索引节点个数强加限制。

prune_dcache()函数扫描未用目录项链表，并对每个要释放的对象调用prune_one_dentry()。后面这个函数执行以下操作：

1. 把这个目录项对象从目录项散列表和其父目录中的目录项对象链表中删除。
2. 调用dentry_iput()，该函数释放使用了d_iput的目录项方法的目录项的索引节点，或者调用iput()函数（如果已定义）。
3. 对目录项的父目录项调用dput()。结果是其引用计数器被减少。
4. 把这个目录项对象返回给slab分配器（请参看第六章中的“从高速缓存释放对象”一节）。

try_to_free_pages()函数

try_to_free_pages()函数是由下面两个函数来调用：

- 当空闲页框个数少于freepages.min，且当前进程的PF_MEMALLOC标志被清除时，由__get_free_pages()函数调用（请参看第六章中的“请求和释放页框”一节）
- 当新分配一个pio_request描述符失败时，由make_pio_request()函数调用（请参看本章前面的“从共享内存映射中换出页”一节）

该函数接收一组gfp_mask标志作为参数，其参数含义与__get_free_pages()函数对应的参数完全相同。具体说来，如果允许内核激活I/O数据传送，那么__GFP_IO标志就被置位，而如果允许内核为了释放内存而丢弃页框的内容，那么__GFP_WAIT标志就被置位。

该函数执行以下两个操作：

- 唤醒*kswapd*内核线程（请参看本章中的“kswapd内核线程”一节）
- 如果在gfp_mask中设置了__GFP_WAIT标志，就调用do_try_to_free_pages()，向其传递gfp_mask参数

do_try_to_free_pages()函数

do_try_to_free_pages()函数被try_to_free_pages()函数和*kswapd*内核线程调用。该函数接受常用的gfp_mask参数，并试图释放至少SWAP_CLUSTER_MAX个页框（通常是32）。还有几个辅助函数被调用来完成这个工作。其中有些函数在释放一个页框后就返回，所以它们必须被重复调用。

do_try_to_free_pages()实现的算法是相当合理的，因为页框是根据使用情况进行释放的。例如，该算法宁可保存目录项高速缓存所使用的页框，也不愿保存slab分配器高速缓存中的未用页框。此外，通过调用那些以减少priority的值面进行回收的函数，do_try_to_free_pages()试图来释放内存。通常，priority越小意味着该函数在退出之前要执行的次数越多。当所有函数都以一个priority值为0而被调用时，do_try_to_free_pages()就放弃。

具体说来，该函数执行以下操作：

1. 通过调用lock_kernel()来获得全局锁。

2. 调用kmem_cache_reap（gfp_mask）从slab分配器高速缓存中回收页框。
3. 把priority局部变量设置成6（最低的优先级）。
4. 通过一系列越来越彻底的搜索操作来释放页，每次循环都增加优先级。特别是，当priority大于或等于0，并且所释放页框的数目小于SWAP_CLUSTER_MAX时，执行以下子步骤：
   a. 反复调用shrink_mmap（priority，gfp_mask），直到不能成功释放属于页高速缓存、交换高速缓存或缓冲区高速缓存的页框为止，或者直到所释放的页框的个数达到SWAP_CLUSTER_MAX为止
   b. 如果允许内核把页写入磁盘（如果gfp_mask中的__GFP_IO标志被置位），反复调用shm_swap（priority，gfp_mask），直到不能成功释放属于IPC共享线性区的页框为止，或者直到所释放的页框个数达到SWAP_CLUSTER_MAX为止
   c. 反复调用swap_out（priority，gfp_mask），直到不能成功地把属于某个进程的页框释放给伙伴系统为止，或者直到所释放的页框个数达到SWAP_CLUSTER_MAX为止
   d. 调用shrink_dcache_memory（priority，gfp_mask）来释放目录项高速缓存中的空闲元素
   e. 减少priority并回到循环开始
5. 调用unlock_kernel()。
6. 如果至少已经释放了SWAP_CLUSTER_MAX个页框，就返回1；否则返回0。

kswapd内核线程

*kswapd*内核线程是另外一种激活回收内存的机制。为什么还需要这个内核线程呢？当空闲内存变得紧缺并且发出另一个内存分配请求时，调用try_to_free_pages()还不足够吗？

不幸的是，实际情形并非如此。有些内存分配请求是由中断和异常处理程序执行的，它们不会阻塞正在等待某一页框被释放的当前进程；还有，有些内存分配请求是由已经获得对临界资源互斥访问权限的内核控制路径实现的，因此就不能激活I/O数

据传送。在极少的情况下，所有的内存分配请求都是由这种内核控制路径完成的，内核将永远不能释放空闲内存。

为了防止出现这种情况，*kswapd* 内核线程就每秒10秒被激活一次。该线程执行kswapd()函数，该函数每次激活时都执行以下操作：

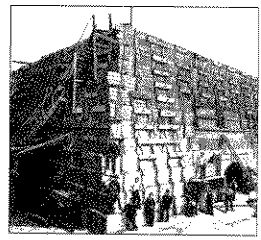
1. 如果nr_free_pages大于freepages.high上限，就不用回收内存，跳到第5步。
2. 把gfp_mask设置成__GFP_IO来调用do_try_to_free_pages()。为了避免递归调用该函数，这个内核线程以PF_MEMALLOC标志置位来执行（请参看本章前面的“从共享内存映射中换出页”一节）。如果该函数不能成功释放SWAP_CLUSTER_MAX个页框，就跳到第5步。
3. 如果current的need_resched域等于0，就跳到第1步（没有更高优先级的进程可以运行，因此就继续回收内存）。
4. need_resched等于1：通过调用schedule()把CPU交给其他进程。*kswapd* 内核线程仍然是可运行的。当该线程重新执行时，跳到第1步。
5. 把current的state设置成TASK_INTERRUPTIBLE。
6. 调用schedule_timeout()，传递给其参数的值为10*HZ，从而强迫该进程自行挂起，10秒以后又重新恢复执行。然后跳到第1步。

对Linux 2.4的展望

现在交换必须考虑RAM区的存在。大部分交换代码已经以更简单更清楚的方法重新编写，这主要归功于新的页高速缓存的实现。交换高速缓存现在仍然是在页高速缓存之上实现的，但是swapper_inode虚索引节点对象已经被一个文件地址空间对象替换。*kpiod* 内核线程已经被取消，因为现在可以直接安全换出共享内存映射页。还有，也不再使用和每个交换区有关的锁数组。

当回收内存时，大部分有趣的改动都涉及选择进程并从中窃取页而采用的策略，这种进程的执行可以更少地引起缺页（回想一下在Linux 2.2中，这是拥有最多页框的进程）。

第十七章
Ext2 文件系统



我们将在本章对I/O和文件系统进行广泛地讨论，主要着眼于当使用一个特定的文件系统时内核所关注的细节问题。因为第二扩展文件系统（Ext2）是Linux所固有的，事实上已在每个Linux系统中得以使用，因此我们自然要对Ext2进行讨论。此外，Ext2在对现代文件系统的高性能支持方面也显示出很多良好的实践性。当然，其他文件系统也具有新而有趣的特点，但它们是为其他操作系统而设计，因此我们在本书中不对各种文件系统和各种平台所具有的特性进行考察。

在“一般特性”一节中引入Ext2后，会像其他章节一样，接着描述所需的数据结构。因为我们着眼于磁盘上存放数据的特定方式，因此必须考虑两种形式的数据结构：“磁盘数据结构”一节显示把Ext2存放在磁盘上的数据结构，而“内存数据结构”一节说明如何把磁盘上的数据结构复制到内存中。

然后，我们讨论对Ext2文件系统所执行的操作。在“创建文件系统”一节，我们讨论如何在磁盘分区创建Ext2。接下来的一部分描述使用磁盘时内核所执行的操作。其中的大部分操作是为索引节点和数据块分配磁盘空间，这些操作相对比较低级。最后，我们讨论如何对Ext2正规文件进行读和写。

一般特性

每个类Unix操作系统都使用自己的文件系统。尽管所有这样的文件系统都与POSIX接口兼容，但每种文件系统的实现方式是不同的。

Linux的第一个版本是基于Minix文件系统的。当Linux成熟时，引入了扩展文件系

统（Ext FS），它包含了几个重要的扩展但提供的性能不令人满意。在1994年引入了第二扩展文件系统（Second Extended Filesystem，Ext2）：它除了包含几个新的特点外，还相当高效和强健，已成为广泛使用的Linux文件系统。

由于Ext2的以下特点，使得它有较高的效率：

- 当创建Ext2文件系统时，系统管理员可以根据预期文件的平均长度来选择最佳块大小（从1024到4096字节）。例如，当文件的平均长度小于几千个字节时，块的大小为1024字节是最佳的，因为这会产生较少的内部碎片，也就是文件长度与存放它的磁盘分区有较少的不匹配（参见第六章中的“内存区管理”一节，在那里讨论了动态内存的内部碎片）。另一方面，大的块对于大于几千字节的文件通常比较合适，因为这导致较少的磁盘传送，因而减轻了系统的开销。
- 当创建Ext2文件系统时，系统管理员可以根据在给定大小的分区上预计存放的文件数来选择给该分区分配多少个索引节点。这可以有效地利用磁盘的空间。
- Ext2文件系统把磁盘块分为组。每组包含在相邻磁道存放的数据块和索引节点。正是这种结构，可以用较少的磁盘平均寻道时间对存放在一个单独块组中的文件进行访问。
- 在磁盘数据块被实际使用之前，Ext2文件系统就把这些块预分配给正规文件。因此，当文件的大小增加时，因为物理上相邻的几个块已被保留，这就减少了文件的碎片。
- 支持快速符号连接。如果符号连接的路径名（参见第一章中的“硬链接和软链接”一节）小于或等于60字节，它就存放在索引节点中而不用通过读一个数据块进行转换。

此外，Ext2还包含了一些使它既强健又灵活的特点：

- 文件更新策略的仔细实现将系统冲突的影响减到最少。例如，当给文件创建一个新的硬连接时，首先增加磁盘索引节点中的硬连接计数器，然后把这个新的名字加到合适的目录中。在这种方式下，如果在更新索引节点后而改变这个目录之前出现一个硬件错误，这样就使索引节点的计数产生错误，但目录是一致的。因此，尽管删除文件时无法自动收回文件的数据块，但并不导致灾难性的后果。如果这种操作的顺序相反（更新索引节点前改变目录），同样的硬件错

误将会导致危险的结果（不一致）：删除最后一个硬连接就会从磁盘删除它的数据块，但新的目录项将指向一个不存在的索引节点。如果那个索引节点号以后又被另外的文件所使用，那么向这个旧目录项的写操作将毁坏这个新的文件。

- 在启动时支持对文件系统的状态进行自动的一致性检查。这种检查是由外部程序 */sbin/e2fsck* 完成的，这个外部程序不仅可以在系统崩溃之后被激活，也可以在一个预定义的文件系统安装数（每次安装操作之后对计数器加1）之后被激活，或者在自从最近检查以来所花的预定义时间之后被激活。
- 支持不可变的文件（不能修改它们）和仅追加（append-only）的文件（只能把数据追加在文件尾）。即使超级用户也不允许他们超越这两种保护。
- 既与 Unix System V Release 4（SVR4）的新文件组 ID 的语义学相兼容，也与BSD的兼容。在SVR4中，新文件采用创建它的进程的组ID；而在BSD中，新文件继承包含它的目录的组ID。Ext2 包含一个安装选项，由你指定采用哪种语义。

在Ext2下一主要版本中正在考虑引入几个另外的特性。一些特性已被实现并以外部补丁的形式来使用。另外一些还仅仅处于计划阶段，但在一些情况下，已经在Ext2的索引节点中为这些特性引入新的域。最重要的一些特点如下：

块片（Block fragmentation）

系统管理员对目前这些磁盘的访问通常选择较大的块。因此，在大块上存放小文件就会浪费很多磁盘空间。这个问题是通过把几个文件存放在同一块的不同片上解决的。

访问控制表（Access Control List，ACL）

访问控制表不是把一个文件的用户分为三类（文件主、同组用户、其他用户），而是对任何特定的用户或用户组，让每个文件与特定的存取权限相关联。

处理压缩和加密文件

这些新的选择（创建一个文件时必须指定）将允许用户在磁盘上存放压缩和或加密的文件。

逻辑删除

一个未删除选项将允许用户在必要时恢复以前已删除的文件内容。

磁盘数据结构

任何Ext2分区中的第一个块从不受Ext2文件系统的管理，因为这一块是为启动扇区所保留的（参见附录一）。Ext2分区的其余部分被分成块组（block group），每个块组的分布图如图17-1所示。正如你从图中所看到的，一些数据结构正好可以放在一块中，而另一些可能需要更多的块。在Ext2文件系统中的所有块组大小相同并被顺序存放，因此，内核可以从块组的整数索引很容易地得到磁盘中一个块组的位置。由于内核尽可能地把属于一个文件的数据块存放在同一块组中，所以块组减少了文件的碎片。块组中的每个块包含下列信息之一：

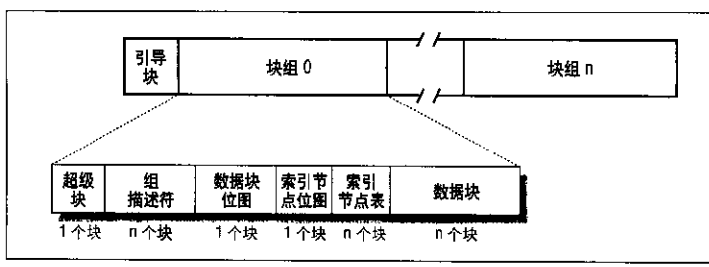


图 17-1 Ext2 分区和 Ext2 块组的分布图

- Ext2文件系统超级块的一个拷贝
- 一组块组描述符的拷贝
- 一个数据块位图
- 一组索引节点
- 一个索引节点位图
- 属于文件的一块数据；即一个数据块

如果一个块中不包含任何有意义的信息，就说这个块是空闲的。

从图17-1中可以看出，超级块与组描述符被复制到每个块组中。只有块组0中所包含的超级块和组描述符才由内核使用，而其余的超级块和组描述符保持不变，事实上，内核甚至不考虑它们。当*/sbin/e2fsck*程序对Ext2文件系统的状态执行一致性

检查时，就引用存放在块组0中的超级块和组描述符，然后把它们拷贝到其他所有的块组中。如果出现数据损坏，并且块组0中的主超级块和主描述符变为无效，那么，系统管理员就可以命令*/sbin/e2fsck*引用存放在一个块组（除了第一个块组）中的超级块和组描述符的旧拷贝。通常情况下，这些多余的拷贝所存放的信息足以让*/sbin/e2fsck*把Ext2分区带回到一个一致的状态。

有多少块组呢？这取决于分区的大小和块的大小。其主要限制在于块位图必须存放在一个单独的块中（块位图用来标识在一组中块的使用和空闲状况）。所以，每组中至多可以有$8 \times b$块，b是以字节为单位的块大小。因此，块组的总数大约是$s/(8 \times b)$，s是总块数。

举例说明，让我们考虑一下8 GB的Ext2分区，块的大小为4KB。在这种情况下，每个4KB的块位图描述32K的数据块，即128MB。因此，最多需要64个块组。显然，块的大小越小，块组数越大。

超级块

Ext2在磁盘上的超级块存放在一个ext2_super_block结构中，它的域在表17-1中列出。__u8、__u16及__u32数据类型分别表示长度为8、16及32位的无符号数，而__s8、__s16、__s32数据类型表示长度为8、16及32位的有符号数。

表17-1 Ext2超级块的域

类型	域	描述
__u32	s_inodes_count	索引节点的总数
__u32	s_blocks_count	以块为单位的文件系统的大小
__u32	s_r_blocks_count	保留的块数
__u32	s_free_blocks_count	空闲块计数器
__u32	s_free_inodes_count	空闲索引节点计数器
__u32	s_first_data_block	第一次使用的块号（总为1）
__u32	s_log_block_size	块的大小
__s32	s_log_frag_size	片的大小
__u32	s_blocks_per_group	每组中的块数
__u32	s_frags_per_group	每组中的片数

表 17-1 Ext2 超级块的域（续）

类型	域	描述
__u32	s_inodes_per_group	每组中的节点数
__u32	s_mtime	最后一次安装操作的时间
__u32	s_wtime	最后一次写操作的时间
__u16	s_mnt_count	安装操作计数器
__u16	s_max_mnt_count	检查之前安装操作的次数
__u16	s_magic	魔数签名
__u16	s_state	状态标志
__u16	s_errors	当检查错误时的行为
__u16	s_minor_rev_level	次版本号
__u32	s_lastcheck	最后一次检查的时间
__u32	s_checkinterval	两次检查之间的时间间隔
__u32	s_creator_os	创建文件系统的操作系统
__u32	s_rev_level	版本号
__u16	s_def_resuid	保留块的缺省 UID
__u16	s_def_resgid	保留块的缺省 GID
__u32	s_first_ino	第一个非保留的索引节点号
__u16	s_inode_size	磁盘上索引节点结构的大小
__u16	s_block_group_nr	这个超级块的块组号
__u32	s_feature_compat	具有兼容特点的位图
__u32	s_feature_incompat	具有非兼容特点的位图
__u32	s_feature_ro_compat	只读兼容特点的位图
__u8 [16]	s_uuid	128 位文件系统标识符
char [16]	s_volume_name	卷名
char [64]	s_last_mounted	最后一个安装点的路径名
__u32	s_algorithm_usage_bitmap	用于压缩
__u8	s_prealloc_blocks	预分配的块数
__u8	s_prealloc_dir_blocks	为目录预分配的块数
__u8 [818]	s_padding	用 null 填充 1024 字节

s_inodes_count域存放索引节点的个数，而s_blocks_count域存放Ext2文件系统的块的个数。

s_log_block_size域以2的幂次方表示块的大小，用1024字节作为单位。因此，0表示1024字节的块，1表示2048字节的块，如此等等。目前s_log_frag_size域与s_log_block_size域相等，因为块片还没有被实现。

s_blocks_per_group、s_frags_per_group与s_inodes_per_group域分别存放每个块组中的块数、片数及索引节点数。

一些磁盘块保留给超级用户(或由s_def_resuid和s_def_resgid域挑选给某一其他用户或用户组)。即使当一般用户没有空闲块可用时，系统管理员也可以用这些块继续使用Ext2文件系统。

s_mnt_count、s_max_mnt_count、s_lastcheck及s_checkinterval域使系统启动时自动地检查Ext2文件系统。在预定义的安装操作数完成之后，或自最后一次一致性检查以来预定义的时间已经用完，这些域就引起*/sbin/e2fsck*的执行(两种检查可以一起进行)。如果Ext2文件系统还没有被全部卸载(例如系统崩溃以后)，或内核在其中发现一些错误强迫启动时也要进行一致性检查。如果Ext2文件系统被安装或未被全部卸载，则s_state域存放的值为0，如果被全部卸载，则这个域的值为1，如果包含错误则值为2。

组描述符和位图

每个块组有自己的组描述符，为ext2_group_desc结构，它的域在表17-2中列出。

表17-2 Ext2组描述符的域

类型	域	描述
__u32	bg_block_bitmap	块位图的块号
__u32	bg_inode_bitmap	索引节点位图的块号
__u32	bg_inode_table	第一个索引节点表块的块号
__u16	bg_free_blocks_count	组中空闲块的个数
__u16	bg_free_inodes_count	组中索引节点的个数
__u16	bg_used_dirs_count	组中目录的个数

表 17-2 Ext2 组描述符的域（续）

类型	域	描述
__u16	bg_pad	对齐到字
__u32 [3]	bg_reserved	用 null 填充 12 个字节

当分配新节点和数据块时用到bg_free_blocks_count、bg_free_inodes_count和bg_used_dirs_count域。这些域确定把最合适的块分配给每个数据结构。位图是位的序列，0表示相应的索引节点块或数据块是空闲的，1表示占用。因为每个位图必须被存放在一个单独的块中，又因为块的大小可以是1024、2048或4096，因此，一个单独的位图描述8192、16,384或32,768个块的状态。

索引节点表

索引节点表由一连串的连续块组成，其中每一块包含索引节点的预定义号。索引节点表第一个块的块号存放在组描述符的bg_inode_table域中。

所有索引节点的大小相同，即128字节。一个1024字节的块可以包含8个索引节点，一个4096字节的块可以包含32个索引节点。为了计算出索引节点表占用了多少块，用一个组中的索引节点总数（存放在超级块的s_inodes_per_group域中）除以每块中的索引节点数。

每个Ext2索引节点是一个ext2_inode结构，它的域在表17-3中列出。

表 17-3 一个 Ext2 磁盘索引节点的域

类型	域	描述
__u16	i_mode	文件类型和访问权限
__u16	i_uid	拥有者的标识符
__u32	i_size	以字节为单位的文件长度
__u32	i_atime	最后一次文件访问的时间
__u32	i_ctime	索引节点最后改变的时间
__u32	i_mtime	文件内容最后改变的时间
__u32	i_dtime	文件删除的时间
__u16	i_gid	组描述符

表17-3 一个Ext2磁盘索引节点的域（续）

类型	域	描述
__u16	i_links_count	硬连接计数器
__u32	i_blocks	文件的数据块数
__u32	i_flags	文件标志
union	osd1	特定的操作系统信息
__u32 [EXT2_N_BLOCKS]	i_block	指向数据块的指针
__u32	i_version	文件版本（对 NFS）
__u32	i_file_acl	文件访问控制表
__u32	i_dir_acl	目录访问控制表
__u32	i_faddr	片的地址
union	osd2	特定的操作系统信息

与POSIX规范相关的很多域类似于VFS索引节点对象的相应域，这已在第十二章的“索引节点对象”一节中讨论过。其余的域与Ext2的特殊实现相关，主要处理块的分配。

特别说明的是，i_size域存放以字节为单位的文件的有效长度，而i_blocks域存放已分配给文件的数据块数（以512字节为单位）。

i_size和i_blocks的值没有必然的联系。因为一个文件总是被存放在整数块中，一个非空文件至少接受一个数据块（因为还没实现片）且i_size可能小于512 × i_blocks。另一方面，我们将在本章后面的“文件的洞”一节中看到，一个文件可能包含有洞。在那种情况下，i_size可能大于512 × i_blocks。

i_block域是有EXT2_N_BLOCKS（通常是15）个指针元素的一个数组，其指针用来标识分配给文件的数据块（参见本章后面的“数据块寻址”一节）。

保留给i_size域的32位把文件的大小限制到最大4GB。事实上，i_size域的最高位没有使用，因此，文件的最大长度被限制为2GB。然而，Ext2文件系统包含一种“脏行为”，允许像康柏的Alpha这样的64位体系结构使用大型文件。从本质上说，索引节点的i_dir_acl域（正规文件没有使用）表示i_size域的32位扩展。因此，文件的大小以64位整数存放在索引节点中。Ext2文件系统的64位版本与32

位版本有些是兼容的，因为一个64位体系结构的Ext2文件系统可以安装在32位体系结构上，反之亦然。但是，在32位体系结构上不能访问大型文件。

回忆一下VFS模型要求每个文件有不同的索引节点号。在Ext2中，没有必要在磁盘上存放文件的索引节点号，因为它的值可以从块组号和它在索引节点表中的相对位置而得出。例如，假设每个块组包含4096个索引节点，我们想知道索引节点13021在磁盘上的地址。在这种情况下，这个索引节点属于第三个块组，它的磁盘地址存放在相应索引节点表的第733个表项中。正如你看到的，索引节点号是Ext2例程用来快速搜索磁盘上合适的索引节点描述符的一个关键字。

各种文件类型如何使用磁盘块

Ext2所认可的文件类型（正规文件、管道文件等）以不同的方式使用数据块。有些文件不存放数据，因此根本就不需要数据块。这部分讨论每种文件类型的存储要求。

正规文件

正规文件是最常用的文件，本章主要关注它。但正规文件只有在开始有数据时才需要数据块。正规文件在刚创建时是空的，并不需要数据块；用系统调用`truncate()`也可以清空它。这两种情况是相同的，例如，当你发布一个包含string *>filename* 的shell命令时，shell创建一个空文件或截断一个现有文件。

目录

Ext2以一种特殊的文件实现了目录，这种文件的数据块存放了文件名和相应的索引节点号。特别说明的是，这样的数据块包含了类型为`ext2_dir_entry_2`的结构。表17-4列出了这个结构的域。因为该结构最后一个名字域是最大为`EXT2_NAME_LEN`（通常是255）个字符的变长数组，因此这个结构的长度是可变的。此外，因为效率的原因，一个目录项的长度总是4的倍数，并在必要时用null字符（`\0`）填充文件名的末尾。`name_len`域存放实际的文件名长度（参见图17-2）。

表17-4 Ext2目录项中的域

类型	域	描述
__u32	inode	索引节点号
__u16	rec_len	目录项长度
__u8	name_len	文件名长度
__u8	file_type	文件类型
char [EXT2_NAME_LEN]	name	文件名

file_type域存放指定文件类型的值（见表17-5），rec_len域可以被解释为指向下一个有效目录项的指针：它是偏移量，与目录项的起始地址相加就得到下一个有效目录项的起始地址。为了删除一个目录项，把它的inode域置为0并适当地增加前一个有效目录项rec_len域的值就足够了。仔细看一下图17-2的rec_len域，你会发现“*oldfile*”项已被删除，因为“*usr*”的rec_len域被置为12+16（“*usr*”和“*oldfile*”目录项的长度）。

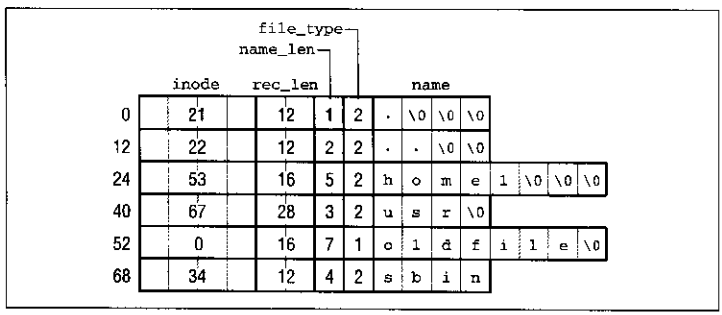


图17-2 EXT2目录的一个例子

符号链

如前所述，如果符号链的路径名达到60个字符，就把它存放在索引节点的i_blocks域，该域是由15个4字节整数组成的数组，因此无需数据块。但是，如果路径名大于60个字符，就需要一个单独的数据块。

设备文件、管道和套接字

这些类型的文件不需要数据块。所有必要的信息都存放在索引节点中。

表 17-5 Ext2 文件的类型

文件类型	描述
0	未知
1	正规文件
2	目录
3	字符设备
4	块设备
5	命名管道
6	套接字
7	符号链

内存数据结构

为了提高效率，当安装 Ext2 文件系统时，存放在 Ext2 分区的磁盘数据结构中的大部分信息被拷贝到 RAM 中，从而使内核避免了后来的很多读操作。那么数据结构如何经常更新呢？让我们考虑一些基本的操作：

- 当创建一个新文件时，必须减少 Ext2 超级块中 s_free_inodes_count 域的值和适当的组描述符 bg_free_inodes_count 域的值。
- 如果内核给一个现有的文件追加一些数据，分配给它的数据块数因此也增加。这就必须修改 Ext2 超级块中 s_free_blocks_count 域的值和组描述符中 bg_free_blocks_count 域的值。
- 即使仅仅重写一个现有文件的部分内容，也要对 Ext2 超级块的 s_wtime 域进行更新。

因为所有的 Ext2 磁盘数据结构都存放在 Ext2 分区的块中，因此，内核利用高速缓存来保持它们是最新的（参见第十四章中的“把脏缓冲区写入磁盘”一节）。

对于与 Ext2 文件系统以及文件相关的每种数据类型，表 17-6 详细说明了在磁盘上

用来表示数据的数据结构、在内存中内核所使用的数据结构、以及单凭经验来决定使用多大容量的高速缓存。频繁更新的数据总需要缓存，也就是说，在相应的Ext2分区被卸载以前，这些数据一直存放在内存并包含在高速缓存中。内核通过让缓冲区的引用计数器一直大于0来达到此目的。

表17-6 Ext2数据结构的VFS映像

类型	磁盘数据结构	内存数据结构	缓存模式
超级块	ext2_super_block	ext2_sb_info	总是缓存
组描述符	ext2_group_desc	ext2_group_desc	总是缓存
块位图	块中的位数组	缓冲中的位数组	固定限制
索引节点位图	块中的位数组	缓冲中的位数组	固定限制
索引节点	ext2_inode	ext2_inode_info	动态
数据块	未指定	缓冲	动态
空闲索引节点	ext2_inode	无	从不缓存
空闲块	未指定	无	从不缓存

在缓冲区高速缓存中不保存“从不缓存”的数据，因为这种数据表示无意义的信息。

在这些极端的模式中，存在另外两种模式：固定限制和动态模式。在固定限制模式中，指定数量的数据结构被保存在缓冲区高速缓存中；当超过这个数时，老的数据结构被刷新到磁盘。在动态模式中，只要相关的对象（索引节点或块）正在使用，其数据就保存在缓冲区高速缓存中；当相应的文件被关闭或块被删除时，shrink_mmap()函数从高速缓存中删除相关的数据并把数据写回磁盘。

ext2_sb_info 和 ext2_inode_info 结构

当Ext2文件系统被安装时，用类型为ext2_sb_info的结构装入VFS超级块的u域（包含特定文件系统的数据），以便内核从总体上能找出与这个文件系统相关的内容。这个结构包含下列信息：

- 磁盘超级块的大部分内容
- 块位图高速缓存，由s_inode_bitmap和s_inode_bitmap_number数组跟踪（参见下一节）

- 索引节点位图高速缓存，由s_inode_bitmap和s_inode_bitmap_number数组跟踪（参见下一节）
- 一个指向磁盘超级块缓冲区的缓冲区首部的指针 s_sbh
- 一个指向磁盘超级块缓冲区的指针 s_es
- 组描述符的个数，s_desc_per_block，被压缩在一个块中
- 一个指向缓冲区的缓冲区首部的一个数组的指针，s_group_desc，缓冲区中包含组描述符（一个单独项就足够了）
- 其他与安装状态、安装选项等相关的数据。

同样地，当属于Ext2文件的索引节点对象被初始化时，用ext2_inode_info类型的结构装载u域，该结构包含下列信息：

- 在磁盘索引节点结构中而不在一般的VFS索引节点对象中的大部分域（参见第十二章中的表12-3）
- 片的大小和片数（还未使用）
- 索引节点所在块组的i_block_group块组索引（参见本章前面的“磁盘数据结构”一节）
- i_alloc_block和i_alloc_count域，在为数据块进行预分配中使用（参见本章后面的“分配一个数据块”一节）
- i_osync域，是一个指定是否同步地更新磁盘索引节点的标志（参见本章后面的“读写Ext2正规文件”一节）

位图高速缓存

当安装一个Ext2文件系统时，内核给Ext2磁盘超级块分配一个缓冲区并从磁盘读取超级块的内容。只有当Ext2文件系统被卸载时才释放这个缓冲区。当内核必须修改Ext2超级块中的一个域时，只是把新值写进相应缓冲区合适的位置，然后把这个缓冲区标记为“脏”。

不幸的是，这种方法并不适合所有的Ext2磁盘数据结构。近年来磁盘容量的数十倍增加导致了索引节点的大小和数据块位图的数十倍增加，因此，我们已经处于这样

一种状况，即把所有的位图同时保存在RAM中不再方便了。例如，考虑4GB的磁盘，块的大小为1KB。因为每个位图填满了一个单独块的所有位，因此其中的每个位图都描述8192个块的状态，即8MB的磁盘空间。块组数是4096 MB/8 MB=512。因为每个块组既需要一个索引节点位图还需要一个数据块位图，因此在内存中存放所有的1024个位图将需要1MB的RAM!

对任一安装的Ext2文件系统，限制Ext2描述符的内存需求所采用的解决方法就是使用大小为EXT2_MAX_GROUP_LOADED（通常为8）的两个高速缓存。一个高速缓存存放大部分最近访问的索引节点位图，而另一个位图存放大部分最近访问的块位图。在高速缓存中包含位图的缓冲区有一个大于0的引用计数器，因此shrink_mmap()从不释放这些缓冲区（参见第十六章中的“从页高速缓存、交换高速缓存和缓冲区高速缓存中回收页”一节）。相反，不在位图高速缓存中的位图缓冲区有一个null的引用计数器，如果空闲内存变得不够时，就释放这些缓冲区。

每个高速缓存都是用有EXT2_MAX_GROUP_LOADED个元素的两个数组实现的。一个数组包含当前在高速缓存中的块组位图的索引，而另一个数组包含引用这些位图的缓冲区首部指针。

ext2_sb_info结构存放属于索引节点位图高速缓存的数组。在s_inode_bitmap域找到块组的索引，在s_inode_bitmap_number域找到指向缓冲区首部的指针。块位图高速缓存的相应数组被存放在s_block_bitmap和 s_block_bitmap_number域。

load_inode_bitmap()函数装入一个指定块组的索引节点位图，并返回这个位图在高速缓存中的位置。

如果位图已不在位图高速缓存中，load_inode_bitmap()就调用read_inode_bitmap()。后一个函数从块组描述符的bg_inode_bitmap域中获得包含这个位图的块号，然后调用bread()分配一个新的缓冲区，如果这个块已不在缓冲区高速缓存中就从磁盘读它。

如果Ext2分区的块组数小于或等于EXT2_MAX_GROUP_LOADED，把这个位图插入到高速缓存数组的某一位置，这个位置的索引总要与作为参数传递给load_inode_bitmap()函数的块组索引相匹配。

否则，如果块组数比高速缓存中的位置多，在必要时采用最近最少使用（LRU）策略从高速缓存中删除一个位图，并把需要的位图插入到高速缓存的第一个位置。图

17-3显示了要引用块组5的三种可能的情况：请求的位图已在高速缓存中、位图不在高速缓存中但有空闲位置以及位图不在高速缓存中也没有空闲位置。

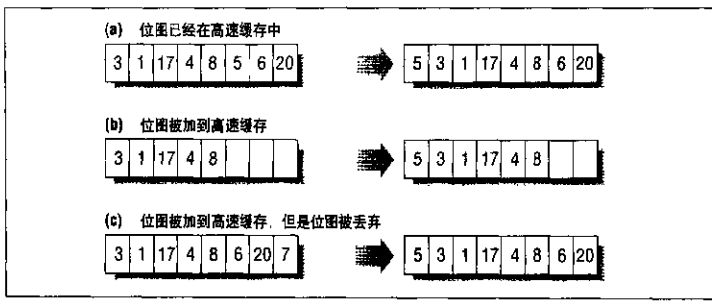


图17-3 给高速缓存增加一个位图

函数load_block_bitmap()和read_block_bitmap()非常类似于load_inode_bitmap()和read_inode_bitmap()，但它们指的是Ext2分区的块位图高速缓存。

图17-4说明了一个安装的Ext2文件系统的内存数据结构。在我们的例子中，有三个块组，其描述符存放在磁盘的三个块中。因此，ext2_sb_info的s_group_desc域指向由这三个缓冲区首部组成的一个数组。尽管内核可以在位图高速缓存中保存2 × EXT2_MAX_GROUP_LOADED个位图，甚至可以在缓冲区高速缓存中存放更多的位图，但我们仅仅显示了索引2的索引节点位图和索引4的块位图。

创建文件系统

格式化一个磁盘分区或软盘与在其上创建一个文件系统不是同一回事。格式化允许磁盘驱动程序读写磁盘上的块，而创建一个文件系统意味着建立本章前面详细描述的结构。

现在的硬盘都由厂商进行了预格式化，不需再进行格式化。软盘则可以用实用程序*/usr/bin/superformat*来进行格式化。

Ext2文件系统是由实用程序*/sbin/mke2fs*创建的。假定用户可以用命令行的标志修改下列的默认选项：

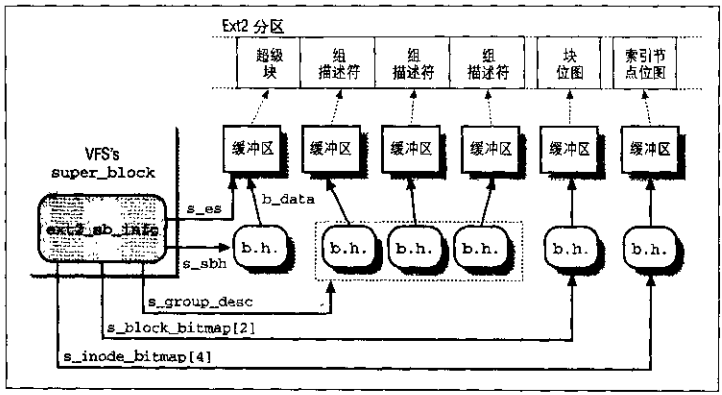


图 17-4 Ext2 的内存数据结构

- 块大小：1024 字节
- 片大小：块的大小
- 所分配的索引节点个数：每4096字节对应一个索引节点
- 保留块的百分比：5%

这个程序执行下列操作：

1. 初始化超级块和组描述符
2. 随意地检查这个分区是否包含有缺陷的块，如果有，创建一个有缺陷块的链表
3. 对于每个块组，保留存放超级块、组描述符、索引节点表及两个位图所需的所有磁盘块
4. 把索引节点位图和每个块组的数据映射位图初始化为 0
5. 初始化每个块组的索引节点表
6. 创建 /root 目录
7. 创建 *lost+found* 目录，由 */sbin/e2fsck* 使用这个目录把丢失的块和有缺陷的块连接起来

8. 在已创建了前两个目录的块组中，更新该块组中的索引节点位图和数据块位图

9. 把有缺陷的块组织起来放在 *lost+found* 目录中

为了具体起见，让我们考虑一下 */sbin/mke2fs* 是如何以缺省选项初始化 Ext2 的 1.4 MB 软盘。

软盘一旦被安装，VFS 就把它看作由 1390 个块组成的一个卷，每块大小为 1024 字节。为了查看磁盘的内容，我们可以执行如下 Unix 命令：

```
$ dd if=/dev/fd0 bs=1k count=1440 | od -tx1 -Ax > /tmp/dump_hex
```

得到了 */tmp* 目录下的一个文件，这个文件包含十六进制的软盘转储内容（注 1）。

通过查看dump_hex文件我们可以看到，由于软盘有限的容量，一个单独的块组描述符就足够了。我们还注意到保留的块数为 72（1440 块的 5%），并且根据默认选项，索引节点表必须为每4096个字节设置一个索引节点，也就是360个索引节点存放在 45 个块中。

表 17-7 概述了按默认选项如何在软盘上建立 Ext2 文件系统。

表 17-7 软盘的 Ext2 块分配

块	内容
0	引导块
1	超级块
2	包含一个单独的块组描述符的块
3	数据块位图
4	索引节点位图
5~49	索引节点表：5~10：保留；11：*lost+found*；12~360：空闲
50	根目录（包括“.”、“..”及“*lost+found*”）
51	*lost+found* 目录（包括“.”及“..”）
52~62	给 *lost+found* 目录预分配保留的块
63~1439	空闲块

注 1：使用 */sbin/dumpesfs* 和 */sbin/debugfs* 实用程序也可以获得有关Ext2文件系统的一些信息。

Ext2 的方法

在第十二章所描述的关于 VFS 的很多方法在 Ext2 都有相应的实现。因为对所有的方法都进行描述将需要整整一本书，因此我们仅仅简单地回顾一下在Ext2中所实现的方法。一旦你真正搞明白了磁盘和内存数据结构，你应当能理解实现这些方法的 Ext2 函数的代码。

Ext2 超级块的操作

除了 VFS 方法中的 clear_inode和umount_begin外，其他的 VFS 超级块操作在 Ext2 中都有专门的实现。超级块方法的地址存放在 ext2_sops 指针数组中。

Ext2 索引节点的操作

很多 VFS 索引节点的操作在Ext2中都有专门的实现，这取决于索引节点所指的文件类型。表17-8列出了对正规文件和目录文件的索引节点所实现的操作。它们的地址分别存放在ext2_file_inode_operations和the ext2_dir_inode_operations 表中。回想一下，当没有定义 Ext2 的相应方法（NULL 指针）时，VFS 就使用自己的通用函数。

表 17-8 对正规文件和目录文件的 Ext2 索引节点的操作

VFS 索引节点操作	Ext2 文件索引节点的方法	Ext2 目录索引节点的方法
lookup	无	ext2_lookup()
link	无	ext2_link()
unlink	无	ext2_unlink()
symlink	无	ext2_symlink()
mkdir	无	ext2_mkdir()
rmdir	无	ext2_rmdir()
create	无	ext2_create()
mknod	无	ext2_mknod()
rename	无	ext2_rename()
readlink	无	无

表17-8 对正规文件和目录文件的Ext2索引节点的操作（续）

VFS索引节点操作	Ext2文件索引节点的方法	Ext2目录索引节点的方法
follow_link	无	无
readpage	generic_readpage()	无
writepage	无	无
bmap	ext2_bmap()	无
truncate	ext2_truncate()	无
permission	ext2_permission()	ext2_permission()
smap	无	无
updatepage	无	无
revalidate	无	无

如果索引节点指的是一个符号链，那么这种索引节点除了readlink和follow_link以外其他方法都为空，这两个方法是分别通过ext2_readlink()和ext2_follow_link()实现的。这些方法的地址存放在ext2_symlink_inode_operations表中。

如果索引节点指的是一个字符设备文件、一个块设备文件或一个命名管道（参见第十八章中的“FIFO”一节），那么这种索引节点的操作不依赖于文件系统，其操作分别位于chrdev_inode_operations、blkdev_inode_operations和fifo_inode_operations表中。

Ext2的文件操作

表17-9列出了Ext2文件系统特定的文件操作。正如你看到的，VFS方法的read和mmap是由很多文件系统共用的通用函数实现的。这些方法的地址存放在ext2_file_operations表中。

表17-9 Ext2的文件操作

VFS的文件操作	Ext2的方法
lseek	ext2_file_lseek()
read	generic_file_read()
write	ext2_file_write()

表 17-9 Ext2 的文件操作（续）

VFS 的文件操作	Ext2 的方法
readdir	无
poll	无
ioctl	ext2_ioctl()
mmap	generic_file_mmap()
open	ext2_open_file()
flush	无
release	ext2_release_file()
fsync	ext2_sync_file()
fasync	无
check_media_change	无
revalidate	无
lock	无

磁盘空间管理

文件在磁盘的存储不同于程序员所看到的文件，这表现在两个方面：块被分散在磁盘上（尽管文件系统尽力保持块连续存放以提高访问时间），而程序员看到的文件似乎比实际的文件大，这是因为程序可以把洞引入文件［通过 lseek() 系统调用］。

在这一节，我们将介绍 Ext2 文件系统如何管理磁盘空间，也就是说，如何分配和释放索引节点和数据块。有两个主要的问题必须考虑：

- 空间管理必须尽力避免文件碎片，也就是说，避免文件存储的物理空间在不相邻盘块的小片上。文件碎片增加了对文件的连续读操作的平均时间，因为在读操作期间，磁头必须频繁地重新定位（注 2）。这个问题类似于在第六章的“伙伴系统算法”一节中所讨论的 RAM 的外部碎片问题。

注 2：请注意，把一个文件跨过块组进行分片是一件坏事，而为了在一个块中存放多个文件把块进行分片（还没实现）是一件好事，这二者之间是不同的。

- 空间管理必须考虑时效性。也就是说，内核应该能从文件的偏移量快速地导出Ext2分区上相应的逻辑块号。内核应该尽可能地限制对磁盘上存放的寻址表的访问次数，因为对该表的访问会极大地增加文件的平均访问时间。

创建索引节点

ext2_new_inode()函数创建Ext2磁盘的索引节点，返回相应的索引节点对象的地址（或失败时为NULL）。它作用于两个参数：新创建的索引节点必须插入到一个目录中，参数dir指的是这个目录的索引节点对象的地址，而参数mode指的是要创建的索引节点的类型。后一个参数还包含一个MS_SYNCHRONOUS标志，这个标志要求当前进程一直挂起到索引节点被分配。该函数执行如下操作：

1. 调用get_empty_inode()分配一个新的索引节点对象，并把它的i_sb域初始化为存放在dir->i_sb中的超级块地址。
2. 调用lock_super()获得对超级块对象的互斥访问。这个函数测试并设置s_lock域的值，如果必要，挂起当前进程直到这个标志变为0。
3. 如果新的索引节点是一个目录，应该尽力安排这个新节点以通过部分地填充块组而使目录都能均匀分散地存放。具体来说，在空闲索引节点数大于平均值（平均值是空闲索引节点总数除以块组的个数）的所有块组中，找一个空闲块最多的块组，从这个块组中给新目录进行分配。
4. 如果新的索引节点不是目录，就在有空闲索引节点的块组中给它进行分配。块组的选择是从包含父目录的块组开始，一直查找下去，具体如下：
   a. 从包含父目录dir的块组开始，执行快速查找算法。这种算法要查找log(*n*)个块组，这里*n*是块组总数。该算法一直向前查找直到找到一个可用的块组，具体如下：如果我们把开始的块组称为i，那么，该算法要查找的块组为*i mod (n)*，*i*+1 *mod (n)*，*i*+1+2 *mod (n)*，*i*+1+2+4 *mod (n)*，……
   b. 如果该算法没有找到含有空闲索引节点的块组，就从第一个块组开始执行彻底的线性查找。
5. 调用load_inode_bitmap()得到所选块组的索引节点位图，并从中寻找第一个空位，这样就得到了第一个空闲磁盘索引节点号。

6. 分配磁盘索引节点：把索引节点位图中的相应位置位，并把含有这个位图的缓冲区标记为脏。此外，如果这个文件系统安装时指定了MS_SYNCHRONOUS标志，则调用ll_rw_block()并等待，直到写操作终止（参见第十二章中的“安装一个普通的文件系统”一节）。

7. 把块描述符的bg_free_inodes_count域减1。如果新的索引节点是一个目录，则增加bg_used_dirs_count。把含有组描述符的缓冲区标记为脏。

8. 把磁盘超级块的s_free_inodes_count域减1，并把包含它的缓冲区标记为脏。把VFS超级块对象的s_dirt域置1。

9. 初始化这个索引节点对象的域。具体说，设置索引节点号i_ino，并把xtime.tv_sec的值拷贝到i_atime、i_mtime及i_ctime。把这个块组的索引赋给ext2_inode_info结构的i_block_group域。关于这些域的含义请参考表17-3。

10. 把新的索引节点对象插入到inode_hashtable。

11. 调用mark_inode_dirty()把这个索引节点对象移到超级块的脏索引节点链表（参看第十二章的“索引节点对象”一节）。

12. 调用unlock_super()释放超级块对象。

13. 返回新索引节点对象的地址。

删除索引节点

用ext2_free_inode()函数删除一个磁盘索引节点，其参数是索引节点对象的地址。内核在进行一系列的清除操作（包括清除文件本身的数据结构和数据）之后，即从索引节点的散列表中删除这个索引节点对象，从适当的目录中删除引用这个索引节点的最后一个硬链接，文件的长度被截为0以收回它的所有数据块之后（参看本章后面的“释放数据块”一节），调用这个函数。这个函数执行下列操作：

1. 调用lock_super()以获得对超级块的互斥访问。

2. 根据每个块组的索引节点号和索引节点的个数计算这个磁盘索引节点所在的块组索引。

3. 调用load_inode_bitmap()以获得索引节点位图。

4. 调用clear_inode()以执行下列操作:

   a. 释放页高速缓存中与这个索引节点相关的所有页，如果其中的一些页被锁定，则挂起当前进程。(这些页被锁可能是因为内核正在读写它们，并且没有办法停止块设备驱动程序。)

   b. 调用超级块对象的clear_inode方法（如果已定义)，但Ext2文件系统没有定义这个方法。

5. 把组描述符的bg_free_inodes_count域加1。如果删除的索引节点是一个目录，也要把bg_used_dirs_count域减1。把这个组描述符所在的缓冲区标记为脏。

6. 把磁盘超级块的s_free_inodes_count域加1，并把包含这个域的缓冲区标记为脏。把超级块对象的s_dirt域置为1。

7. 清除索引节点位图中相应的位，并把包含这个位图的缓冲区标记为脏。此外，如果文件系统以MS_SYNCHRONIZE标志安装、调用ll_rw_block()并等待，直到在位图缓冲区上的写操作终止。

8. 调用unlock_super()以解开对超级块对象的锁定。

数据块寻址

每个非空的正规文件都由一组数据块组成。这些块或者由文件内的相对位置（它们的文件块号）所标识，或者由磁盘分区内的位置（它们的逻辑块号、在第十三章的"缓冲区首部"一节中做了解释）所标识。

从文件内的偏移量f导出相应数据块的逻辑块号需要两个步骤:

- 从偏移量f导出文件的块号，即在偏移量f处的字符所在的块索引。
- 把文件的块号转化为相应的逻辑块号。

因为Unix文件不包含任何控制字符，因此，导出文件的第f个字符所在的文件块号是相当容易的，只是用f除以文件系统块的大小，并取整即可。

例如，让我们假定块的大小为4KB。如果f小于4096，那么这个字符就在文件的第

一个数据块中，其文件的块号为0。如果f等于或大于4096而小于8192，则这个字符就在文件块号为1的数据块中等等。

只用关心文件的块号确实不错。但是，由于Ext2文件的数据块在磁盘上并不是相邻的，因此把文件的块号转换为相应的逻辑块号可不是这么直接了当的。

因此，Ext2文件系统必须提供一种方法，用这种方法可以在磁盘上建立每个文件块号与相应逻辑块号之间的关系。在索引节点内部部分实现了这种映射（回到了AT&T Unix的早期版本）。这种映射也包括一些专门的数据块，可以把这些数据块看成是用来处理大型文件的索引节点的扩展。

磁盘索引节点的i_block域是一个有EXT2_N_BLOCKS个元素且包含逻辑块号的数组。在下面的讨论中，我们假定EXT2_N_BLOCKS的默认值为15。如图17-5所示，这个数组表示一个大型数据结构的初始化部分。正如你从图中所看到的，数组的15个元素有4种不同的类型：

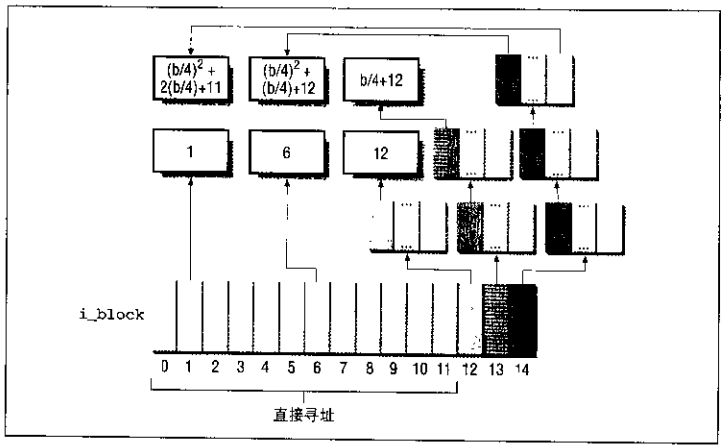


图17-5 对文件的数据块进行寻址的数据结构

- 最初的12个元素产生的逻辑块号与文件最初的12个块对应，即对应的文件块号从0到11。

- 索引12中的元素包含一个块的逻辑块号，这个块代表逻辑块号的一个二级数组。这个数组对应的文件块号从12到$b/4+11$，这里b是文件系统的块大小（每个逻辑块号占4个字节，因此我们在式子中用4做除数）。因此，内核必须先用指向一个块的指针访问这个元素，然后，用另一个指向包含文件最终内容的块的指针访问那个块。
- 索引13中的元素包含一个块的逻辑块号，而这个块包含逻辑块号的一个二级数组，这个二级数组的数组项依次指向三级数组，这个三级数组存放的才是逻辑块号对应的文件块号，范围从$b/4+12$到$(b/4)^2+(b/4)+11$。
- 最后，索引14中的元素利用了三级间接索引，第四级数组中存放的才是逻辑块号对应的文件块号，范围从$(b/4)^2+(b/4)+12$到$(b/4)^3+(b/4)^2+(b/4)+11$。

在图17-5中，块内的个数表示相应的文件块数。箭头（描绘存放在数组元素中的逻辑块号）指示了内核如何找到包含文件实际内容的那个块。

注意这种机制是如何支持小文件的。如果文件需要的数据块小于12，那么两次访问磁盘就可以检索到任何数据：一次是读磁盘索引节点i_block数组的一个元素，另一次是读所需要的数据块。对于大文件来说，可能需要3~4次的磁盘访问才能找到需要的块。实际上，这是一种最坏的估计，因为索引节点、缓冲区及页高速缓存都有助于极大地减少实际访问磁盘的次数。

也要注意文件系统的块大小是如何影响寻址机制的。因为大的块大小允许Ext2把更多的逻辑块号存放在一个单独的块中。表17-10显示了对每种块大小和每种寻址方式所存放文件大小的上限。例如，如果块的大小是1024字节，并且文件包含的数据最多为268KB，那么，通过直接映射可以访问文件最初的12KB数据，通过简单的间接映射可以访问剩余的13到268KB的数据。对于4096字节的块，两次间接映射就完全满足了对2GB文件的寻址（2GB是32位体系结构上的Ext2文件系统所允许的最大值）。

表17-10 数据块寻址的文件大小上界

块大小	直接	一次间接	二次间接	三次间接
1024	12 KB	268 KB	63.55 MB	2 GB
2048	24 KB	1.02 MB	513.02 MB	2 GB
4096	48 KB	4.04 MB	2 GB	—

文件的洞

文件的洞是正规文件的一部分，它是一些空字符但没有存放在磁盘的任何数据块中。洞是Unix文件一直存在的一个特点。例如，下列的Unix命令创建了第一个字节是洞的文件。

```
$ echo -n "X" | dd of=/tmp/hole bs=1024 seek=6
```

现在，*/tmp/hole* 有6145个字符（6144个null字符加一个X字符），然而，这个文件只占磁盘上一个数据块。

引入文件的洞是为了避免磁盘空间的浪费。它们被广泛地用在数据库应用中，更一般地说，用于在文件上执行散列法的所有应用。

文件洞在Ext2的实现是基于动态数据块的分配：只有当进程需要向一个块写数据时，才真正把这个块分配给文件。每个索引节点的i_size域定义程序所看到的文件大小，包括洞，而i_blocks域存放分配给文件有效的数据块数（以512字节为单位）。

在前面dd命令的例子中，假定*/tmp/hole*文件被创建在块大小为4096的Ext2分区上。其相应磁盘索引节点的i_size域存放的数为6145，而i_blocks域存放的数为8（因为每4096字节的块包含8个512字节的块）。i_block数组的第二个元素（对应块的文件块号为1）存放已分配块的逻辑块号，而数组中的其他元素都为空（参看图17-6）。

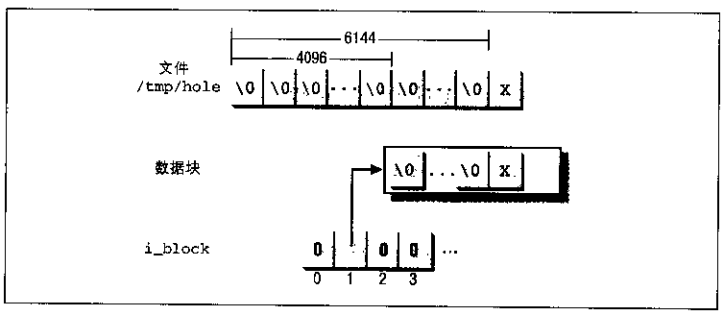


图17-6 起始部分有洞的文件

分配一个数据块

当内核要分配一个新的数据块来保存Ext2正规文件的数据时，就调用ext2_getblk()函数。这个函数依次处理在“数据块寻址”一节中所描述的那些数据结构，并在必要时调用ext2_alloc_block()函数在Ext2分区实际搜索一个空闲的块。

为了减少文件的碎片，Ext2文件系统尽力在已分配给文件的最后一个块附近找一个新块分配给该文件。如果失败，Ext2文件系统又在包含这个文件索引节点的块组中搜寻一个新的块。作为最后一个办法，可以从其他一个块组中获得空闲块。

Ext2文件系统使用数据块的预分配策略。文件并不仅仅获得所需要的块，而是获得一组多达8个邻接的块。ext2_inode_info结构的i_prealloc_count域存放预分配给某一文件但还没有使用的数据块数，而i_prealloc_block域存放下一次要使用的预分配块的逻辑块号。当下列情况发生时，释放预分配但一直没有使用的块：当文件被关闭时，文件被删除时，或关于引发块预分配的写操作而言，有一个写操作不是顺序的时候。

ext2_alloc_block()函数接受的参数为指向索引节点对象的一个指针和一个*goal*。goal是一个逻辑块号，表示新块的首选位置。ext2_getblk()函数根据下列的试探法设置goal参数：

1. 如果正被分配的块与前面已分配的块有连续的文件块号，则goal就是前一块的逻辑块号加1。这很有意义，因为程序所看到的连续的块在磁盘上将会是相邻的。
2. 如果第一条规则不适用，并且至少给文件已分配了一个块，那么goal就是这些块的逻辑块号中的一个。更确切地说，goal是在要分配给文件的块之前已分配块的逻辑块号。
3. 如果前面的规则都不适用，那么goal就是包含文件索引节点的块组中的第一个块（不必空闲）的逻辑块号。

ext2_alloc_block()函数检查goal是否指向文件的某个预分配块。如果是，就分配相应的块并返回它的逻辑块号；否则，丢弃所有剩余的预分配块并调用ext2_new_block()。

ext2_new_block()函数用下列策略在Ext2分区内搜寻一个空闲块：

1. 如果传递给 ext2_alloc_block()的首选块（goal）是空闲的，就分配它。
2. 如果 goal 为忙，检查首选块后的 64 个块之中是否有空闲的块。
3. 如果在首选块附近没有空闲块，就从包含 goal 的块组开始，查找所有的块组。对每个块组：
   a. 寻找至少有 8 个相邻空闲块的一个组块。
   b. 如果没有找到这样的一组块，就寻找一个单独的空闲块。

只要找到一个空闲块，搜索就结束。在结束前，ext2_new_block()函数还尽力在找到的空闲块附近的块中找 8 个空闲块作预分配，并把磁盘索引节点的 i_prealloc_block 和 i_prealloc_count 域置为适当的块位置及块数。

释放数据块

当进程删除一个文件或把它的长度截为 0 时，其所有数据块必须收回。这是通过调用 ext2_truncate()函数（其参数是这个文件的索引节点对象的地址）来完成的。实际上，这个函数扫描磁盘索引节点的 i_block 数组以确定所有数据块的位置，把这些数据块分成物理上相邻的组，然后调用 ext2_free_blocks()函数释放这些组。

ext2_free_blocks()函数释放含有一个或多个相邻数据块的组。除 ext2_truncate()调用它外，当丢弃文件的预分配块时也主要调用它（参见前面的“分配一个数据块”一节）。其参数如下：

inode
　描述文件的索引节点对象的地址

block
　要释放的第一个块的逻辑块号

count
　要释放的相邻块数

这个函数调用 lock_super()获得对 Ext2 文件系统超级块的互斥访问，然后对每个要释放的块执行下列操作：

1. 获得要释放块所在块组的块位图

2. 把块位图中要释放的块的对应位清0，并把位图所在的缓冲区标记为脏
3. 把块组描述符的bg_free_blocks_count域加1，并把相应的缓冲区标记为脏
4. 把超级块的s_free_blocks_count域加1，并把相应的缓冲区标记为脏，设置超级块对象的s_dirt标记
5. 如果Ext2文件系统安装时设置了MS_SYNCHRONOUS标志，则调用ll_rw_block()并等待，直到对这个位图缓冲区的写操作终止

最后，这个函数调用unlock_super()释放超级块。

读写Ext2正规文件

我们在第十二章已经描述了虚拟文件系统如何通过read()或write()系统调用来识别所访问文件的类型，如何调用适当的文件操作表中的相应方法。现在，我们用所有必要的手段来理解在Ext2文件系统中到底是如何实际地读写一个正规文件的。

然而，因为读操作已详细地讨论过，在此无需谈及更多。如表17-9所示，Ext2的读方法是通过generic_file_read()函数实现的，这已在第十五章的“从正规文件读取数据”一节中描述过。

下面让我们集中讨论Ext2的写操作，其实现函数为ext2_file_write()，它作用于如下四个参数：

fd
　被写文件的文件描述符

buf
　被写数据所在的线性区地址

count
　被写的字节数

ppos
　指向一个变量的指针，该变量存放把数据写进文件时在文件中的偏移量

这个函数执行下列操作：

1. 从文件中删除任一超级用户的特权（在第十九章中将描述用*setuid*程序防止入侵）。
2. 如果以O_APPEND标志的设置打开文件，则把被写数据文件的偏移量置为文件的末尾。
3. 如果以同步方式（置O_SYNC标志）打开文件，则把磁盘索引节点的ext2_inode_info结构中的i_osync域置为1。当把一个数据块分配给文件时测试这个标志，只要这个数据块被修改，则内核能同步更新磁盘上的索引节点。
4. 正如以前所做的，从文件的偏移量和文件系统的块大小计算被写的第一个字节所在的文件块号和在块内的相对偏移量（参见前面的“数据块寻址”一节）。
5. 对于要写的每一块，执行下列子步骤：
   a. 调用ext2_getblk()以获得磁盘上的数据块，当必要时分配它。
   b. 如果这个块必须被部分地重写，且缓冲区的内容不是最新的，则调用ll_rw_block()并等待直到读操作终止。
   c. 把要写进这个块的字节从进程的地址空间拷贝到缓冲区，并把缓冲区标记为脏。
   d. 调用update_vm_cache()，用缓冲区高速缓存的内容同步页高速缓存的内容。
   e. 如果以同步方式打开了文件，则把这个缓冲区插入一个局部数组中。如果这个数组被填满（有32个元素），则调用ll_rw_block()来开始写操作并等待，直到写操作终止。
6. 如果以同步方式打开了文件，则清磁盘索引节点的i_osync标志，并调用ll_rw_block()来开始对一直留在局部数组中的缓冲区进行写操作，并等待到I/O数据传送结束。
7. 更新索引节点对象的i_size域。
8. 把索引节点对象的i_ctime和i_mtime域置为xtime.tv_sec，并把这个索引节点标记为脏。
9. 更新变量*ppos，它存放把数据写进文件时在文件中的偏移量（通常是文件指针）。

10. 返回写入文件的字节数。

对 Linux 2.4 的展望

在正规的Ext2文件中，利用generic_file_write()函数进行写操作有几个有益的效果。其中之一是突破了文件大小的2GB限制，即使在32位的体系结构上也可以访问大型文件。然而，在“一般特性”一节的末尾所提到的那些特点还没有在Linux 2.4中得以实现。很可能在目前正在测试的Ext3文件系统中将发布它们。

第十八章
进程通信

本章介绍用户态的进程之间如何进行同步和交换数据。在第十一章中我们已经介绍了很多同步的主题，但是参与的对象是内核控制路径，而不是用户态程序。我们在充分讨论I/O管理和文件系统之后，就可以继续讨论用户态进程的同步。这些进程都要依靠内核来实现彼此之间的同步以及交换数据。

正如我们在第十二章中的“Linux文件加锁”一节中已经看到的一样，通过创建一个文件（可能是空文件）并使用适当的VFS系统调用对该文件加锁和解锁，就可以在用户态进程之间实现粒度很粗的同步。同理，通过把数据存放在使用锁保护的临时文件中就可以在进程之间实现数据的共享。这种方法的代价很高，因为它需要访问磁盘文件系统。出于这个原因，所有的Unix内核都包含一组系统调用，这组系统调用不用与文件系统打交道就可以支持进程通信。而且，已经开发了几个封装函数并将其加入到适当的库来加速进程对内核发出同步请求的执行过程。

通常，应用程序员有调用不同通信机制的各种需求。这里列出Unix系统的基本机制，还有Linux中所特有的允许进程间通信的机制：

管道和FIFO（命名管道）

最适合在进程之间实现生产者/消费者的交互。有些进程往管道中写入数据，而另外一些进程则从管道中读出数据。

信号量

顾名思义，这是在第十一章中的“使用内核信号量加锁”一节中讨论过的内核信号量的用户态版本。

消息

允许进程异步地交换消息（小块数据）。可以认为消息是传递附加信息的信号。

共享内存区

当进程之间在高效地共享大量数据时，这是一种最适合实现的交互方式。

本书并没有介绍另外一种通用的通信机制——套接字（socket）。正如我们在前面章节中介绍的一样，套接字最初引入是为了允许在应用程序和网络接口之间实现数据的通信（请参看第十二章中的“设备文件”一节）。套接字还可以用作相同主机上的进程之间的通信工具。例如，X Window系统图形接口就是使用套接字来允许客户端和X服务器交换数据。我们在本书中并没有包括这些内容，因为这些内容需要详细讨论网络，这已经超出了本书的范围。

管道

管道（pipe）是所有Unix都愿意提供的一种进程间通信机制。管道是进程之间的一个单向数据流：一个进程写入管道的所有数据都由内核定向到另一个进程，另一个进程由此就可以从管道中读取数据。

在Unix的命令shell中，可以使用|操作符来创建管道。例如，下面的语句通知shell创建两个进程，并使用一个管道把这两个进程连接在一起：

```
$ ls | more
```

第一个进程（执行*ls*程序）的标准输出被重定向到管道中；第二个进程（执行*more*程序）从这个管道中读取输入。

注意，执行下面这两条命令也可以得到相同的结果：

```
$ ls > temp
$ more < temp
```

第一个命令把*ls*的输出重定向到一个正规文件中；接下来，第二个命令强制*more*从这个正规文件中读取输入。当然，通常使用管道比使用临时文件更方便，这是因为：

- shell语句比较短，也比较简单。
- 没有必要创建临时正规文件，这个文件以后还必须删除。

使用管道

管道被看作是打开的文件，但在已经装载的文件系统中没有相应的映像。可以使用pipe()系统调用来创建一个新管道，这个系统调用返回一对描述符。进程可以在read()系统调用中使用第一个文件描述符从管道中读取数据，同样地也可以在write()系统调用中使用第二个文件描述符向管道中写入数据。

POSIX只定义了半双工的管道，因此即使pipe()系统调用返回了两个描述符，每个进程在使用一个文件描述符之前仍得把另外一个文件描述符关闭。如果所需要的是双向数据流，那么进程必须通过两次调用pipe()来使用两个不同的管道。

有些Unix系统，例如System V Releas 4，实现了全双工的管道，并允许两个文件描述符既可以被写入也可以被读取。Linux采用了另外一种解决方法：每个管道的文件描述符仍然都是单向的，但是在使用一个描述符之前不必把另外一个描述符关闭。

让我们回顾一下前面的那个例子。当命令shell对ls|more语句进行解释时，实际上要执行以下操作：

1. 调用pipe()系统调用。让我们假设pipe()返回文件描述符3（该管道的读通道）和4（该管道的写通道）。
2. 两次调用fork()系统调用。
3. 两次调用close()系统调用来释放文件描述符3和4。

第一个子进程必须执行*ls*程序，它要执行以下操作：

1. 调用dup2(4, 1)把文件描述符4拷贝到文件描述符1。从现在开始，文件描述符1就代表该管道的写通道。
2. 两次调用close()系统调用来释放文件描述符3和4。
3. 调用execve()系统调用来执行*/bin/ls*程序（请参看第十九章中的“exec类函数”一节）。缺省情况下，这个程序要把自己的输出写到文件描述符为1的那个文件（标准输出）中，也就是说，写入这个管道中。

第二个子进程必须执行*more*程序。因此，该进程执行以下操作：

1. 调用dup2(3, 0)把文件描述符3拷贝到文件描述符0。从现在开始，文件描述符0就代表该管道的读通道。
2. 两次调用close()系统调用来释放文件描述符3和4。
3. 调用execve()系统调用来执行*/bin/more*程序。缺省情况下，这个程序要从文件描述符为0的那个文件（标准输入）中读取输入。也就是说，从这个管道中读取输入。

在这个简单的例子中，管道是被两个进程使用。但是，由于管道的这种实现方式，一个管道可以供任意个进程使用（注1）。显然，如果两个或者更多个进程对同一个管道进行读写，那么这些进程必须使用文件加锁机制（请参看第十一章中的"Linux文件加锁"一节）或者IPC信号量机制［请参看本章后面的"IPC信号量"一节］对自己的访问进行显式地同步。

除了pipe()系统调用之外，很多Unix系统都提供了两个名为popen()和pclose()的封装函数来处理在使用管道的过程中产生的所有脏工作。只要使用popen()函数创建一个管道，就可以使用包含在函数库（fprintf()，fscanf()等等）中的高层I/O函数对这个管道进行操作。

在Linux中，popen()和pclose()都包含在C函数库中。popen()函数接收两个参数：可执行文件的路径名filename和定义数据传输方向的字符串type。该函数返回一个指向FILE数据结构的指针。popen()函数实际上执行以下操作：

1. 使用pipe()系统调用创建一个新管道。
2. 创建一个新进程，该进程又要执行以下操作：
   a. 如果type是r，就把和该管道的写通道相关的文件描述符拷贝到文件描述符1（标准输出）；否则，如果type是w，就把和该管道的读通道相关的文件描述符拷贝到文件描述符0（标准输入）。
   b. 关闭pipe()所返回的文件描述符。
   c. 调用execve()系统调用来执行filename所指定的程序。

注1：由于大部分shell都提供只能用于两个进程连接的管道，需要多于两个进程所使用的管道必须使用诸如C之类的编程语言自行编写。

3. 如果type是r，就关闭和该管道的写通道相关的文件描述符；否则，如果type是w，就关闭和该管道的读通道相关的文件描述符。

4. 返回FILE文件指针所指向的地址，这个指针指向仍然打开的管道所涉及的任一文件描述符。

在popen()函数被调用之后，父进程和子进程就可以通过这个管道交换信息：父进程可以使用该函数所返回的FILE指针来读（如果type是r）或写（如果type是w）数据；子进程所执行的程序分别把输入写入标准输出或从标准输入中读取数据。

pclose()函数接收popen()所返回的文件指针作为参数，它会简单地调用wait4()系统调用并等待popen()所创建的进程的结束。

管道数据结构

我们现在又一次在系统调用的层次考虑问题。只要管道一被创建，进程就可以使用read()和write()这两个VFS系统调用来访问这个管道。因此，对于每个管道来说，内核都要创建一个索引节点对象和两个文件对象，一个文件对象用于读，另外一个对象用于写。当进程希望从管道中读取数据或向管道中写入数据时，必须使用适当的文件描述符。

当索引节点对象指向一个管道时，其u域就包含了一个如表18-1所示的pipe_inode_info结构。

表18-1 pipe_inode_info结构

类型	域	说明
char *	base	内核缓冲区的地址
unsigned int	start	所读取的内核缓冲区的位置
unsigned int	lock	互斥访问使用的加锁标志
struct wait_queue *	wait	管道/FIFO等待队列
unsigned int	readers	读进程的标志（或编号）
unsigned int	writers	写进程的标志（或编号）
unsigned int	rd_openers	在为读操作打开一个FIFO时使用
unsigned int	wr_openers	在为写操作打开一个FIFO时使用

除了一个索引节点对象和两个文件对象之外，每个管道都还有自己的管道缓冲区(pipe buffer)，它是一个单独的页，其中包含了已经写入管道等待读出的数据。该页的地址保存在pipe_inode_info结构的base域中。这个索引节点对象的i_size域存放了已经写入管道等待读出的数据的字节数。从此之后，我们把这个数字称为当前管道的大小(pipe size)。

读进程和写进程都会访问这个管道，因此内核必须记录在缓冲区中的两个当前位置：

- 下一个要读取的字节的偏移量，该值存放在这个pipe_inode_info结构的start域中
- 下一个要写入的字节的偏移量，该值是从start和管道大小中计算出来的

为了防止在管道的数据结构中出现竞争条件，内核禁止对管道缓冲区进行并发访问。为了实现这个功能，必须使用pipe_inode_info数据结构中的lock域。不幸的是，lock域并不能满足需求。正如我们将要看到的一样，POSIX把一些管道操作定义成是原子的。而且，POSIX标准允许写进程在管道满时挂起，等待读进程清空缓冲区(请参看本章后面的“向管道中写入数据”一节)。这些需求可以使用另外一个i_atomic_write信号量满足，这个信号量可以在索引节点对象中找到：它可以在其他写进程由于缓冲区满而挂起时防止进程再开始执行写操作。

管道的创建和撤消

管道是作为一组VFS对象来实现的，它没有相应的磁盘映像。正如我们会在后面的讨论中看到的一样，只要某个进程有一个指向管道的文件描述符，那么这个管道就仍然留在系统中。

pipe()系统调用会调用sys_pipe()函数，后者又会调用do_pipe()函数。为了创建一个新的管道，do_pipe()函数要执行以下操作：

1. 为该管道的读通道分配一个文件对象和一个文件描述符，并把这个文件对象的flag域设置成O_RDONLY，并把f_op域初始化成read_pipe_fops表的地址。
2. 为该管道的写通道分配一个文件对象和一个文件描述符，并把这个文件对象的flag域设置成O_WRONLY，并把f_op域初始化成write_pipe_fops表的地址。

3. 调用get_pipe_inode()函数，该函数为该管道分配一个索引节点对象并对其进行初始化。该函数还要为该管道缓冲区分配一个页，并把该页的地址存放在这个pipe_inode_info结构的base域中。

4. 分配一个目录项对象，并使用这个对象把这两个文件对象和这个索引节点对象链接在一起（请参看第十二章中的“通用文件模型”一节）。

5. 把这两个文件描述符返回给用户态的进程。

发出一个pipe()系统调用的进程是最初唯一一个可以读写访问这个新管道的进程。为了表示该管道实际上既有一个读进程，又有一个写进程，就要把这个pipe_inode_info数据结构的readers和writers域都初始化成1。通常，只要相应管道的文件对象一直由某个进程打开，这两个域才都被设置成1；如果相应的文件对象已经被释放，那么这个域就被设置成0，因为不会再有任何进程访问这个管道。

新进程的创建并不会增加readers和writers域的值，因此这两个值都不会超过1（注2）。但是，父进程仍然使用的所有文件对象的引用计数器的值都会增加[请参看第三章中的“clone()，fork()及vfork()系统调用”一节]。因此，即使父进程死亡时这个对象都不会被释放，该管道仍会一直打开供子进程使用。

只要进程对一个管道的文件描述符调用close()系统调用，内核就要对相应的文件对象执行fput()函数，这会增加引用计数器的值。如果这个计数器变成0，那么该函数就调用这个文件操作的release方法[请参看第十二章中的“close()系统调用”和“与进程相关的文件”两节]。

pipe_read_release()和pipe_write_release()函数都用来实现该管道的文件对象的release方法。这两个函数分别把这个pipe_inode_info结构的readers和writers域设置成0。然后，每个函数都要调用pipe_release()函数。这个函数把在该管道的等待队列中睡眠的所有进程全部唤醒，这样这些进程就可以识别出该管道状态的变化。而且，该函数还要检测是否readers和writers都等于0；如果都等于0，就释放该管道缓冲区的所在的页框。

注2：正如我们将看到的一样，readers和writers域在结合FIFO使用时用作计数器而不是标志。

从管道中读数据

希望从管道中读取数据的进程发出一个read()系统调用，指定与该管道的读通道相关的描述符作为读取的文件描述符。正如在第十二章中的“read()和write()系统调用”一节中介绍的一样，内核最终调用与适当的文件描述符相关的文件操作表中的read方法。在管道的情况下，read方法在read_pipe_fops表中的表项指向pipe_read()函数。

pipe_read()比较复杂，因为POSIX标准定义了管道的读操作的一些要求。表18-2说明了所期望的read()系统调用的行为，它从一个大小（要读取的管道缓冲区的字节数）为p的管道中读取n个字节。注意读操作是非阻塞的：在这种情况下，只要所有可用的字节（即使是0个）一被拷贝到用户地址空间中，读操作就完成（注3）。还要注意只有在该管道为空，而且当前没有进程正在使用与该管道的写通道相关的文件对象时，read()系统调用才会返回0。

表18-2 从一个管道中读取n个字节

	至少有一个写进程		
管道大小p	阻塞读	非阻塞读	没有写进程
p=0	等待某一数据，拷贝这个数据并返回数据的大小	返回-EAGAIN	返回0
0<p<n	拷贝p个字节并返回p：在这个管道的缓冲区还剩0个字节		
p≥n	拷贝n个字节并返回n：在这个管道的缓冲区还剩p-n个字节		

该函数执行以下操作：

1. 确定管道大小（存放在索引节点的i_size域中）是否是0。如果是，还要确定在等待其他进程向该管道中写入数据时该函数必须返回还是被阻塞（请参看表18-2）。I/O操作的类型［阻塞（blocking）或非阻塞（nonblocking）］是通

注3：非阻塞操作通常都是通过在open()系统调用中指定O_NONBLOCK标志进行请求。这个方法并不适合管道，因为管道不能被打开。但是，进程可以通过对相应的文件描述符发出一个fcntl()系统调用来请求对管道执行非阻塞操作。

过该文件对象的f_flags域中的O_NONBLOCK标志来说明的。如果需要，就在把当前进程插入pipe_inode_info数据结构的wait域所指向的等待队列之后，调用interruptible_sleep_on()函数挂起当前进程。

2. 检查pipe_inode_info数据结构的lock域。如果该域不为0，就说明有其他进程正在访问该管道。在这种情况下，根据读操作的类型（阻塞或非阻塞）或者挂起当前进程，或者立即结束这个系统调用。
3. 增加lock域的值。
4. 从该管道的缓冲区中把所请求数目的字节（如果缓冲区太小，就是可用的数目的字节）拷贝到用户地址空间中。
5. 减少lock域的值。
6. 调用wake_up_interruptible()函数，唤醒在该管道的等待队列中睡眠的所有进程。
7. 返回拷贝到用户地址空间的字节数。

向管道中写入数据

希望向管道中写入数据的进程发出一个write()系统调用，指定与该管道的写通道相关的描述符作为这个管道的文件描述符。内核通过调用适当文件对象的write方法来满足这个请求，write_pipe_fops表中相应的项指向pipe_write()函数。

表18-3说明了由POSIX标准所定义的write()系统调用的行为，该函数请求把n个字节写入一个缓冲区中有u个未用的字节的管道中。具体地说，该标准要求涉及少量字节数的写操作必须自动地执行。更确切地说，如果两个或者多个进程并发地写入一个管道，那么任何少于4096个字节（管道缓冲区的大小）的写操作都必须单独完成，而不能与其他进程对同一个管道的写操作交叉进行。但是，超过4096个字节的写操作可能就不是原子的，也可能会强制调用进程挂起。

还有，如果管道没有读进程（也就是说，如果管道的索引节点对象的readers域的值是0），那么任何对管道执行的写操作都会失败。在这种情况下，内核会向写进程发送一个SIGPIPE信号，并停止write()系统调用，使其返回一个-EPIPE错误码，这个错误码就表示我们熟悉的“Broken pipe”消息。

表18-3 把n个字节写入一个管道

可用缓冲区的空间 u	至少有一个读进程		没有读进程
	阻塞写	非阻塞写	
$u<n\leq4096$	等待，直到有 $n-u$ 个字节被释放为止，拷贝 n 个字节，并返回 n	返回 -EAGAIN	发送 SIGPIP 信号并返回 -EPIPE
$n>4096$	拷贝 n 个字节（必要时要等待）并返回 n	如果 $u>0$，就拷贝 u 个字节并返回 u，否则就返回 -EAGAIN	
$u\geq n$	拷贝 n 个字节并返回 n		

pipe_write()函数执行以下操作：

1. 检查管道是否至少有一个读进程。如果不是，就向当前进程发送一个SIGPIPE信号并返回 -EPIPE错误码。
2. 释放管道的索引节点的i_sem信号量，这是通过sys_write()函数实现的（请参看第十二章中的“read()和write()系统调用”一节），可以获得同一个索引节点的i_atomic_write信号量（注4）。
3. 检查要写入的字节数是否不超过管道缓冲区的大小：
   a. 如果没有超过，这个写操作就必须是原子的。因此，就检查这个缓冲区是否有足够的空闲空间来存放要写入的所有字节。
   b. 如果要写入的字节数大于缓冲区的大小，那么只要有空闲空间，这个操作就开始执行。因此，就检查是否至少有一个空闲字节。
4. 如果缓冲区没有足够的空闲空间，而且这个写操作是阻塞的，就把当前进程插入该管道的等待队列并将其挂起，直到有些数据被从该管道中读走为止。注意i_atomic_write信号量还没有被释放，因此其他进程就不能开始对这个缓冲区执行写操作。如果这个写操作是非阻塞的，就返回 -EAGAIN错误码。

注4：i_sem信号量防止多个进程开始写入文件，因此也防止多个进程开始写入管道。由于一些作者不知道的原因，Linux宁愿使用专用的管道信号量。

5. 检查pipe_inode_info数据结构的lock域。如果其值不为0，就说明有其他进程正在对该管道执行读操作，因此就根据这个写操作是阻塞的还是非阻塞的，或者挂起当前进程，或者立即结束这个写操作。
6. 增加lock域的值。
7. 把所请求数目的字节（如果该缓冲区的大小太小，就是空闲字节数）从用户空间拷贝到管道的缓冲区中。
8. 如果还有要写入的字节，就返回步骤4。
9. 在所请求的数据全部都被写入之后，减少lock域的值。
10. 调用wake_up_interruptible()函数唤醒在管道的等待队列中睡眠的所有进程。
11. 释放i_atomic_write信号量并获得i_sem信号量［以便sys_write()可以安全地释放后者］。
12. 返回为管道缓冲区所写的字节数。

FIFO

虽然管道是一种十分简单、灵活、有效的通信机制，但是顾名思义就可以知道它有一个主要的缺点，这就是用户无法打开一个现有的管道。除非管道是由一个公共的祖先进程创建的，否则两个任意进程就不能共享同一个管道。

这个缺点在很多应用程序中都存在。例如，考虑一个数据库引擎服务器，该服务器连续地轮询要发出查询的客户端进程，并把数据库查询的结果返回客户端进程。服务器和给定客户端之间的每次交互都可以使用一个管道进行处理。但是客户端进程通常是根据需要在用户显式查询数据库时由命令shell创建的，因此，服务器进程和客户端进程就不能方便地共享管道。

为了消除这种限制，Unix系统引入了一种称为命名管道（named pipe）或者FIFO［FIFO代表“先进先出（First In, First Out）”：最先写入这个文件的字节总是被最先读出］的特殊文件类型（注5）。

注5： 从System V Release 3开始，FIFO都是作为全双工（双向）对象实现的。

FIFO文件和设备文件类似：二者都有一个磁盘索引节点，但是二者都没有使用数据块。有了磁盘索引节点的帮助，任何进程都可以访问FIFO，因为在系统的目录树中包含FIFO文件名。除了有一个文件名之外，FIFO与无名管道类似，它们也包含一个内核缓冲区来临时存放在两个进程或者多个进程之间进行交换的数据。由于FIFO使用了内核缓冲区，因此其效率要比临时文件高得多。

现在我们回到那个数据库的例子，我们可以使用FIFO来代替管道简单地建立服务器和客户端之间的通信。启动时，服务器创建一个FIFO，由客户端程序用来发出自己的请求。在建立连接之前，每个客户端程序都另外创建一个FIFO（服务器程序可以把查询结果写入这个FIFO），并在自己对服务器发出的最初请求中包括这个FIFO的名字。

创建并打开一个FIFO

进程通过执行mknod()（注6）系统调用创建一个FIFO（请参看第十三章中的“设备文件”一节），传递的参数是这个新FIFO的路径名以及S_IFIFO (0x10000)与这个新文件的权限位掩码进行逻辑或的结果。POSIX引入了一个名为mkfifo()的系统调用专门用来创建FIFO。这个系统调用是在Linux以及System V Release 4中是作为一个调用mknod()的C库函数实现的。

FIFO一旦被创建，就可以使用普通的open()、read()、write()和close()系统调用进行访问，但是VFS对FIFO的处理方法比较特殊，因为FIFO的索引节点及文件操作都是专用的，并且不依赖于FIFO所在的文件系统。

POSIX标准定义了open()系统调用对命名管道执行的操作。这些操作和所请求的访问类型、I/O操作的种类（阻塞或非阻塞）以及其他正在访问这个FIFO的进程的存在状况有关。

进程可以为读操作、写操作或者读写操作打开一个FIFO。与相应的文件对象相关的文件操作根据这三种情况被设置成特定的方法。

注6：实际上，mknod()几乎可以用于创建任何种类的文件：块设备文件、字符设备文件、FIFO甚至是正规文件（但是该函数不能创建目录和套接字）。

当进程打开一个FIFO时，VFS就执行一些与处理设备文件所执行的操作相同的操作（请参看第十三章中的“VFS对设备文件的处理”一节）。和这个打开的FIFO相关的索引节点对象是由一个文件系统相关的read_inode超级块方法进行初始化的。这个方法总要检查磁盘上的索引节点是否代表一个FIFO：

```
if ((inode->i_mode & 00170000) == S_IFIFO)
    init_fifo(inode);
```

init_fifo()函数把这个索引节点对象的i_op域设置成fifo_inode_operations表的地址。该函数还要把存放在这个索引节点对象中的pipe_inode_info数据结构的所有域都初始化为0（请参看表18-1）。

然后filp_open()函数［该函数由sys_open()函数调用，请参看第十二章中的“open()系统调用”一节］就填充这个索引节点对象的其余域，并把这个新文件对象的f_op域初始化为这个索引节点对象的i_op->default_file_ops域的内容。结果是这个文件操作表被设置成def_fifo_fops。然后filp_open()函数就从这个操作表中调用open方法，在本例的特殊情况中，这是由fifo_open()函数实现的。

fifo_open()函数会检查pipe_inode_info数据结构中的readers和writers域的值。在引用FIFO时，这两个域分别存放了读进程和写进程的个数。如果需要，该函数会把当前进程挂起，直到有读进程或写进程访问这个FIFO为止。表18-4说明了fifo_open()函数可能的行为。而且该函数还会通过把这个文件对象的f_op域设置成如表18-5所示的预定义的表的地址，从而为要使用的文件操作集进一步确定特定的行为。最后，该函数还要检查这个pipe_inode_info数据结构的base域是否为NULL；如果为NULL，该函数要为这个FIFO的内核缓冲区获得一个空闲页框并将其地址存放在base域中。

表18-4 fifo_open()函数的行为

访问类型	阻塞	非阻塞
只读，会写	成功返回	成功返回
只读，不写	等待一个写程序	成功返回
只写，会读	成功返回	成功返回
只写，不读	等待一个读程序	返回-ENXIO
读/写	成功返回	成功返回

FIFO的四个专用文件操作表的主要区别是read和write方法的实现不同。如果访问类型允许读操作，那么read方法是使用pipe_read()函数实现的；否则，read方法就是使用bad_pipe_r()函数实现的，该函数只是返回一个错误码。类似地，如果访问类型允许写操作，那么write方法就是使用pipe_write()函数实现的；否则，write方法就是使用bad_pipe_w()函数实现的，该函数也只是返回一个错误代码。

根据POSIX标准，即使FIFO没有写进程，进程也可以采用非阻塞模式为读操作成功打开这个FIFO。在这种情况下，不能直接使用pipe_read()函数实现read方法，因为在它发现该管道为空而且没有写进程时就返回一个-EAGAIN错误码。采用的解决方案就是使用一个中间函数connect_read()实现了read方法，如果没有写进程，该函数就返回0；否则，就把这个文件对象的f_op域设置成read_fifo_fops，然后调用pipe_read()。

表18-5 FIFO的文件操作

访问类型	文件操作	read方法	Write方法
只读，会写	read_fifo_fops	pipe_read()	bad_pipe_w()
只读，不写	connecting_fifo_fops	connect_read()	bad_pipe_w()
只写	write_fifo_fops	bad_pipe_r()	pipe_write()
读/写	rdwr_fifo_fops	pipe_read()	pipe_write()

从一个FIFO中读写数据

read()和write()系统调用对于FIFO的引用和对其他文件类型的引用一样，都是由VFS通过read()和write()文件对象方法进行处理的。如果允许对文件进行操作，那么这个文件操作表中的相应项就指向pipe_read()和pipe_write()函数（请参看前面的“从管道中读数据”和“向管道中写入数据”两节）。

VFS由此就可以对FIFO和无名管道进行同样的读写处理。但是，与无名管道形成对照，同一文件描述符既可以用于FIFO的读操作，也可以用于FIFO的写操作。

System V IPC

IPC 是进程间通信（Interprocess Communication）的缩写。它表示一组系统调用，这组系统调用允许用户态进程：

- 通过信号量和其他进程进行同步
- 向其他进程发送消息或者从其他进程处接受消息
- 和其他进程共享一个线性区

IPC 最初是在一个名为“Columbus Unix”的开发版 Unix 变体中引入的，之后在 AT&T 的 System III 中采用。现在在大部分 Unix 系统包括 Linux 中都可以找到它。

IPC 数据结构是在进程请求 IPC 资源（IPC resource，信号量、消息队列或者共享内存段）时动态创建的。每个 IPC 资源都是持久的，除非被进程显式地释放，否则永远驻留在内存中。任何进程都可以使用 IPC 资源，即使祖先进程创建了 IPC 资源，但有些进程并不共享这个祖先进程，这样的进程也可以使用 IPC 资源。

由于一个进程可能需要相同类型的多个 IPC 资源，因此每个新资源都是使用一个 32 位的 IPC 关键字（IPC key）来标识的，这和系统的目录树中的文件名类似。每个 IPC 资源都有一个 32 位的 IPC 标识符（IPC identifier），这与和打开文件相关的文件描述符有些类似。IPC 标识符由内核分配给 IPC 资源，在系统内部是唯一的，而 IPC 关键字可以由程序员自由地选择。

当两个或者更多的进程要通过一个 IPC 资源进行通信时，这些进程都要引用该资源的 IPC 标识符。

使用 IPC 资源

IPC 资源的创建是根据这个新资源是信号量、消息队列还是共享内存段分别调用 `semget()`、`msgget()` 或者 `shmget()` 函数实现的。

这三个函数的主要目的都是从 IPC 关键字（作为第一个参数传递）中获得相应的 IPC 标识符，进程以后就可以使用这个标识符对这个资源进行访问。如果还没有 IPC 资

源和这个IPC关键字相关，就会创建一个新的资源。如果一切运行良好，那么该函数就返回一个正的IPC标识符；否则，就返回一个如表18-6所示的错误码。

表18-6 在请求IPC标识符时返回的错误码

错误码	说明
EACCESS	进程没有适当的访问权限
EEXIST	进程试图创建一个和现有的关键字相同的IPC资源
EIDRM	把这个资源标记成要删除的
ENOENT	不存在具有所请求的关键字的IPC资源，而且进程没有请求创建这个资源
ENOMEM	没有存储空间供IPC资源使用
ENOSPC	已经超过了IPC资源数目的最大限制

假设两个独立的进程想要共享一个公共的IPC资源。这可以使用两种方法来实现：

- 这两个进程统一使用固定的、预定义的IPC关键字。这是最简单的情况，对于由很多进程实现的任一复杂的应用程序这种方式也工作得很好。然而，也有可能另外一个无关的程序也使用了相同的IPC关键字。在这种情况下，IPC函数可能被成地功调用，但返回错误资源的IPC标识符（注7）。
- 一个进程通过指定IPC_PRIVATE作为自己的IPC关键字来发布semget()、msgget()或shmget()函数。因此一个新的IPC资源被分配，并且这个进程或者可以与应用程序中的另一个进程共享自己的IPC标识符（注8），或者可以创建另一个进程。这种方法可以确保IPC资源不会偶然被其他应用程序使用。

semget()、msgget()和shmget()函数的最后一个参数都可以包括两个标志。IPC_CREAT说明如果这个IPC资源不存在，就必须创建它；IPC_EXCL说明如果这个资源已经存在而且设置了IPC_CREAT标志，那么该函数就必定失败。

注7：ftok()函数试图从作为参数传递的文件路径名和一个8位对象标识符中获得一个新关键字。但是这并不能担保是一个唯一关键字，因为也有可能使用不同路径名和对象标识符的两个不同应用程序会返回同一个IPC关键字，不过这种机会很小。

注8：当然，这就意味着存不基于IPC的进程之间的另一个通信通道。

即使进程使用了IPC_CREAT和IPC_EXCL标志，也没有办法保证对一个IPC资源进行互斥访问，因为其他进程也可能用自己的IPC标识符引用了这个资源。

为了把不正确地引用错误资源的风险降到最小，内核不会在IPC标识符一空闲时就反复使用它。相反，分配给一个资源的IPC标识符总是大于分配给前一个相同类型的资源的标识符。(唯一的例外发生在32位的IPC标识符溢出时。)每个IPC标识符的计算都是通过结合使用相对于资源类型的位置使用序号（slot usage sequence number)、已分配资源的任意位置索引（slot index）以及内核中为可分配资源的最大数目所选定的值。如果我们使用s来代表位置使用序号，M来代表资源的最大数目，i来代表位置索引，此处$0 \le i < M$，那么每个IPC资源的ID都可以按如下来计算:

$$\text{IPC 标识符} = s \times M + i$$

位置使用序号s被初始化成0、每次分配资源时增加1。在两次连续资源分配时，位置索引i才能增加。该值只有在资源已经被回收时才会减少，但是此时增加的位置使用序号就可以确保下一次分配的资源的新IPC标识符大于前一个标识符。

每个IPC资源都和一个ipc_perm数据结构关联，该结构的域如表18-7所示。uid、gid、cuid和cgid分别存放了该资源的创建者的用户标识符、组标识符以及当前资源属主的用户标识符和组标识符。模式位掩码包括六个标志，分别存放了该资源的属主、组以及其他用户的读、写访问权限。IPC访问许可权和第一章中的“访问权限和文件模式”一节中介绍的文件访问许可权类似，唯一不同的是这里没有执行许可权标志。

表18-7 ipc_perm结构中的域

类型	域	说明
int	key	IPC关键字
unsigned short	uid	属主UID
unsigned short	gid	属主GID
unsigned short	cuid	创建者UID
unsigned short	cgid	创建者GID
unsigned short	mode	许可权位掩码
unsigned short	seq	位置使用序号

ipc_perm数据结构也包括一个key域和一个seq域，前者指的是相应资源的IPC关键字，后者存放的是用来计算该资源的IPC标识符所使用的位置使用序号。

semctl()、msgctl()和shmctl()函数都可以用来处理IPC资源。IPC_SET子命令允许进程来改变属主的用户标识符和组标识符以及ipc_perm数据结构中的许可权位掩码。IPC_STAT和IPC_INFO子命令取得和资源有关的信息。最后IPC_RMID子命令释放IPC资源。根据IPC资源的种类的不同，还可以使用其他专用的子命令(注9)。

一旦一个IPC资源被创建，进程就可以通过一些专用函数对这个资源进行操作了。进程可以执行semop()函数获得或释放一个IPC信号量。当进程希望发送或接收一个IPC消息时，就分别使用msgsnd()和msgrcv()函数。最后，进程可以分别使用shmat()和shmdt()函数把一个共享内存段附加到自己的地址空间中或者取消这种附加关系。

ipc()系统调用

所有的IPC函数都必须通过适当的Linux系统调用实现。实际上，在Intel 80x86体系结构中，只有一个名为ipc()的IPC系统调用。当进程调用一个IPC函数时，比如说msgget()，该函数实际上调用C库中的一个封装函数，该封装函数又通过传递msgget()的所有参数加上一个适当的子命令代码(在本例中是MSGGET)来调用ipc()系统调用。sys_ipc()服务例程检查子命令代码，并调用内核函数实现所请求的服务。

ipc()"多路复用"系统调用是从早期的Linux版本中继承而来的，它在一个动态模块中包括了IPC代码(参见附录二)。在system_call表中为可能未实现的内核部分保留几个系统调用入口并没有什么意义，因此内核设计者就采用了多路复用的方法。

现在，System V IPC不再作为一个动态模块被编译，因此也就没有理由使用单个IPC系统调用。事实上，Linux在康柏的Alpha体系结构上为每个IPC函数都提供了一个系统调用。

注9：IPC设计的另外一个缺点是用户态的进程不能自动创建并初始化IPC资源，因为这两个操作是由两个不同的IPC函数执行的。

IPC 信号量

IPC信号量和在第十一章中介绍的内核信号量非常类似：二者都是计数器，用来为多个进程共享的数据结构提供受控访问。如果受保护的资源是可用的，那么信号量的值就是正数；如果受保护的资源现在不能使用，那么信号量的值就是负数或0。想要访问资源的进程把信号量的值加1。只有在原来的值是正数时才可以使用信号量；否则，进程必须等待这个信号量变成正数。当进程释放受保护资源时，就把信号量的值增加1，在这样处理的过程中，其他所有正在等待这个信号量的的进程都必须被唤醒。实际上，由于以下两个主要的原因，IPC信号量比内核信号量的处理更复杂：

- 每个IPC信号量都是一个或者多个信号量值的集合，而不像内核信号量一样只有一个值。这意味着同一个IPC资源可以保护多个独立的共享数据结构。在资源正在被分配的过程中，必须把每个IPC信号量中的信号量的个数指定为semget()函数的一个参数，但是该值不能大于SEMMSL（通常是32）。从现在开始，我们就把信号量内部的计数器称为原始信号量（primitive semaphore）。
- IPC规范创建了一种故障保险的机制，这是针对进程不能取消以前对信号量执行的操作就死亡的情况的。当进程选择使用这种机制时，由此引起的操作就是所谓的可取消的信号量操作。当进程死亡时，如果该进程从来都没有开始执行操作，那么所有的IPC信号量都可以恢复成原来的值。这有助于防止其他使用相同信号量的进程产生死锁。

首先我们简要描绘一下，当进程想访问IPC信号量所保护的一个或者多个资源时所执行的典型步骤。这个进程：

1. 调用semget()封装函数来获得IPC信号量描述符，它把保护共享资源的IPC信号量的IPC关键字作为参数。如果进程希望创建一个新的IPC信号量，就还要指定IPC_CREATE或者IPC_PRIVATE标志以及所需要的原始信号量（请参看本章前面的“使用IPC资源”一节）。
2. 调用semop()封装函数来测试并减少所有原始信号量所涉及的值。如果所有的测试全部成功，就执行减少操作，结束函数并允许这个进程访问受保护的资源。如果有些信号量正在使用，那么进程通常都会被挂起，直到某个其他进程释放这个资源为止。函数接收的参数为IPC信号量标识符、对原始信号量进行

自动操作所指定的一组数以及这种操作的个数。作为选项，该进程也可以定义SEM_UNDO标志，这个标志通知内核撤消那些不释放原始信号量就让进程退出的操作。

3. 当放弃受保护的资源时，就再次调用semop()函数来原子地增加所有有关的原始信号量。

4. 作为选项，调用semctl()封装函数，在参数中指定IPC_RMID标志把这个IPC信号量从系统中删除。

现在我们就可以讨论内核是如何实现IPC信号量的。有关的数据结构如图18-1所示。静态分配的semary数组包括SEMMNI个值（通常是128）。这个数组中的每个元素都可以假设是以下值之一：

- IPC_UNUSED (-1)：没有IPC资源引用这个位置。
- IPC_NOID (-2)：IPC资源正在被分配或者撤消。
- 动态分配给这个IPC信号量资源所在的内存区的地址。

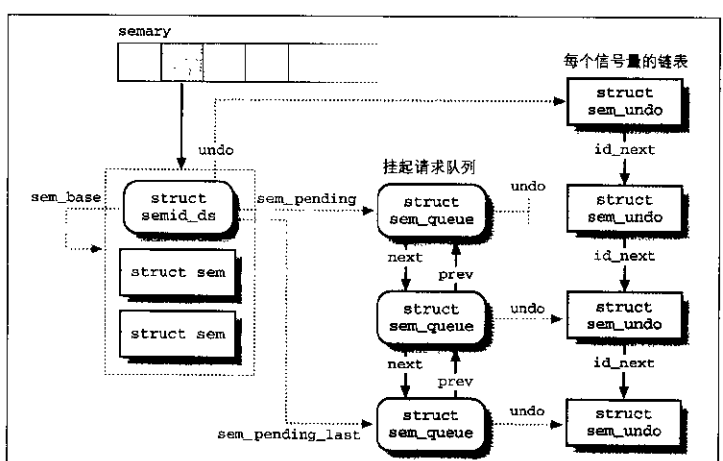


图18-1 IPC信号量数据结构

semary数组的索引号表示前面提到的位置索引*i*。在必须分配新IPC资源时，内核要扫描这个数组并使用包含IPC_UNUSED值的第一个数组元素（位置）。位置索引可以简单地从IPC标识符中通过屏蔽它的高位来获得（请参看前面的“使用IPC资源”一节）。

IPC信号量所在的第一个内存区位置存放了一个semid_ds类型的描述符，其域如表18-8所示。这个内存区中的其他所有位置存放了几个sem数据结构，这个IPC信号量资源中的每个原始信号量各使用一个这种结构。semid_ds结构的sem_base域指向这个内存区的第一个sem结构。sem数据结构只包括两个域：

semval

信号量的计数器的值。

sempid

最后一个访问信号量的进程的PID。进程可以使用semctl()封装函数查询该值。

表18-8 semid_ds结构中的域

类型	域	说明
struct ipc_perm	sem_perm	ipc_perm数据结构
long	sem_otime	最后一个semop()的时间标记
long	sem_ctime	最后一次修改的时间标记
struct sem *	sem_base	指向第一个sem结构的指针
struct sem_queue *	sem_pending	挂起操作
struct sem_queue **	sem_pending_last	最后一次挂起操作
struct sem_undo *	undo	取消请求
unsigned short	sem_nsems	数组中信号量的个数

可取消的信号量操作

如果一个进程突然放弃执行，那么它就不能取消已经开始执行的操作（例如，释放自己保留的信号量）。因此通过把这些操作定义成可取消的（undoable），进程就可以让内核把信号量返回到一致状态并允许其他进程继续执行。进程可以在semop()函数中指定SEM_UNDO标志来要求可取消的操作。

为了有助于内核撤消给定进程对给定的IPC信号量资源所执行的无效操作，有关的信息被存放在一个sem_undo数据结构中。这个结构实际上包含信号量的IPC标识符及一个整数数组，这个数组表示由于进程执行的所有可取消操作而对原始信号量值进行的修改。

有一个简单的例子可以说明如何使用这种sem_undo元素。考虑一个进程使用具有4个原始信号量的一个IPC信号量资源，并假设该进程调用semop()函数把第一个计数器的值增加1并把第二个计数器的值减少2。如果该函数指定了SEM_UNDO标志，sem_undo数据结构中的第一个数组元素中的整数值就被减少1，而第二个元素就被增加2，其他两个整数都保留不变。同一进程对这个IPC信号量执行的更多的可取消操作根据存放在sem_undo结构中的整数而改变。当进程存在时，该数组中的非零值就对应一个或者多个对相应的原始信号量不平衡的操作，内核简单地给相应的原始信号量增加这个非零值来抵消这些操作，换而言之，由异常中断的进程所做的修改被退回，而由其他进程所做的修改仍然能反映信号量的状态。

对于每个进程来说，内核都要记录以可取消操作所处理的所有信号量资源，这样如果进程意外地退出，就可以回滚这些操作。还有，内核还必须对每个信号量都记录它所有的sem_undo结构，这样只要进程使用semctl()来强行给一个原始信号量的计数器赋一个明确的值或者撤消一个IPC信号量资源时，内核就可以快速访问这些结构。

借助于两个链表，我们称之为每个进程的链表和每个信号量的链表，内核就可以有效地处理这些任务。第一个链表记录一个给定进程使用可取消操作处理的所有信号量。第二个链表记录对一个具有可取消操作的给定信号量进行操作的所有进程。更确切地说：

- 每个进程链表包含的所有sem_undo数据结构与IPC信号量相对应，该进程在这些IPC信号量上已经执行了可取消操作。进程描述符的semundo域指向该链表的第一个元素，而每个sem_undo数据结构的proc_next域指向该链表中的下一个元素。
- 每个信号量链表包含的所有sem_undo数据结构与在该信号量上执行可取消操作的进程相对应。semid_ds数据结构的undo域指向这个链表的第一个元素，而每个sem_undo数据结构的id_next域指向该链表中的下一个元素。

当进程结束时，每个进程的链表才被使用。sem_exit()函数由do_exit()调用，后者会遍历这个链表，并为进程所涉及的每个IPC信号量平息非平衡操作产生的影响。与此相对照，当进程调用semctl()函数强行给一个原始信号量赋一个明确的值时，每个信号量的链表才被使用。在sem_undo数据结构引用这个IPC信号量资源时，内核把所有这个结构中的数组的相应元素设置成0，因为再对这个原始信号量执行的上一个可取消操作的影响进行消除已没有任何意义。此外，在IPC信号量被销毁时，每个信号量链表也被使用，通过把semid域设置成-1而使所有有关的sem_undo数据结构都变为无效（注10）。

挂起请求队列

内核给每个IPC信号量都分配了一个挂起请求队列（queue of pending pending reqest），用来标识正在等待数组中的其中一个信号量的进程。这个队列是一个sem_queue数据结构的双向链表，其域如表18-9所示。该队列中的第一个和最后一个挂起请求分别由semid_ds结构中的sem_pending和sem_pending_last域所指向。这最后一个域允许把这个链表作为一个FIFO进行简单的处理，新的挂起请求都被加到这个链表的末尾，这样就可以稍后被服务。挂起请求最重要的域是nsops和sops，前者存放这个挂起操作所涉及的原始信号量的个数，后者指向一个说明每个信号量操作的整型数组。sleeper域存放睡眠进程所在的等待队列的地址。

表18-9 sem_queue结构中的域

类型	域	说明
struct sem_queue *	next	指向下一个队列元素的指针
struct sem_queue **	prev	指向上一个队列元素的指针
struct wait_queue *	sleeper	指向正在睡眠的进程的等待队列的指针
struct sem_undo *	undo	指向sem_undo结构的指针
int	pid	进程描述符
int	status	操作的完成状态
struct semid_ds *	sma	指向IPC信号量的描述符

注10：注意这些数据结构仅仅是无效而已，并没有被释放，因为从所有进程的每个进程链表中删除这些数据结构代价太高了。

表18-9 sem_queue结构中的域（续）

类型	域	说明
struct sembuf *	sops	指向挂起操作的数组的指针
int	nsops	挂起操作的个数
int	alter	改变操作的标志

图18-1显示了一个有三个挂起请求的IPC信号量。其中两个涉及可取消操作，因此sem_queue数据结构的undo域指向相应的sem_undo结构。第三个挂起请求的undo域为NULL，因为相应的操作是不可取消的。

IPC消息

进程彼此之间可以通过IPC消息进行通信。进程产生的每个消息都被发送到一个IPC消息队列（message queue）中，这个消息一直存放在队列中直到另一个进程将其读走为止。

消息是由一个固定大小的首部（header）和一个可变长度的正文（text）组成的；可以使用一个整数值（消息类型）标识消息，这就允许进程有选择地从消息队列中获取消息（注11）。只要进程从IPC消息队列中读出一个消息，内核就把这个消息删除。因此，一个进程只能接收一个给定的消息。

为了发送一个消息，进程要调用msgsnd()函数，传递给它以下参数：

- 目标消息队列的IPC标识符
- 消息正文的大小
- 用户态缓冲区的地址。缓冲区中包含消息类型，之后紧跟的就是消息正文

进程要获得一个消息就要调用msgrcv()函数，传递给它如下参数：

- IPC消息队列资源的IPC标识符

注11：正如我们将看到的一样，消息队列是使用一个链表来实现的。因为消息可以按照非“先进先出”的次序获得，因此“消息队列”这个名字并不恰当。但是，新消息通常都被放到这个链表的末尾。

- 指向用户态缓冲区的指针，消息类型和消息正文应该到被拷贝到这个缓冲区
- 该缓冲区的大小
- 一个值 t，指定应该获得什么消息

如果 t 的值为0，就返回队列中的第一个消息。如果 t 为正数，就返回队列中类型等于 t 的第一个消息。最后，如果 t 为负数，就返回消息类型小于等于 t 绝对值的最小的第一个消息。

与IPC消息队列有关的数据结构如图18-2所示。静态分配的数组msgque包括MSGMNI个值（通常是128）。与semary数组类似，msgque数组中的每个元素都可以采用的值为IPC_UNUSED、IPC_NOID或者一个IPC消息队列描述符的地址。

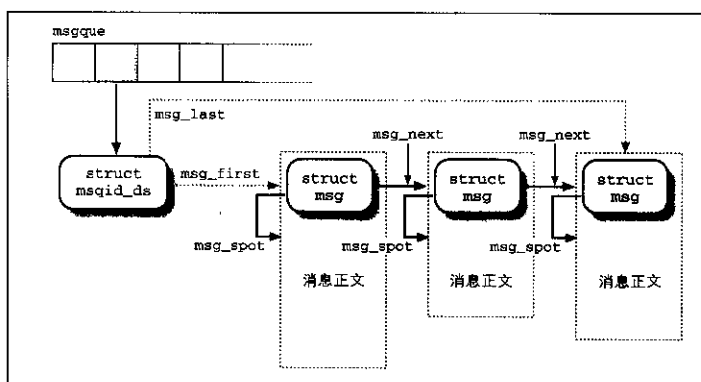


图18-2 IPC消息队列数据结构

这个消息队列描述符是一个msqid_ds结构，其域如表18-10所示。最重要的域是msg_first和msg_last，二者分别指向这个链表的第一个消息和最后一个消息。rwait域指向一个等待队列，这个队列中包括当前正在等待消息队列中某个消息的所有进程。反之，wwait域则指向另一个等待队列，该队列包括当前正在等待消息队列中有空闲空间的所有进程，以便这些进程可以向消息队列中加入一个新消息。该队列中的所有消息的消息首部和消息正文的总大小不能超过存放在msg_qbytes域中的值。缺省的最大值是MSGMNB，也就是16,384字节。

表18-10 msqid_ds结构

类型	域	说明
struct ipc_perm	msg_perm	Ipc_perm数据结构
struct msg *	msg_first	队列中的第一个消息
struct msg *	msg_last	队列中的最后一个消息
long	msg_stime	最后一次调用msgsnd()的时间
long	msg_rtime	最后一次调用msgrcv()的时间
long	msg_ctime	最后一次修改的时间
struct wait_queue *	wwait	正在等待空闲空间的进程
struct wait_queue *	rwait	正在等待消息的进程
unsigned short	msg_cbytes	队列中的当前字节数
unsigned short	msg_qnum	队列中的消息个数
unsigned short	msg_qbytes	队列中的字节的最大个数
unsigned short	msg_lspid	最后一次调用msgsnd()的PID
unsigned short	msg_lrpid	最后一次调用msgrcv()的PID

每个消息都被放在一个动态分配的内存区中。这个内存区的开头存放了消息首部，消息首部是一个msg类型的数据结构，其域如表18-11所示。消息正文存放在这个内存区的剩余空间中。消息首部的msg_spot域包含消息正文的起始地址，而msg_ts域包含消息正文的长度，这个长度不能超过MSGMAX（通常是4056）字节。

表18-11 msg结构

类型	域	说明
struct msg *	msg_next	队列中的下一个元素
Long	msg_type	消息类型
char *	msg_spot	消息正文地址
time_t	msg_stime	调用msgsnd()的时间
short	msg_ts	消息正文的大小

最后，每个消息都通过自己消息首部的msg_next域与消息队列中的下一个消息相链接。

IPC 共享内存

最有用的IPC机制是共享内存，这种机制允许两个或多个进程通过把公共数据结构放入一个共享内存段（chared memory segment）来访问它们。如果进程要访问这种数据结构所在的共享内存段，就必须在自己的地址空间中增加一个线性区（请参看第七章中的“线性区”一节），这个线性区映射了与这个共享内存段相关的页框。这样的页框可以由内核通过请求调页进行简单的处理（请参看第七章中的“请求调页”一节）。

与信号量以及消息队列一样，系统调用shmget()函数来获得一个共享内存段的IPC标识符，如果这个共享内存段不存在，就创建它。

shmat()函数调用用来把一个共享内存段“附加（attach）”到一个进程上。该函数使用这个IPC共享内存资源的标识符作为参数，并试图把一个共享内存区加入到调用进程的地址空间中。调用进程可以获得这个线性区的起始线性地址，但是这个地址通常并不重要，访问这个共享内存段的每个进程都可以使用自己地址空间中的不同地址。shmat()函数不修改进程的页表。我们稍后会介绍在进程试图访问属于这个新线性区的页时内核究竟怎样进行处理。

shmdt()函数被调用来“分离（detach）”由IPC标识符所指定的共享内存段，也就是说把相应的共享内存区从进程的地址空间中删除。回想一下IPC共享内存资源是持久的，即使现在没有进程在使用它，相应的页也不能被丢弃，但是可以被换出。

图18-3说明了实现IPC共享内存所使用的主要数据结构。静态分配的数组shm_segs包括SHMMNI个值（通常是128）。和semary以及msgque数组类似，shm_segs中的每个元素的值都可能为IPC_UNUSED、IPC_NOID或者一个IPC共享内存段描述符的地址。

每个IPC共享内存段描述符都是一个shmid_kernel结构，其域如表18-12所示。用户态进程可以访问的域都包含在这个描述符内部一个名为u的shmid_ds数据结构中。其内容可以通过shmctl()函数进行访问。

u.shm_segsz和shm_npages域分别存放了这个共享内存段的以字节为单位的大小和以页为单位的大小。虽然用户态进程可能需要任意长度的共享内存段，但是所分配的内存段的长度只能是页大小的整数倍，因为内核必须用一个线性区来映射这个内存段。

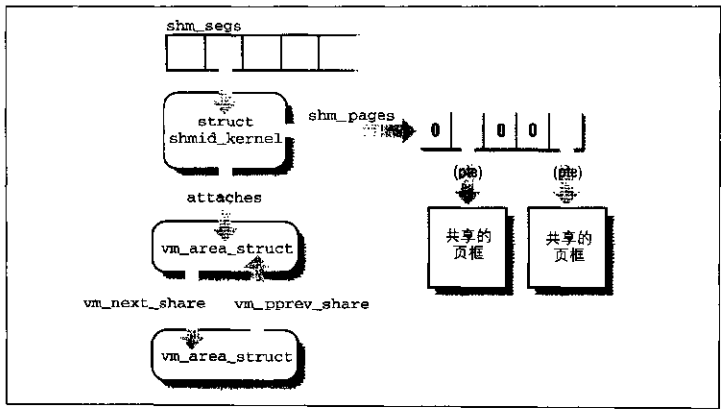


图 18-3 IPC 共享内存数据结构

shm_pages域指向一个数组，这个数组中的一个元素对应这个内存段中的一个页。每个元素都以页表项的格式存放了一个32位的值（请参看第二章中的“常规分页”一节）。如果页框现在还没有分配给这个页，那么该元素就是0。否则，它就是一个正规页表项，其中包含一个页框或一个换出页描述符的物理地址。

表 18-12 shmid_kernel结构中的域

类型	域	说明
struct ipc_perm	u.shm_perm	ipc_perm数据结构
int	u.shm_segsz	共享内存区的大小（以字节为单位）
long	u.shm_atime	最后一次附加线性区的时间
long	u.shm_dtime	最后一次分离线性区的时间
long	u.shm_ctime	最后一次修改的时间
unsigned short	u.shm_cpid	创建者的PID
unsigned short	u.shm_lpid	最后一次访问共享内存段的进程的PID
unsigned short	u.shm_nattch	当前附加线性区的个数
unsigned long	shm_npages	共享内存区的大小（以页为单位）

表18-12 shmid_kernel结构中的域（续）

类型	域	说明
unsigned long *	shm_pages	指向页框PTE的数组的指针
struct vm_area_struct *	attaches	指向VMA描述符链表的指针

在图18-3所说明的例子中，共享内存段包括5个页。其中三个还没有被访问，而其他两个页都存放在RAM中。

与共享内存段相关的所有线性区的vm_area_struct描述符形成一个双向链表，attaches域就指向该链表的第一个元素。该链表是通过描述符的vm_next_share和vm_pprev_share域实现的。链表中元素的个数存放在u.shm_nattch域中。在图18-3中，这个共享内存段已经被附加到两个进程的地址空间中。

在对IPC共享内存段进行映射时，vm_area_struct描述符的一些域有特殊的意义：

vm_start和vm_end
　限定线性区的线性地址的范围

vm_pte
　把共享内存段的索引存放在shm_segs数组中

vm_ops
　指向一个名为shm_vm_ops的线性区操作表

为IPC共享内存段请求调页

通过shmat()加入进程的页都是哑元页（dummy page）；该函数把一个新线性区加入一个进程的地址空间中，但是它不修改该进程的页表。我们现在就可以解释这些页如何变成可用的。

因为shmat()函数不修改页表，“缺页”就发生在进程试图访问共享内存段的一个单元时。相应的异常处理程序确定这个缺少的地址是在进程地址空间的内部，而且相应的页表项为空。因此，它就会调用do_no_page()函数（请参看第七章中的“请求调页”一节）。接下来，这个函数又要检查是否为这个线性区定义了nopage方法。

然后调用这个方法，并把页表项设置成所返回的地址（请参看第十五章中的“对内存映射进行请求调页”一节）。

IPC共享内存所使用的线性区通常都定义了nopage方法。这是通过shm_nopage()函数实现的，该函数执行以下操作：

1. 从线性区描述符的vm_pte域中提取出这个共享内存段在shm_segs数组中所对应的索引。
2. 从线性区描述符的vm_start域和所请求的地址中计算出在该段内的逻辑页号。
3. 访问由该段的shmid_kernel描述符的shm_pages域所引用的数组，并获得与这个缺少的地址所在的页相对应的页表项。根据这个表项的值考虑三种情况：
   - 空项：还没有给该页分配页框。在这种情况下，分配一个新页框并将其页表项存放在shm_pages数组中。
   - 有Present标志的正规项：该页已经存放在某个页框中。从shm_pages的相应项中提取出它的物理地址。
   - 换出页标识符：该页已经被交换到磁盘上。分配一个新的页框，从磁盘中读出页，将其拷贝到页框中，并把新的页表项存放在shm_pages数组中。实际上，这种情况下所执行的操作和共享内存段中所包含页的换入过程是一致的（在后面介绍）。
4. 增加在上一步中所分配或所标识的页框的引用计数器的值。
5. 返回这个页框的物理地址。

do_no_page()函数设置所缺少的地址在进程的页表中所对应的表项，这样函数就指向该方法所返回的页框。

IPC共享内存段的换出页

内核在把共享内存段中包含的页换出时要谨慎。假设有两个进程P1和P2正在访问一个共享内存段中的页。再假设swap_out()函数试图释放一个页框，这个页框分配给了进程P1，进程P1和进程P2共享该页（请参看第十六章中的“页换出”一节）。根据标准的换出规则，这个共享页应该被拷贝到磁盘中然后再释放，换出页标识符应该被写入P1对应的页表项中。但是，这个标准的过程不能正常工作，因为进程P2

可能通过自己的页表试图访问该页面相应的页表项仍然指向已释放的页框，所以各种数据崩溃的情况都可能发生。

try_to_swap_out()函数（请参看第十六章中的“try_to_swap_out()函数”一节）通过检查这个线性区是否包括一个swapout方法来识别这种特例。如果定义了这个方法，那么这个页框就不会释放给Buddy系统，其引用计数器只是简单地被减少1，P1页表中的相应项被清空。shm_vm_ops中的swapout方法是一个空函数，因为这个方法必须是非空的，只有这样才能让内核知道这个内存区是共享的，因此它必须指向某个函数，即使这个函数什么都不做。P2可以安全地访问这个页框，因为它依然包括这个IPC共享内存的页。

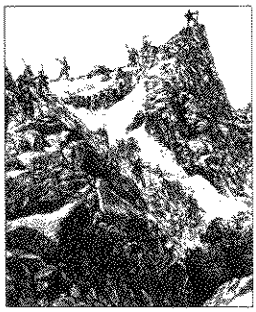
共享内存段和任何IPC资源一样，都是持久性资源。这意味着进程不再使用的共享内存段的页框仍然由shm_pages数组访问。这些页框可以由shm_swap()函数交换到磁盘上，该函数周期性地调用do_try_to_free_pages()函数（请参看第十六章中的“try_to_free_pages()函数”一节）。后者反复地扫描shm_segs数组所引用的所有描述符，并为每个描述符检查分配给每个段的所有页框。如果确定某个页框的引用计数器的值等于1，就可以把相应的页安全地交换到磁盘上。所用的换出过程与非共享页使用的过程类似，唯一不同的是换出页标识符是存放在shm_pages数组中的。

总而言之，尽管共享内存页是在进程的页表中，但交换机制对这些页的处理与其他页并不相同。这种页的换出页标识符不出现在页表项中，而出现在shm_pages数组中。当进程试图寻址一个换出的页时，这个空页表项就会触发一个“缺页”异常。内核接收在shm_pages数组中的换出页标识符，并执行换入过程。

对 Linux 2.4 的展望

用来表示信号量和消息的静态数组已经被删除，取而代之的是动态数据结构。现在可以处理更多的IPC消息。

IPC共享内存区以不同的方法来实现：引入一种新的*/proc*文件系统，表示为*sysvipc*。现在只包括一个名为*shm*的目录，其中为每个IPC共享内存区都包含一个虚拟文件。



第十九章
程序的执行

第三章所描述的“进程”概念在Unix中是用来表示正在运行的一组程序竞争系统资源的行为。作为最后的一章，本章将集中讨论程序和进程之间的关系。我们会专门描述内核是如何通过程序文件的内容建立进程的执行上下文。尽管把一组指令装入内存并让CPU执行看起来并不是什么大问题，但内核还必须灵活处理以下几方面的问题：

不同的可执行文件格式

Linux的一个著名之处就是能执行其他操作系统所编译的二进制文件。

共享库

很多可执行文件并不包含运行程序所需要的所有代码，而是希望在运行时由内核从函数库装入函数。

执行上下文中的其他信息

包括程序员熟悉的命令行参数与环境变量。

程序是以可执行文件的形式存放在磁盘上的，可执行文件既包括被执行函数的目标代码也包括这些函数所使用的数据。程序中的很多函数是所有程序员都可使用的服务例程，它们的目标代码被包含在所谓“库”的特殊文件中。实际上，一个库函数的代码或被静态地拷贝到可执行文件中（静态库），或在运行时被连接到进程（共享库，因为它们的代码由很多独立的进程所共享）。

当装入并运行一个程序时，用户可以提供影响程序执行方式的两种信息：命令行参数和环境变量。用户在shell提示符下紧跟文件名输入的就是命令行参数。环境变量（例如HOME和PATH）是从shell继承来的，但用户在装入并运行程序前可以修改任何环境变量。

我们在“可执行文件”一节将解释一个程序的执行上下文到底是什么。在“可执行格式”一节我们会提及一些Linux所支持的可执行格式，并显示Linux如何改变它的“个性”以执行其他操作系统所编译的程序。最后，在“exec类函数”一节会描述执行一个新程序的进程所需的系统调用。

可执行文件

在第一章中把进程定义为一个“执行上下文”。这就意味着进行特定的计算需要收集必要的信息，包括所访问的页，打开的文件，硬件寄存器的内容等等。可执行文件（executable file）是一个正规文件，它描述了如何初始化一个新的执行上下文，也就是如何开始一个新的计算。

假定一位用户想在当前目录下显示文件，他知道在shell提示符下只要简单地敲入外部命令——*/bin/ls*（注1）就可得到这个结果。命令shell创建一个新进程，新进程又调用系统调用execve()（参看本章后面的“exec类函数”一节），其中传递的一个参数就是*ls*可执行文件的全路径名，在本例中即*/bin/ls*。sys_execve()服务例程找到相应的文件，检查可执行格式，并根据存放在其中的信息修改当前进程的执行上下文。因此，当这个系统调用终止时，新进程开始执行存放在可执行文件中的代码，也就是执行目录显示。

当进程开始执行一个新程序时，它的执行上下文发生很大的变化，这是因为在进程的前一个计算执行期间所获得的大部分资源会被抛弃。在前面的例子中，当进程开始执行*/bin/ls*时，它用execve()系统调用传递来的新参数代替shell的参数，并获得一个新的shell环境（参见后面的“命令行参数和shell环境”一节）；从父进程继承的所有页（与写时复制机制共享）被释放，以便在一个新的用户态地址空间开始

注1：在Linux中，可执行文件的路径不是固定的，这依赖于所使用的发布版本。对于所有的Unix系统，已经提议了几个标准的命名模式（如FHS和FSSTND）。

执行新的计算；甚至进程的特权都可能改变（参看后面的“进程的信任状和能力”一节）。然而，进程的PID不改变，并且新的计算从前一个计算继承所有打开的文件描述符，当然这些文件描述符是在执行execve()系统调用时还没有自动关闭的描述符（注2）。

进程的信任状和能力

从传统上看，Unix系统与每个进程的一些信任状（credential）相关，信任状把进程与一个特定的用户或用户组捆绑在一起。信任状在多用户系统上尤为重要，因为信任状可以决定每个进程能做什么，不能做什么，这样既保证了每个用户个人数据的完整性，也保证了系统整体上的稳定性。

信任状的使用既需要在进程的数据结构方面给予支持，也需要在被保护的资源方面给予支持。文件就是一种显而易见的资源。因此，在Ext2文件系统中，每个文件都属于一个特定的用户，并被捆绑于某个用户组。文件的拥有者可以决定对某个文件允许哪些操作，以在文件的拥有者、文件的用户组及其他所有用户之间做出区别。当某个进程试图访问一个文件时，VFS总是根据文件的拥有者和进程的信任状所建立的许可权检查访问的合法性。

进程的信任状被存放在进程描述符的几个域中，如表19-1所示。这些域包括系统中用户和用户组的标识符，与之可以相比较的通常是存放在被访问文件索引节点中的标识符。

表19-1 传统的进程信任状

名字	说明
uid, gid	用户和组的实际描述符
euid, egid	用户和组的有效描述符
fsuid, fsgid	文件访问的用户和组的有效描述符

注2：缺省情况下，在发生execve()系统调用之后，已被进程打开的文件将保持打开状态。然而，如果该进程设置了files_struct结构的close_on_exec域的相应位，那么该文件将自动被关闭（参见第十二章中的表12-6）。此任务由fcntl()系统调用完成。

表 19-1 传统的进程信任状（续）

名字	说明
groups	补充的组描述符
suid, sgid	用户和组保存的描述符

一个空的 UID 指定给 root 超级用户，而空的 GID 指定给 root 超级组。只要有关进程的信任状存放了一个空值，内核总是允许这个进程做任何事情。因此，也可以用进程信任状来检查与文件无关的操作，如涉及系统管理或硬件处理的那些操作，如果在某个进程的信任状中所存放的 UID 为空，则操作被允许；否则被禁止。

当一个进程被创建时，总是继承父进程的信任状。不过，这些信任状以后可以被修改，这发生在当进程开始执行一个新程序时，或者当进程发布合适的系统调用时。通常情况下，进程的 uid、euid、fsuid 及 suid 域具有相同的值。然而，当进程执行 *setuid* 程序时，即可执行文件的 *setuid* 标志被设置时，euid 和 fsuid 域被置为这个文件拥有者的标识符。几乎所有的检查都涉及这两个域中的一个：fsuid 用于与文件相关的操作，而 euid 用于其他所有的操作。这也同样适用于组描述符的 uid、euid、fsuid 及 suid 域。

我们用一个例子来说明如何使用 fsuid 域，考虑一下当用户想改变她的口令时的普通情况。所有的口令都存放在一个公共文件中，但用户不能直接编辑这样的文件，因为它是受保护的。因此，用户调用一个名叫 */usr/bin/passwd* 的系统程序，它可以设置 *setuid* 标志，而且它的拥有者是超级用户。当 shell 创建的进程执行这样一个程序时，进程的 euid 和 fsuid 域被置为 0，即超级用户的 PID。现在，这个进程可以访问这个文件，因为当内核执行访问控制表时在 fsuid 域发现了值 0。当然，*/usr/bin/passwd* 程序除了让用户改变自己的口令外，并不允许做其他任何事情。

从 Unix 的历史发展可以得出一个教训，即 *setuid* 程序是相当危险的：恶意用户可以以这样的方式触发代码中的一些编程错误（bug），从而强迫 *setuid* 程序执行程序的最初设计者从未安排的操作。这可能常常危及整个系统的安全。为了减少这样的风险，Linux 与所有现代 Unix 操作系统一样，让进程只有在必要时才获得 setuid 特权，并在不需要时删除它们。当以几个保护级别实现用户应用程序时，可以证明这种特点是很有用的。进程描述符包含一个 suid 域，在 setuid 程序执行以后在该域

中正好存放有效标识符（euid和fsuid）的值。进程可以通过setuid()、setresuid()、setfsuid()和setreuid()系统调用改变有效标识符（注3）。

表19-2显示了这些系统调用是怎样影响进程的信任状的。请注意，如果调用进程还没有超级用户特权，即它的euid域不为空，那么，只能用这些系统调用来设置在这个进程的信任状域已经有的值。例如，一个普通用户进程可以通过调用系统调用setfsuid()强追它的fsuid值为500，但这只有在其他信任状域中有一个域已经有相同的值500时才行。

表19-2 设置进程信任状的系统调用语义

	setuid (e)		setresuid (u,e,s)	setreuid (u,e)	setfsuid (f)
	euid=0	euid 0			
uid	设置为e	不改变	设置为u	设置为u	不改变
euid	设置为e	设置为e	设置为e	设置为e	不改变
fsuid	设置为e	设置为e	设置为e	设置为e	设置为f
suid	设置为e	不改变	设置为s	设置为e	不改变

为了理解四个用户ID域之间的复杂关系，让我们考虑一下setuid()系统调用的效果。这些操作是不同的，这依赖于调用者进程的euid域是否被置为0（即进程有超级用户特权）或被置为一个正常的UID。

如果euid域为空，这个系统调用就把调用进程的所有信任状域（uid、euid、fsuid及suid）置为参数e的值。一个超级用户进程因此就可以删除自己的特权而变为由普通用户拥有的一个进程。例如，在用户登录时，系统以超级用户特权创建一个新进程，但这个进程通过调用setuid()系统调用删除自己的特权，然后开始执行用户的login shell程序。

如果euid域不为空，这个系统调用只修改存放在euid和fsuid中的值，让其他两个域保持不变。这就允许进程执行*setuid*程序，以让存放在euid和fsuid中的有

注3：通过发出相应的setgid()，setresgid()，setfsgid()和setregid()系统调用可以改变GID的有效信任状。

效特权交替地置为uid（进程起的作用是装入并执行可执行文件）和suid（进程起的作用是拥有可执行文件）。

进程的能力

Linux用能力（capability）表示另一个进程信任状模型。一种能力仅仅是一个标志，它表明是否允许进程执行一个特定的操作或一组特定的操作。这个模型不同于传统的“超级用户对普通用户”模型，在后一种模型中，一个进程要么能做任何事情，要么什么也不能做，这取决于它的有效UID。如表19-3所示，在Linux内核中已包含了很多能力。

表19-3 Linux的能力

名字	说明
CAP_CHOWN	忽略对文件和组的拥有者进行改变的限制
CAP_DAC_OVERRIDE	忽略文件的访问许可权
CAP_DAC_READ_SEARCH	忽略文件/目录读和搜索的许可权
CAP_FOWNER	忽略对文件拥有者的限制
CAP_FSETID	忽略对*setid*和*setgid*标志的限制
CAP_KILL	忽略对信号挂起的限制
CAP_SETGID	允许*setgid*标志的操作
CAP_SETUID	允许*setuid*标志的操作
CAP_SETPCAP	转移/删除对其他进程所许可的能力
CAP_LINUX_IMMUTABLE	允许对仅追加和不可变文件的修改
CAP_NET_BIND_SERVICE	允许绑定到低于1024 TCP/UDP的套接字
CAP_NET_BROADCAST	允许网络广播和监听多点传送
CAP_NET_ADMIN	允许一般的网络管理
CAP_NET_RAW	允许使用RAW和PACKET套接字
CAP_IPC_LOCK	允许页和共享内存的加锁
CAP_IPC_OWNER	跳过IPC拥有者的检查
CAP_SYS_MODULE	允许内核模块的插入和删除
CAP_SYS_RAWIO	允许通过ioperm()和iopl()访问I/O端口

表 19-3 Linux 的能力（续）

名字	说明
CAP_SYS_CHROOT	允许使用 chroot()
CAP_SYS_PTRACE	允许在任何进程上使用 ptrace()
CAP_SYS_PACCT	允许配置进程的计账
CAP_SYS_ADMIN	允许一般的系统管理
CAP_SYS_BOOT	允许使用 reboot()
CAP_SYS_NICE	忽略对 nice() 的限制
CAP_SYS_RESOURCE	忽略对几个资源使用的限制
CAP_SYS_TIME	允许系统时钟和实时时钟的操作
CAP_SYS_TTY_CONFIG	允许配置 tty 设备

任何时候，每个进程只需要有限种能力，这是其主要优势。因此，即使一位有恶意的用户发现了使用有潜在错误程序的方法，他也只能非法地执行有限个操作类型。

例如，假定一个有潜在错误的程序只有CAP_SYS_TIME能力。在这种情况下，利用其错误的恶意用户只能在非法地改变实时时钟和系统时钟方面获得成功。他并不能执行任何其他特权的操作。

进程可以分别用 capget() 和 capset() 系统调用显示地获得和设置它的能力。但是，不管是 VFS 还是 Ext2 文件系统目前都不支持这种能力模型，所以，当进程执行一个可执行文件时，没有办法把这个文件与本该强加的一组能力联系起来。对 Linux 2.2的终端用户来说，能力没什么用处，不过，我们还是能预测这种状况将很快得到改变。

事实上，Linux 内核已经包括一些能力方面的设置。例如，让我们考虑一下允许用户改变进程静态优先级的 nice() 系统调用。在传统的模型中，只有超级用户才能提升一个优先级：内核因此应该检查调用进程描述符的 euid 域是否为 0。然而，Linux 内核定义了一个叫CAP_SYS_NICE的能力，就正好对应着这种操作。内核通过调用 capable() 函数并把 CAP_SYS_NICE 值传给这个函数来检查这个标志的值。

正是由于一些“兼容性小巧程序”已被加入到内核代码中，这种方法才起作用。每当一个进程把 euid 和 fsuid 域设置为 0 时（或者通过调用表 19-2 中的一个系统调

用，或者通过执行超级用户所拥有的*setuid*程序），内核就设置进程的所有能力，以便所有的检查成功。类似地，当进程把euid和fsuid域重新置为进程拥有者的实际UID时，内核删除所有的能力。

命令行参数和shell环境

当用户敲入一个命令时，为满足这个请求而装入的程序可以从shell接受一些命令行参数（command-line argument）。例如，当用户敲入命令：

```
$ ls -l /usr/bin
```

以获得在*/usr/bin*目录下的全部文件列表时，shell进程创建一个新进程执行这个命令。这个新进程装入*/bin/ls*可执行文件。在这样做的过程中，从shell继承的大多数执行上下文被丢弃，但三个单独的参数ls、-l和/usr/依然被保持。一般情况下，新进程可以接受任意个参数。

传递命令行参数的约定依赖于所用的高级语言。在C语言中，程序的main()函数把传递给程序的参数个数和指向字符串指针数组的地址作为参数。下列的原型以形式化表示这种标准：

```
int main(int argc, char *argv[])
```

再回到前面的例子，当*/bin/ls*程序被调用时，argc的值为3，argv[0]指向ls字符串，argv[1]指向-l字符串，而argv[2]指向*/usr/bin*字符串。argv数组的末尾处总以空指针来标记，因此，argv[3]为NULL。

在C语言中传递给main()函数的第三个可选参数是包含环境变量（environment variable）的参数。当进程用到它时，main()的声明如下：

```
int main(int argc, char *argv[], char *envp[])
```

envp参数指向环境串的指针数组，形式如下：

```
VAR_NAME=something
```

在这里，VAR_NAME表示一个环境变量的名字，而“=”后面的子串表示赋给变量的实际值。envp数组的结尾用一个空指针标记，就像argv数组。环境变量用来定制

进程的执行上下文，为用户或其他进程提供一般的信息，或允许进程交叉用execve()系统调用保存一些信息。

命令行参数和环境串都被放在用户态堆栈，正好在返回地址之前（参看第八章中的“参数传递”一节）。图19-1显示了用户态堆栈的底部位置。注意环境变量位于栈底附近正好在一个null的长整数之后。

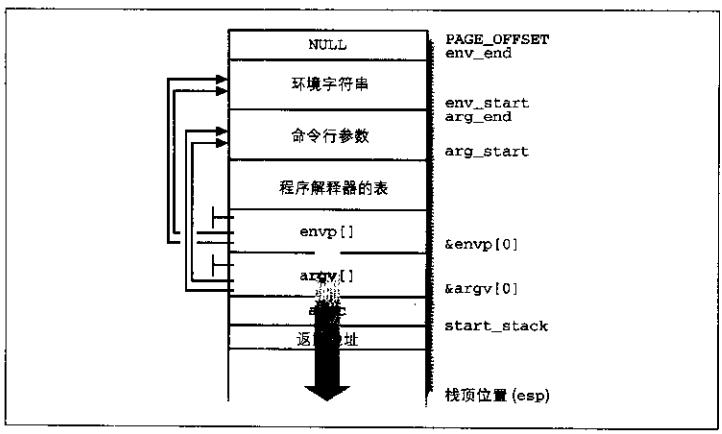


图19-1 用户态堆栈的栈底位置

库

每个高级语言的源代码文件都是经过几个步骤才转化为目标文件（object file）的，目标文件中包含的是汇编语言指令的机器代码，它们和相应的高级语言指令对应。目标文件并不能被执行，因为它不包含源代码文件（例如库函数或同一程序中的其他源代码文件）所用的全局外部符号名的线性地址。这些地址的分配或解析是由链接程序完成的，链接程序把程序所有的目标文件收集起来并构造可执行文件。链接程序还分析程序所用的库函数并以本章后面所描述的方式把它们粘合成可执行文件。

任何程序，甚至最小的程序都会利用C库。请看下面的一行C程序：

```
void main(void) { }
```

尽管这个程序没有做任何事情，但还是需要做很多工作来建立执行环境（参见本章后面的“exec类函数”一节）并在程序终止时杀死这个进程（参看第三章中的“撤消进程”一节）。尤其是当main()函数终止时，C编译程序把exit()系统调用插入到目标代码中。

从第八章我们知道，程序通常通过C库中的封装例程调用系统调用。C编译器亦如此。任何可执行文件除了包括对程序的语句进行编译所直接产生的代码外，还包括一些“粘合”代码来处理用户态进程与内核之间的交互。这样的粘合代码有一部分存放在C库中。

除了C库，Unix系统中还包含很多其他的函数库。一般的Linux系统可能轻而易举地就有50个不同的库。这里仅仅列举其中的两个：数学库 *libm* 包含浮点操作的基本函数，而X11库 *libX11* 收集了所有X11窗口系统图形接口的基本底层函数。

传统Unix系统中的所有可执行文件都是基于静态库的。这就意味着链接程序所产生的可执行文件不仅包括原程序的代码，还包括程序所引用的库函数的代码。

静态库的一大缺点是：它们占用大量的磁盘空间。的确，每个静态链接的可执行文件都复制库代码的一部分。

因此，现代Unix系统利用了共享库（shared library）。可执行文件不用再包含库的目标代码，而仅仅指向库名。当程序被装入内存执行时，一个叫做程序解释器（program interpreter）的程序就专注于分析可执行文件中的库名，确定所需库在系统目录树中的位置，并使执行进程可以使用所请求的代码。

共享库对提供文件内存映射的系统尤为方便，因为它们减少了执行一个程序所需的主内存量。当程序解释器必须把某一共享库链接到进程时，并不拷贝目标代码，而是仅仅执行一个内存映射，把库文件的相关部分映射到进程的地址空间中。这就允许共享库机器代码所在的页框被使用相同代码的所有进程共享。

共享库也有一些缺点。动态链接的程序启动时间通常比静态链接的程序长。此外，动态链接的程序的可移植性也不如静态链接的好，因为当系统中所包含的库版本发生变化时，动态链接的程序可能就不能适当地执行。

用户可以请求一个程序被静态地链接。例如，GCC编译器提供-static选项，即告诉链接程序使用静态库而不是共享库。

程序段和进程的线性区

从逻辑上说，Unix程序的线性地址空间传统上被划分为几个叫做段（segment）的区间（注4）：

正文段

包含可执行代码

数据段

包含初始化的数据，也就是说，初值存放在可执行文件中的所有静态变量和全局变量（因为程序在启动时必须知道它们的值）

bss段

包含未初始化的数据，也就是说，初值没有存放在可执行文件中的所有全局变量（因为程序在引用它们之前才赋值）

堆栈段

包含程序的堆栈，堆栈中有返回地址、参数和被执行函数的局部变量

每个mm_struct描述符（参见第七章中的“内存描述符”一节）都包含一些域来标识相应进程某一特定线性区的作用：

`start_code`, `end_code`

程序的源代码所在线性区的起始和终止线性地址，即可执行文件中的代码。因为正文段包含共享库，但可执行文件不包含，因此，由这些域所划分的线性区是正文段的子集。

`start_data`, `end_data`

程序的原初始化数据所在线性区的起始和终止线性地址，正如在可执行文件中所指定的那样。这两个域指定的线性区大体上与数据段对应。事实上，`start_data`就是`end_code`之后第一页的地址，因此用不着这个域，但要用`end_data`域。

注4：之所以称为“段”是因为第一代Unix系统使用一种段寄存器实现了线性地址区间。但是，Linux并不依赖这种Intel微处理器的分段机制来实现程序分段。

start_brk, brk

存放线性区的起始和终止线性地址，该线性区包含动态分配给进程的内存区（参看第七章的“堆的管理”一节）。有时把这部分线性区叫做堆。

start_stack

正好在main()的返回地址之上的地址。如图19-1所示，更高的地址被保留（回想一下栈是向低地址增长）。

arg_start, arg_end

命令行参数所在的堆栈部分的起始地址和终止地址。

env_start, env_end

环境串所在的堆栈部分的起始地址和终止地址。

注意，共享库和文件的内存映射使得基于程序段的进程地址空间分类有点过时，因为每个共享库被映射到与前面所讨论的线性区不同的线性区。

现在，我们通过一个简单的例子来描述Linux内核是如何把共享库映射到进程的地址空间。我们照样假设用户态进程的地址空间范围从0x00000000到0xbfffffff。我们考虑一下*/sbin/init*程序，它创建和监视在操作系统外层实现的所有进程的活动（参看第三章的“内核线程”一节）。init进程对应的线性区如表19-4所示（可以从*/proc/1/maps*文件得到这样的信息）。注意，这里列出的所有区域都是依靠私有内存映射实现的（在表19-4中许可权一栏出现的字母p）。这并不令人惊讶：这些线性区的存在仅仅是为了给进程提供数据，当进程执行指令时，可以修改这些线性区的内容，但与它们相关的磁盘上的文件保持不变。这就是私有内存映射起作用的原因。

表19-4 init进程的线性区

地址范围	许可权	映射的文件
0x08048000-0x0804cfff	r-xp	*/sbin/init* 在偏移量 0 处
0x0804d000-0x0804dfff	rw-p	*/sbin/init* 在偏移量 0x4000 处
0x0804e000-0x0804efff	rwxp	匿名
0x40000000-0x40005fff	r-xp	*/lib/ld-linux.so.1.9.9* 在偏移量 0 处
0x40006000-0x40006fff	rw-p	*/lib/ld-linux.so.1.9.9*在偏移量 0x5000处
0x40007000-0x40007fff	rw-p	匿名

表 19-4 init 进程的线性区（续）

地址范围	许可权	映射的文件
0x4000b000-0x40092fff	r-xp	*/lib/libc.so.5.4.46* 在偏移量 0 处
0x40093000-0x40098fff	rw-p	*/lib/libc.so.5.4.46* 在偏移量 0x87000 处
0x40099000-0x400cafff	rw-p	匿名
0xbfffd000-0xbfffffff	rwxp	匿名

从 0x8048000 开始的线性区是与 */sbin/init* 文件相关的某一部分的内存映射，范围从 0 到 20479 字节（在 */proc/1/maps* 文件中只列出这个区域的起始和终止地址，但很容易导出区域的大小）。许可权指定这个区域是可执行的（包含目标代码）、只读的（不可写，因为指令执行期间不能改变）并且是私有的，因此我们可以猜出这个区域映射了程序的正文段。

从 0x804d000 开始的线性区是与 */sbin/init* 相关的另一部分的内存映射，范围从 6384（对应于表 19-4 显示的偏移量 0x4000）到 20479。因为许可权指定这个私有区域可以被写，我们可以推断出它映射了程序的数据段。

从 0x0804e000 开始的下一页线性区是匿名的，也就是说，它与任何文件都无关。它或许与 *init* 的 bss 段相关。

类似地，从 0x40000000、0x40006000 及 0x40007000 开始的线性区分别对应 */lib/ld-linux.so.1.9.9* 程序（实际上是 ELF 共享库的程序解释器）的正文段、数据段和 bss 段。这个程序解释器从不单独执行，它总是处于一个进程执行另一个程序而形成的地址空间内的内存映射中。

在这个系统中，C 库就存放在 */lib/libc.so.5.4.46* 文件中。C 库的正文段、数据段及 bss 段被映射到从地址 0x4000b000 开始的接着三个段中。请记住，只要私有区域所包含的页框不被修改，就可以由几个进程以写时复制机制来共享这些页框。因为正文段是只读的，因此，C 库可执行代码所在的页框几乎由所有当前正在执行的进程所共享（除了静态链接的所有进程）。

最后，从 0xbfffd000 到 0xbfffffff 的最后一个匿名线性区与用户态堆栈相关。我们已在第七章中的“缺页异常处理程序”一节解释了在必要时堆栈是如何自动朝低地址方向扩展。

执行跟踪

执行跟踪（execution tracing）是一个程序监视另一个程序执行的一种技术。被跟踪的程序一步一步地执行，直到接受到一个信号或调用一个系统调用。执行跟踪由调试程序（debugger）广泛使用，当然还使用其他技术（包括在被调试程序中插入断点及运行时访问它的变量）。与往常一样，我们将集中讨论内核怎样支持执行跟踪而不讨论调试程序怎样工作。

在Linux中，通过ptrace()系统调用进行执行跟踪，这个系统调用能处理表19-5所列的命令。设置了CAP_SYS_PTRACE能力的进程可以跟踪系统中的任何进程（除了init）。相反，没有CAP_SYS_PTRACE能力的进程*P*只能跟踪与*P*有相同属主的进程。此外，两个进程不能同时跟踪一个进程。

表19-5 ptrace命令

命令	说明
PTRACE_TRACEME	对当前进程开始执行跟踪
PTRACE_ATTACH	对另一个进程开始执行跟踪
PTRACE_DETACH	终止执行跟踪
PTRACE_KILL	杀死被跟踪的程序
PTRACE_CONT	重新恢复执行
PTRACE_SYSCALL	重新恢复执行直到下一个系统调用的边界
PTRACE_SINGLESTEP	对一条单独的汇编指令重新恢复执行
PTRACE_PEEKTEXT	从正文段读一个32位值
PTRACE_PEEKDATA	从数据段读一个32位值
PTRACE_POKETEXT	把一个32位值写入正文段
PTRACE_POKEDATA	把一个32位值写入数据段
PTRACE_PEEKUSR	读CPU的普通和调试寄存器
PTRACE_POKEUSR	写CPU的普通和调试寄存器
PTRACE_GETREGS	读有特权的CPU寄存器
PTRACE_SETREGS	写有特权的CPU寄存器
PTRACE_GETFPREGS	读浮点寄存器
PTRACE_SETFPREGS	写浮点寄存器

ptrace()系统调用修改被跟踪进程描述符的p_pptr域以使它指向跟踪进程，因此，跟踪进程变为被跟踪进程的有效父进程。当执行跟踪终止时，也就是当以PTRACE_DETACH命令调用ptrace()时，这个系统调用把p_pptr设置为p_opptr的值，恢复被跟踪进程原来的父进程（参看第三章中的“进程之间的亲属关系”一节）。

与被跟踪程序相关的几个监控事件为：

- 一条单独汇编指令执行的结束
- 进入一个系统调用
- 从一个系统调用退出
- 接收到一个信号

当一个监控的事件发生时，被跟踪的程序停止，并且将SIGCHID信号发送给它的父进程。当父进程希望恢复子进程的执行时，就使用PTRACE_CONT、PTRACE_SINGLESTEP和PTRACE_SYSCALL命令中的一条命令，这取决于父进程要监控哪种事件。

PTRACE_CONT命令只重新恢复执行，子进程将一直执行到收到另一个信号。这种跟踪是通过进程描述符中的PF_PTRACED标志实现的，而这个标志的检查是由do_signal()函数进行的（参看第九章中的“接收信号”一节）。

PTRACE_SINGLESTEP命令强迫子进程执行下一条汇编语言指令，然后又停止它。这种跟踪是基于Intel机器eflags寄存器的TF陷阱标志而实现的，当这个标志为1时，在任一条汇编语言指令之后正好产生一个“debug”异常。相应的异常处理程序只是清掉这个标志，强迫当前进程停止，并发送SIGCHLD信号给父进程。注意，设置TF标志并不是特权操作，因此用户态进程即使在没有ptrace()系统调用的情况下，也能强迫单步执行。内核检查进程描述符中的PF_DTRACE标志以跟踪子进程是否通过ptrace()进行单步执行。

PTRACE_SYSCALL命令使被跟踪的进程重新恢复执行，直到一个系统调用被调用。进程停止两次，第一次是在系统调用开始时，第二次是在系统调用终止时。这种跟踪是利用进程描述符中的PF_TRACESYS标志实现的，这个标志是在system_call()汇编语言的函数中被检查（参看第八章中的“system_call()函数”一节）。

也可以利用Intel Pentium处理器的一些调试特点来跟踪一个进程。例如，父进程使用PTRACE_POKEUSR命令为子进程设置r0，...dr7调试寄存器的值。当一个监控事情发生时，产生“Debug”异常，异常处理程序然后挂起被调试的进程并给父进程发送SIGCHLD信号。

可执行格式

Linux正式的可执行格式是*ELF*（Executable and Linking Format），它由Unix系统实验室开发并在Unix世界相当流行。几个著名的Unix操作系统（如System V Release 4和Sun的Solaris 2）都把ELF作为它们的主要可执行格式。

Linux的旧版支持名叫*a.out*的另一种格式（汇编程序的OUTput格式），实际上，在Unix世界有好几种版本使用这种格式。因为现在ELF非常实用，因此已经很少用a.out格式。

Linux支持很多其他不同格式的可执行文件，在这种方式下，Linux能运行为其他操作系统所编译的程序，如MS-DOS的EXE程序，或Unix BSD的COFF可执行格式。有几种可执行格式，如Java或*bash*，是与平台相关的。

由类型为linux_binfmt的对象所描述的可执行格式实质上提供以下三种方法：

load_binary
: 通过读存放在可执行文件中的信息为当前进程建立一个新的执行环境。

load_shlib
: 用于动态地把一个共享库捆绑到一个已经在运行的进程，这是由uselib()系统调用激活的。

core_dump
: 在名为core的文件中存放当前进程的执行上下文。这个文件通常是在进程接收到一个缺省操作为“dump”的信号时被创建的，其格式取决于被执行程序的可执行类型。（参见第九章的“接收信号之前所执行的操作”一节。）

所有的linux_binfmt对象都处于一个简单的连接链表中，第一个元素的地址被存放在formats变量中。可以通过调用register_binfmt()和unregister_binfmt()函数在链表中插入和删除元素。在系统启动期间，为每个编译进内核的可执行格式

都执行 register_binfmt()函数。当实现了一个新的可执行格式的模块正被装载时，这个函数也被执行，当模块被卸载时，unregister_binfmt()函数被执行。

在 formats 链表中的最后一个元素总是对解释脚本（interpreted script）的可执行格式进行描述的一个对象。这种格式只定义了 load_binary 方法。其相应的 do_load_script()函数检查这种可执行文件是否以两个 #! 字符开始。如果是，这个函数就以另一个可执行文件的路径名作为参数解释第一行的其余部分，并把脚本文件名作为参数传递过去以执行这个脚本文件（注 5）。

Linux 允许用户注册自己定义的可执行格式。对这种格式的识别或者通过存放在文件前 128 字节的魔数，或者通过文件类型的扩展名。例如，MS-DOS 的扩展名由 "." 把三个字符从文件名中分离出来：*.exe* 扩展名标识可执行文件，而 *.bat* 扩展名标识 shell 脚本。

每个自定义格式都与一个解释程序相关联，内核把原始的自定义可执行文件作为参数来自动调用解释程序。这种机制类似于脚本的格式，但更强大，因为它对自定义格式不强加任何限制。为了注册一个新的格式，用户以下列格式向 */proc/sys/fs/binfmt_misc/register* 文件中写入一个字符串：

```
:name:type:offset:string:mask:interpreter:
```

这里，每个域的含义如下：

name
: 新格式的标识符

type
: 识别类型（M 表示魔数，E 表示扩展）

offset
: 魔数在文件中的起始偏移量

string
: 或者以魔数，或者以扩展名匹配的字节序列

注 5：只要以用户 shell 能识别的语言把文件写入脚本文件，即使不以 #! 字符开始，也可能执行这个脚本文件。但是，在这种情况下，由 shell 以用户载入的命令来对这种脚本进行解释，因此并不直接涉及内核。

mask

屏蔽 string 中的一些位的字符串

interpreter

程序解释器的完整路径名

例如，超级用户执行的下列命令将使内核识别出Microsoft Windows的可执行格式：

```
$ echo ':DOSWin:M::0:MZ::0xff:/usr/local/bin/wine:' > \
   /proc/sys/fs/binfmt_misc/register
```

Windows可执行文件的前两个字节是魔数MZ，由*/usr/local/bin/wine*程序解释器执行这个可执行文件。

执行域

在第一章已提到，Linux的一个巧妙的特点就是能执行其他操作系统所编译的程序。当然，只有内核运行的平台与可执行文件包含的机器代码对应的平台相同时这才是可能的。对这些“外来”程序提供两种支持：

- 模拟执行（emulated execution）：程序中包含的系统调用与POSIX不兼容时才有必要执行这种程序
- 原样执行（native execution）：只有程序中所包含的系统调用完全与POSIX兼容时才有效

Microsoft MS-DOS和Windows程序是被模拟执行的，因为它们包含的API不能被Linux所认识，因此不能原样执行。象DOSemu或Wine这样的模拟程序（出现在上一节末尾的例子中）被调用来把每个API调用转换为一个模拟的封装函数调用，而封装函数调用又使用现有的Linux系统调用。因为模拟程序主要是作为用户态的应用程序来执行，因此我们在此不做进一步的讨论。

另一方面，不用太费力就可以执行为其他操作系统编译的与POSIX兼容的程序，因为与POSIX兼容的操作系统都提供了类似的API。（尽管实际上并不总是这种情况，但API应该相同）。内核必须消除的细微差别通常涉及如何调用系统调用或如何给各种信号编号。这种信息存放在类型为 exec_domain 的执行域描述符（execution domain descriptor）中。

进程可以指定它的执行域，这是通过设置进程描述符的personality域，以及存放在exec_domain域中exec_domain数据结构所对应的地址来实现的。进程可以通过发布一个叫做personality()的系统调用来改变它的个性（personalify）。表19-6列出了这个系统调用的参数所接收的典型值。在C库中不包含相应的封装程序，因为程序员不希望直接改变他们程序的个性。相反，系统调用personality()通过“粘合”代码来建立进程的执行上下文（参见下一节，“exec类函数”）。

表19-6 Linux内核所支持的主要个性

个性	操作系统
PER_LINUX	标准执行域
PER_SVR4	System V Release 4
PER_SVR3	System V Release 3
PER_SCOSVR3	SCO Unix version 3.2
PER_WYSEV386	Unix System V/386 Release 3.2.1
PER_ISCR4	交互式Unix
PER_BSD	BSD Unix
PER_XENIX	Xenix
PER_IRIX32	SGI Irix-5 32位
PER_IRIXN32	SGI Irix-6 32位
PER_IRIX64	SGI Irix-6 64位

exec类函数

Unix系统提供了一个函数家族，这些函数能用可执行文件所描述的新上下文代替进程的上下文。这样的函数名以前缀exec开始，后跟一个或两个字母，因此，家族中的一个普通函数被当作exec类函数来引用。

表19-7中列出exec类函数，它们的不同在于如何解释参数。

表 19-7 exec 类函数

函数名	路径搜索	命令行参数	环境数组
execl()	否	列表	否
execlp()	是	列表	否
execle()	否	列表	是
execv()	否	数组	否
execvp()	是	数组	否
execve()	否	数组	是

每个函数的第一个参数表示被执行文件的路径名。路径名可以是绝对路径或是当前进程目录的相对路径。此外，如果路径名中不包含 "/" 字符，execlp()和execvp()函数在 PATH 环境变量指定的所有目录中搜索这个可执行文件。

除了第一个参数，execl()，execlp()和execle()函数包含的其他参数个数是可变的。每个参数指向新程序命令行参数的一个字符串，正如函数名中 l 字符所隐含的一样，这些参数组织成一个列表（最后一个值为 NULL）。通常情况下，第一个命令行参数复制可执行文件名。相反，execv()，execvp()和execve()函数指定单个参数的命令行参数，正如函数名中的 v 字符所隐含的一样，这单个参数是指向命令行参数串的指针向量地址。数组的最后一个元素必须存放 NULL 值。

execle()和execve()函数的最后一个参数是指向环境串的指针数组的地址；数组的最后一个元素照样必须为 NULL。其他函数对新程序环境参数的访问是通过 C 库定义的外部全局环境变量 environ 进行的。

所有的 exec()类函数［除 execve()外］都是 C 库定义的封装例程，并利用了 execve()系统调用，这是 Linux 所提供的处理程序执行的唯一系统调用。

sys_execve()服务例程接受下列参数：

- 可执行文件路径名的地址（在用户态地址空间）。
- 以NULL结束的字符串指针数组的地址（数组和串均在用户态地址空间）。每个字符串表示一个命令行参数。

- 以NULL结束的字符串指针数组的地址(数组和串均在用户态地址空间)。每个字符串以NAME=value形式表示一个环境变量。

sys_execve()把可执行文件路径名拷贝到一个新分配的页框。然后调用do_execve()函数，传递给它的参数为指向这个页框的指针、指针数组的指针及用户态寄存器内容要保存到内核态堆栈的位置指针。do_execve()依次执行下列操作:

1. 静态地分配一个linux_binprm数据结构，并用新的可执行文件的数据填充这个结构。
2. 调用open_namei()以获得目录项对象，因此也就获得了与这个可执行文件相关的文件对象和索引节点对象。如果失败，返回适当的错误码。
3. 调用prepare_binprm()函数填充linux_binprm数据结构。这个函数又依次执行下列操作:
   a. 检查文件的许可权，看是否允许这个文件执行；如果不允许，返回错误码。
   b. 检查这个文件是否可以被写(即索引节点的i_writecount域不为空)；如果是，返回一个错误码。
   c. 初始化linux_binprm结构的e_uid 和e_gid域，考虑可执行文件*setuid*和*setgid*标志的值。这些域分别表示有效的用户ID和组ID。也要检查进程的能力(在本章前面的"进程的信任状和能力"一节中绍了能力的技巧)。
   d. 用可执行文件的前128字节填充linux_binprm结构的buf域。这些字节包含的是适合于识别可执行文件格式的一个魔数和其他信息。
4. 把文件路径名、命令行参数及环境串拷贝到一个或多个新分配的页框中。(最终，它们会被分配给用户态地址空间)。
5. 调用search_binary_handler()函数对formats链表进行扫描，并尽力应用每个元素的load_binary方法，把linux_binprm数据结构传递给这个函数。只要load_binary方法成功应答了文件的可执行格式，对formats的扫描就终止。
6. 如果可执行文件格式不在formats链表中，释放所分配的所有页框并返回错误码-ENOEXEC，表示Linux不认识这个可执行文件格式。
7. 否则，返回从这个文件可执行格式的load_binary方法中所获得的代码。

可执行文件格式对应的load_binary方法执行下列操作(我们假定这个可执行文件所在的文件系统允许文件进行内存映射并需要一个或多个共享库):

1. 检查存放在文件前128字节中的一些魔数以确认可执行格式。如果魔数不匹配，则返回错误码-ENOEXEC。
2. 读可执行文件首部。这个首部描述程序的段和所需的共享库。
3. 从可执行文件获得程序解释器的路径名，用程序解释器来确定共享库的位置并把它们映射到内存。
4. 获得程序解释器的目录项对象(也就获得了索引节点对象和文件对象)。
5. 检查程序解释器的执行许可权。
6. 把程序解释器的前128字节拷贝到linux_binprm结构的buf域。
7. 对程序解释器的类型执行一些一致性检查。
8. 调用flush_old_exec()函数释放前一个计算所占用的几乎所有资源。这个函数又依次执行下列操作:
   a. 如果信号处理程序的表为其他进程所共享，那么就分配一个新表并把原来表的引用计数器减1，这是通过调用make_private_signals()函数完成的。
   b. 通过重新设置每个信号的缺省操作来更新信号处理程序表，这是通过调用release_old_signals()和flush_signal_handlers()函数完成的。
   c. 调用exec_mmap()函数释放相应的内存描述符，所有的线性区以及分配给这个进程的所有页框，并清除进程的页表。
   d. 用可执行文件的路径名设置进程描述符的comm域。
   e. 调用flush_thread()函数清除浮点寄存器的值和在TSS段保存的调试寄存器的值。
   f. 调用flush_old_files()函数关闭所有打开的文件，这些打开的文件在进程描述符的files->close_on_exec域设置了相应的标志(参看第十二章中的“与进程相关的文件”一节，注6)。

注6: 可以通过fcntl()系统调用来读取和修改这些标志。

现在，我们已到达不能回头的地步：如果某事出了错，这个函数再不能恢复前一个计算。

9. 建立进程新的个性，即建立进程描述符的personality域。

10. 调用setup_arg_pages()函数为进程的用户态堆栈分配一个新的线性区描述符，并把那个线性区插入到进程的地址空间。setup_arg_pages()还给新的线性区分配包含命令行参数和环境变量串的页框。

11. 调用do_mmap()函数创建一个新线性区来对可执行文件正文段（即代码）进行映射。这个线性区的起始地址依赖于可执行文件的格式，因为程序的可执行代码通常是不可重定位的。因此，这个函数假定从某一特定逻辑地址的偏移量开始（因此就从某一特定的线性地址开始）装入正文段。ELF程序被装入的起始线性地址为0x08048000。

12. 调用do_mmap()函数创建一个新线性区来对可执行文件的数据段进行映射。这个线性区的起始地址也依赖于可执行文件的格式，因为可执行代码希望在特定的偏移量（即特定的线性地址）处找到它自己的变量。在ELF程序中，数据段正好被装在正文段之后。

13. 为可执行文件的其他专用段分配另外的线性区，通常是无。

14. 调用一个能装入程序解释器的函数。如果程序解释器是ELF的可执行格式，这个函数就叫做load_elf_interp()。一般情况下，这个函数执行第11步到13步的操作，不过要用程序解释器代替被执行的文件。程序解释器的正文段和数据段在线性区的起始地址是由程序解释器本身指定的；但它们处于高地址区（通常高于0x40000000），这是为了避免与被执行文件的正文段和数据段所映射的线性区发生冲突（参看前面的“程序段和进程的线性区”一节）。

15. 根据新程序的个性设置进程描述符的exec_domain域。

16. 确定进程新的能力。

17. 清掉进程描述符中的PF_FORKNOEXEC标志。这是POSIX标准要求为进程计账的一个标志，当进程被创建时设置它，当进程执行一个新的程序时清除它。

18. 创建特定的程序解释器表并把它们存放在用户态堆栈，如图19-1所示，这些表处于命令行参数和指向环境串的指针数组之间。

19. 设置进程的内存描述符的start_code、end_code、end_data、start_brk、brk及start_stack域。

20. 调用do_mmap()函数创建一个新的匿名线性区来映射程序的bss段。(当进程写入一个变量时，就触发请求调页，因此分配一个页框)。这个线性区的大小是在可执行程序被连接时就计算出的。因为程序的可执行代码通常是不可重新定位的，因此，必须指定这个线性区的起始地址。在ELF程序中，bss段正好装在数据段之后。

21. 调用start_thread()宏修改保存在内核态堆栈但属于用户态寄存器的eip和esp的值，以使它们分别指向程序解释器的入口点和新的用户态堆栈的栈顶。

22. 如果进程正被跟踪，向它发送一个SIGTRAP信号。

23. 返回值0(成功)。

当execve()系统调用终止且调用进程重新恢复它在用户态的执行时，执行上下文被戏剧性地改变，调用系统调用的代码不再存在。从这个意义上看，我们可以说execve()从未成功返回。取而代之的是，要执行的新程序已被映射到进程的地址空间。

但是，新程序还不能执行，因为程序解释器还必须照顾共享库的装载(注7)。

尽管程序解释器运行在用户态，但我们在这里简要概述一下它是如何操作的。它的第一个工作就是从内核保存在用户态堆栈的信息(处于环境串指针数组和arg_start之间)开始，为自己建立一个基本的执行上下文。然后，程序解释器必须检查被执行的程序，以识别哪个共享库必须被装入，在每个共享库中哪个函数被有效地请求。接下来，解释器发布几个mmap()系统调用来创建线性区以对程序实际使用的库函数(正文和数据)将存放的页进行映射。然后，解释器更新对共享库符号的所有引用，这是根据库的线性区线性地址来进行的。最后，程序解释器通过跳转到被执行程序的主入口点而终止它的执行。从现在开始，进程将执行可执行文件的代码和共享库的代码。

注7：如果可执行文件是静态链接的，即如果不需要共享库，事情就简单多了。load_binary方法只需将程序的正文段、数据段、bss段和堆栈段映射到进程线性区，然后把用户态eip寄存器的内容设置为新程序的入口点即可。

你可能已注意到，执行程序是一个相当复杂的活动，它涉及内核设计的很多因素，如进程抽象、内存管理、系统调用及文件系统。这使你认识到Linux不可思议的工作片段！

对Linux 2.4的展望

Linux 2.4增加了几个个性：新内核能执行为SunOS、Sun Solaris和RISCOS操作系统所写的程序。不过，execve()系统调用的实现与Linux 2.2还完全相同。

附录一
系统启动

本附录介绍当用户打开自己计算机电源之后所发生的事情，也就是说，Linux内核映像是如何被拷贝到内存的，又是如何被执行的。简而言之，我们先讨论内核是如何启动的，然后讨论整个系统是如何启动的。

*bootstrap*这个术语的原意是指一个人要穿上靴子站起来。在操作系统中，这个术语专门表示把一部分操作系统装载到主存中并让处理器执行它，也表示内核数据结构的初始化、一些用户进程的创建以及用户进程之间控制权的转移。

计算机启动是一个冗长乏味的任务，因为最开始时几乎每个硬件设备（包括RAM）都处于一种随机的、不可预知的状态。而且，启动过程在很大程度上都依赖于计算机的体系结构。和以前一样，我们在本附录中特指IBM的PC体系结构。

史前时代：BIOS

计算机在加电的那一刻，几乎是毫无用处的，因为RAM芯片中包含的是随机数据，此时还没有操作系统在运行。在开始启动时，有一个特殊的硬件电路在CPU的一个引脚上产生一个RESET逻辑值。在RESET产生以后，就把处理器的一些寄存器（包括cs和eip）设置成固定的值，并执行在物理地址0xfffffff0处找到的代码。硬件把这个地址映射到某个只读、持久的存储芯片中，这就是通常称为ROM（只读内存）的一种内存。ROM中所存放的程序集传统上称为BIOS（基本输入/输出系统），

因为它包括几个中断驱动的低级程序，一些操作系统（包括微软的MS-DOS）可以用这些程序来处理组成计算机的硬件设备。

Linux一旦被初始化，就不再使用BIOS，而是为计算机上的每个硬件设备提供各自的设备驱动程序。实际上，因为必须在实模式下执行BIOS过程，而内核要在保护模式下执行（请参看第二章中的“硬件的分段单元”一节），所以即使在二者之间共享函数是有益的，但是也不能那样处理。

BIOS使用实模式的地址，因为只有在计算机加电启动时才可以使用BIOS。一个实模式的地址由一个 *seg* 段和一个 *off* 偏移量组成。相应的物理地址可以这样计算：*seg**16 +*off*。所以CPU寻址电路把逻辑地址转换成物理地址根本就不需要全局描述符表、局部描述符表和页表。显然，对ODT、LDT和页表进行初始化的代码必须在实模式下运行。

Linux在启动阶段要集中使用BIOS，此时Linux必须要从磁盘或者其他外部设备中获取内核映像。BIOS启动过程实际上执行以下4个操作：

1. 对计算机硬件执行一系列的测试，用来检测现在都有什么设备以及这些设备是否正常工作。这个步骤通常称为POST（上电自检）。在这个阶段中，会显示一些信息，例如BIOS版本号。
2. 初始化硬件设备。这个步骤在现代基于PCI的体系结构中相当关键，因为它可以保证所有的硬件设备操作不会引起IRQ线与I/O端口的冲突。在本步骤的最后，会显示系统中所安装的所有PCI设备的一个列表。
3. 搜索一个操作系统来启动。实际上，根据BIOS的设置，这个过程可能要试图访问（按照用户预定义的次序）系统中软盘、硬盘和CD-ROM的第一个扇区（引导扇区）。
4. 只要找到一个有效的设备，就把第一个扇区的内容拷贝到RAM中从物理地址0x00007c00开始的位置，然后跳转到这个地址处，开始执行刚才装载进来的代码。

本附录其余的部分会带你体验从最原始的开始状态到运行Linux系统的整个历程。

远古时代：引导装入程序

引导装入程序（boot loader）是由BIOS用来把操作系统的内核映像装载到RAM中所调用的一个程序。让我们简要地描绘一下引导装入程序在IBM的PC体系结构中是如何工作的。

为了从软盘上启动，必须把第一个扇区中所存放的指令装载到RAM中并执行；这些指令再把包含内核映像的其他所有扇区都拷贝到RAM中。

从硬盘启动的实现有点不同。硬盘的第一个扇区称为主引导记录（Master Boot Record，MBR），该扇区中包括分区表（注1）和一个小程序，该程序用来装载要启动的操作系统所在分区的第一个扇区。诸如Microsoft Windows 98之类的操作系统使用分区表中所包含的一个活动（active）标志来标识这个分区（注2）。按照这种方法，只有那些内核映像存放在活动分区中的操作系统才可以被启动。正如我们将在后面看到的一样，Linux的处理方式更加灵活，因为Linux使用一个巧妙的程序把这个包含在MBR中的不完善的程序替换掉，这个程序称为LILO，它允许用户来选择要启动的操作系统。

从软盘中启动Linux

把Linux内核存放在一张软盘上的唯一一种方法是对内核映像进行压缩。正如我们将要看到的一样，压缩是在编译时完成的。而装入程序所要处理的是解压缩。

如果Linux内核是从一张软盘中被装载的，那么引导装入程序就相当简单。它是以汇编语言编写，存放在*arch/i386/boot/bootsec.S*文件中。当通过编译内核源代码而产生一个新的内核映像时，这个汇编语言文件所产生的可执行代码被放在这个内核映像文件的开头。因此，生成一张包含Linux内核的可启动的软盘就很简单。可以把磁盘上从第一个扇区开始的内核映像拷贝到软盘中来创建软盘。当BIOS装载软盘的第一个扇区时，实际上就是拷贝引导装入程序的代码。

注1：每个典型的分区表项都包括一个分区的开始扇区和结束扇区以及对此进行处理的操作系统的类型。

注2：这个活动标志可以使用诸如MS-DOS的FDISK之类的程序进行设置。

BIOS通过跳转到物理地址0x00007c00来调用引导装入程序，装入程序执行以下操作：

1. 把自己从0x00007c00处拷贝到0x00090000处。
2. 设置从0x00003ff4开始的实模式堆栈。这个堆栈照样会向低地址方向延伸。
3. 建立磁盘参数表，这是BIOS用来处理软盘设备驱动程序的。
4. 调用一个BIOS过程显示“Loading”信息。
5. 调用一个BIOS过程从软盘中装载内核映像的setup()代码，并把该代码放入从0x00090200开始的RAM中。
6. 调用一个BIOS过程从软盘中装载其余的内核映像，并把内核映像放入从低地址0x00010000（适用于使用make zImage编译的小内核映像）或者高地址0x00100000（适用于使用make bzImage编译的大内核映像）开始的RAM中。在以下的讨论中，我们将分别称为内核映像是“低装载”到RAM中或者“高装载”到RAM中。对于大内核映像的支持是最近才引入的，虽然它使用的启动模式与原来相同，但是却把数据放在不同的物理内存地址，以避免在第二章中的“保留的页框”一节中所介绍的ISA黑洞问题。
7. 跳转到setup()代码。

从硬盘启动

在大多数情况下，Linux内核是从硬盘装入的，并需要一个两阶段引导装入程序。在Intel系统上，Linux最常使用的引导装入程序就是所谓的LILO（LInux LOader）。其他的体系结构也存在相应的程序。LILO或许被装在MBR上以代替那个小程序（装载激活的引导扇区），或许被装在一个磁盘分区（通常是激活的）的引导扇区上。在这两种情况下，最终的结果是相同的：在启动过程中装入程序被执行时，用户都可以选择装入哪个操作系统。

LILO引导装入程序被分为两部分，因为不划分的话，它太大而无法装进MBR中。MBR或者分区引导扇区包括一个小的引导装入程序，由BIOS把这个小程序装入从地址0x00007c00开始的RAM中。这个小程序又把自己移到地址0x0009a000，建立实模式栈（从0x0009b000到0x0009a200），并把LILO的第二部分装入到从地址0x0009b000开始的RAM中。第二部分又依次从磁盘读取可用操作系统的映射

表，并提供给用户一个提示符，因此用户就可以从中选择一个操作系统。最后，用户选择了被装入的内核后（或给一个延迟时间以使LILO选择一个缺省值），引导装入程序就可以把相应分区的引导扇区拷贝到RAM中并执行它，或直接把内核映像拷贝到RAM中。

假定Linux内核映像必须被导入，LILO引导装入程序依赖于BIOS例程，LILO执行的操作本质上等价于在前面关于软盘部分所描述的集成进内核映像中的引导装入程序。这个装入程序显示“Loading Linux”信息，把内核映像中所集成的引导装入程序拷贝到地址0x0009000，把setup()代码拷贝到地址0x00090200，并把内核映像的其余部分拷贝到地址0x00010000或0x00100000。然后，跳转到setup()代码。

中世纪：setup()函数

链接程序把setup()汇编语言函数代码放在紧跟内核中所集成的引导装入程序之后，也就是说，内核映像文件的偏移量0x200处。引导装入程序因此就可以很容易地确定setup()代码的位置，并把它拷贝到从物理地址0x00090200开始的RAM中。

setup()函数必须初始化计算机中的硬件设备并为内核程序的执行建立环境。虽然BIOS已经初始化了大部分硬件设备，但是Linux并不依赖于BIOS，而是以自己的方式重新初始化设备以增加可移植性和健壮性。setup()本质上执行以下操作：

1. 调用一个BIOS过程来检查系统中可用RAM的数量。
2. 设置键盘重复延时和速率。（当用户一直按下一个键超过一定的时间，键盘设备就反复地向CPU发送相应的键盘码。）
3. 初始化视频卡。
4. 重新初始化磁盘控制器并检测硬盘参数。
5. 检查IBM微通道总线（MCA）。
6. 检查PS/2设备（总线鼠标）。
7. 检查对高级电源管理（APM）BIOS的支持。
8. 如果内核映像被低装载到RAM中（在物理地址0x00010000处），就把它移动

到物理地址0x00001000处。反之，如果内核映像被高装载到RAM中，就不用移动。这个步骤是必须的，因为为了能在软盘上存储内核映像并节省启动的时间，存放在磁盘上的内核映像都是压缩的，解压程序需要一些空闲空间作为临时缓冲区（紧挨着RAM中的内核映像）。

9. 建立一个临时中断描述符表（IDT）和一个临时全局描述符表（GDT）。

10. 如果需要，重置浮点单元（FPU）。

11. 重新编写可编程中断控制器（PIC），并把16个硬件中断（IRQ线）映射到从32到47的中断向量。内核必须执行这个步骤，因为BIOS错误地把硬件中断映射到从0到15的中断向量，而这些中断向量早已用于CPU的异常（请参看第四章中的“异常”一节）。

12. 通过设置cr0状态寄存器中的PE位把CPU从实模式切换到保护模式。正如我们在第二章中的“内核页表”一节中已经介绍过的一样，临时内核页表包含在swapper_pg_dir中，pg0相等地把线性地址映射成相同的物理地址。因此，从实模式到保护模式的转换很顺利。

13. 跳转到startup_32()汇编语言函数。

文艺复兴时期：startup_32()函数

有两个不同的startup_32()函数，我们此处所使用的这个函数是在*arch/i386/boot/compressed/head.S*文件中实现的。在setup()结束之后，该函数就已经被移动到物理地址0x00100000处或者0xC0001000处，这取决于内核映像是被高装载到RAM中还是低装载到RAM中。

该函数执行以下操作：

1. 初始化段寄存器和一个临时堆栈。

2. 用0填充由_edata和_end符号标识的内核未初始化数据区（请参看第二章中的“保留的页框”一节）。

3. 调用decompress_kernel()函数来解压内核映像。首先显示“Uncompressing Linux...”信息。完成内核映像的解压之后，显示“OK, booting the kernel.”信息。如果内核映像是低装载的，那么解压后的内核就被放在物理地址

0x00100000处。否则，如果内核映像是高装载的，那么解压后的内核就被放在位于这个压缩映像之后的临时缓冲区中。然后，解压后的映像就被移动到最终的从物理地址0x00100000开始的位置。

4. 跳转到物理地址0x00100000处。

第二个startup_32()函数实际上为第一个Linux进程（进程0）建立执行环境。该函数执行以下操作：

1. 把段寄存器初始化为最终值。
2. 为进程0建立内核态堆栈（请参看第三章中的“内核线程”一节）。
3. 调用setup_idt()，用空中断处理程序填充IDT（请参看第四章中的“初步IDT的初始化”一节）。
4. 把从BIOS中获得的系统参数和传递给操作系统的参数放入第一个页框中（请参看第二章中的“保留的页框”一节）。
5. 识别处理器的模式。
6. 用GDT和IDT表的地址来填充gdtr和idrt寄存器。
7. 跳转到start_kernel()函数。

现代：start_kernel()函数

start_kernel()函数完成Linux内核的初始化工作。几乎内核每个部分的初始化都用到这个函数。我们只介绍其中的一部分：

- 调用paging_init()函数初始化页表（请参看第二章中的“内核页表”一节）。
- 调用mem_init()函数初始化页描述符（请参看第六章中的“页框管理”一节）。
- IDT最后的初始化工作通过调用trap_init()函数（请参看第四章中的“异常处理”一节）和init_IRQ()函数（请参看第四章中的“IRQ数据结构”一节）来完成。
- 调用kmem_cache_init()函数和kmem_cache_sizes_init()函数初始化slab分配器（请看第六章的“通用和专用高速缓存”一节）。

- 调用 time_init()函数初始化系统日期和时间（请参看第五章中的“实时时钟”一节）。
- 调用 kernel_thread()函数为进程 1 创建内核线。正如我们在第三章的“内核线程”一节中已经描述的一样，这个内核线程又会创建其他的内核线程并执行 */sbin/init* 程序。

在 start_kernel()开始执行之后会显示“Linux version 2.2.14 . . .”信息，除此之外，在 init 函数和内核线程执行的最后阶段还会显示很多其他信息。最后，就会在控制台上出现熟悉的登录提示符（如果在启动时所启动的是 X Window 系统，那么就会出现一个图形窗口），通知用户 Linux 内核已经启动，现在正在运行。

附录二

模块

正如我们在第一章中所介绍的一样，模块（module）是Linux用来高效地利用微内核的理论优点而不会降低系统性能的一种方法。

是否使用模块？

当系统程序员希望给Linux内核增加新功能时，就面临一个进退两难的问题：他们应该编写新代码从而将其作为一个模块进行编译还是应该将这些代码静态地链接到内核中？

通常，系统程序员都倾向于把新代码作为一个模块来实现。因为模块可以根据需要进行链接，这样内核就不会因为装载那些数以百计的很少使用的程序而变得非常庞大，这一点我们后面就会看到。几乎Linux内核的每个高层组件（component）——文件系统、设备驱动程序、可执行格式、网络层等等——都可以作为模块进行编译。

然而，有些Linux代码必须被静态链接，也就是说相应组件或者被包含在内核中，或者根本不被编译。特别是当组件需要对静态链接到内核中的数据结构或函数进行修改时就发生这种情况。

例如，假设某个组件必须在进程描述符中引入新域。链接一个模块并不能修改诸如`task_struct`之类已经定义的数据结构，因为即使模块使用这个数据结构的修改

版，所有静态链接的代码看到的仍是原来的版本，这样就很容易发生数据崩溃。对此问题一种局部的解决方法就是“静态地”把新域加到进程描述符，从而让这个组件可以使用这些域，而不用考虑组件究竟是如何被链接的。然而，如果该组件从未被使用，那么，在每个进程描述符中都复制这个额外的域就是对内存的浪费。如果这个新组件对进程描述符的大小有很大的增加，那么，只有新组件被静态地链接到内核，它才可能通过在数据结构中增加所需要的域获得较好的系统性能。

再例如，考虑一个内核组件，它要替换静态链接的代码。显然，这样的组件不能作为一个模块编译，因为在链接模块时内核不能修改已经在RAM中的机器码。例如，系统不可能链接一个改变页框分配方法的模块，因为Buddy系统函数总是被静态地链接到内核中。

内核有两个主要的任务来执行模块的管理。第一个任务确保内核的其他部分可以访问该模块的全局符号，例如指向模块主函数的入口。模块还要知道这些符号在内核及其他模块中的地址。因此，在链接模块时，一定要解决模块间的引用关系。第二个任务是记录模块的使用情况，以便在其他模块或者内核的其他部分正在使用这个模块时，就不能把这个模块取消。系统使用了一个简单的引用计数器来记录每个模块的引用次数。

模块的实现

模块在文件系统中是以ELF对象文件存储的。内核只会考虑那些由*/sbin/insmod*程序装载到RAM中的模块（请参看后面的“模块的链接和取消”一节），对于每个模块，系统都分配一个包含以下数据的内存区：

- 一个module对象
- 表示模块名的一个以null结束的字符串（所有的模块都必须有唯一的名字）
- 实现模块功能的代码

module对象描述了一个模块，其域如表B-1所示。由一个简单链表把所有的module对象链接在一起，其中每个对象的next域都指向链表的下一个元素。链表的第一个元素是由module_list变量进行寻址的。但是实际上，链表的第一个元素通常都是相同的，其名为kernel_module，指向一个静态链接到内核代码中的伪模块。

表B-1 模块对象

类型	域名	说明
unsigned long	size_of_struct	module对象的大小
struct module *	next	下一个链表元素
const char *	name	指向模块名的指针
unsigned long	size	模块大小
atomic_t	uc.usecount	模块引用计数器
unsigned long	flags	模块标志
unsigned int	nsyms	导出符号的个数
unsigned int	ndeps	引用模块的个数
struct module_symbol *	syms	导出符号表
struct module_ref *	deps	被引用模块（该模块所引用的模块）的链表
struct module_ref *	refs	正在引用的模块（引用该模块的模块）链表
int (*)(void)	init	初始化方法
void (*)(void)	cleanup	清除方法
struct exception_table_entry *	ex_table_start	异常表的开始
struct exception_table_entry *	ex_table_end	异常表的结束

给模块（包括module对象和模块名）分配的内存区总量包含在size域中。

正如我们在第八章中的“动态地址检查：修正代码”一节中已经介绍的一样，每个模块都有自己的异常表。该表包括（如果有）模块的修正代码的地址。在链接模块时，该表被拷贝到RAM中，其开始地址和结束地址分别保存在module对象的ex_table_start和ex_table_end域中。

模块引用计数器

每个模块都有一个引用计数器，存放在相应module对象的uc.usecount域中。在

开始执行模块功能所涉及的操作时递增这个计数器，在操作结束时递减这个计数器。只有这个引用计数器为0时，模块才可以被取消。

例如，假设MS-DOS文件系统层已经作为模块被编译，而且这个模块已经在运行时被链接。最开始时，该模块的引用计数器是0。如果用户装载一张MS-DOS软盘，那么这个模块引用计数器就被递增1。反之，当用户卸载这张软盘时，这个计数器就被减1。

导出符号

当链接一个模块时，必须用合适的地址替换在模块对象代码中所引用的所有全局内核符号（变量和函数）。这个操作与在编译用户态的程序时链接程序所执行的操作（请参看第十九章中的“库”一节）非常类似，这是委托给*/sbin/insmod*外部程序完成的（在后面的“模块的链接和取消”一节介绍）。

内核使用一个特殊的表存放模块都可以访问的那些符号以及相应的地址。内核符号表（kernel symbol table）处于内核代码段的__ksymtab部分，其开始地址和结束地址是由C编译器所产生的两个符号来指定：__start___ksymtab和__stop___ksymtab。对于静态链接到内核内部的代码，使用EXPORT_SYMBOL宏强制C编译器向这个表中增加一个指定的符号。

只有某一现有的模块实际使用的内核符号才会保存在这个表中。如果系统程序员在某些模块中需要访问一个已经导出的内核符号，那么他只要在Linux源代码的*kernel/ksyms.c*文件中增加相应的EXPORT_SYMBOL宏就可以了。

已链接的模块也可以导出自己的符号，这样其他模块就可以访问这些符号。模块符号表（module symbol table）处于模块代码段的__ksymtab部分。如果模块源代码包括EXPORT_NO_SYMBOLS宏，那么该模块中就没有任何符号被加入符号表。要从模块中导出部分符号，程序员就必须在包含*include/linux/module.h*的头文件之前定义EXPORT_SYMTAB宏。然后，他就可以使用EXPORT_SYMBOL宏来导出一个特定的符号。如果在模块的代码中既没有出现EXPORT_NO_SYMBOLS宏，也没有出现EXPORT_SYMTAB宏，那么该模块的所有全局符号就全部被导出来了。

在链接模块时，__ksymtab部分中的符号表被拷贝到内存区中，这个内存区的地址被存放在module对象的syms域中。静态链接的内核所导出的符号以及链接到模块

中的所有符号都可以通过读*/proc/ksyms*文件或者使用query_module()系统调用进行访问（在下一节“模块的链接和取消”中介绍）。

模块依赖

一个模块(B)可以引用由另一个模块(A)所导出的符号。在这种情况下，我们就说B被装载到A的上面，或者说A被B使用。为了链接模块B，必须首先链接模块A；否则，对于模块A所导出的那些符号的引用就不能被正确地链接到B中。简而言之，在这两个模块之间存在着依赖。

与B相关的module对象的deps域指向一个描述B所使用的所有模块的链表。在我们的例子中，A的module对象就出现在这个链表中。ndeps域存放B所使用的模块个数。反之，A的refs域指向一个描述在A之上装载的所有模块（因此，在装载模块B时，B的module对象就包含在这个链表中）的链表。每当有模块在A上装载时，都必须修改refs链表。为了确保模块A不会在B之前被删除，每次在A上装载模块时都要递增A的引用计数器。

当然，除了A和B之外，还会有其他模块(C)装载到B上，依此类推。模块的堆叠(stack)是对内核源代码进行模块化的一种有效方法，目的是为了加速内核的开发以及提高内核的可移植性。

模块的链接和取消

用户可以通过执行*/sbin/insmod*外部程序把一个模块链接到正在运行的内核中。该程序执行以下操作：

1. 从命令行中读取要链接的模块名。
2. 确定模块对象代码所在的文件在系统目录树中的位置。对应的文件通常都是在*/lib/modules*的某个子目录中。
3. 计算存放模块代码、模块名和module对象所需要的内存区的大小。
4. 调用create_module()系统调用，向它传递新模块的模块名和大小。相应的sys_create_module()服务例程执行以下操作：

a. 检查是否允许该用户链接这个模块（当前进程必须具有CAP_SYS_MODULE能力）。只要给内核增加新功能的情况存在，就可以访问系统中的所有数据和进程，安全性是至关重要的一个因素。

b. 调用find_module()函数，扫描module对象的module_list链表来查找一个指定名字的模块。如果找到这个模块，说明该模块已被链接，因此这个系统调用结束。

c. 调用vmalloc()为新模块分配一个内存区。

d. 初始化这个内存区开始处的module对象的域，并把模块名拷贝到紧接该对象后面。

e. 把module对象插入到module_list所指向的链表。

f. 返回分配给这个模块的内存区的起始地址。

5. 用QM_MODULES子命令反复调用query_module()系统调用来获得所有已链接模块的模块名。

6. 用QM_SYMBOL子命令反复调用query_module()系统调用来获得内核符号表和所有已经链接到内核的模块的符号表。

7. 使用内核符号表、模块符号表以及create_module()系统调用所返回的地址重新定位该模块文件中所包含的文件对象的代码。这就意味着用相应的逻辑地址偏移量来替换所有出现的外部符号和全局符号。

8. 在用户态地址空间中分配一个内存区，并把module对象、模块名以及为正在运行的内核所重定位的模块代码的一个拷贝装载到这个内存区中。该对象的address域指向重定位的代码。如果该模块定义了init_module()函数，那么init域就被设置成该模块的init_module()函数重新分配的地址。（实际上所有的模块都定义了一个这种名字的函数，该函数在下一步执行模块所需要的初始化工作时就会被调用。）同理，如果模块定义了cleanup_module()函数，那么cleanup域就被设置成模块的cleanup_module()函数所重新分配的地址。

9. 调用init_module()系统调用，向它传递上一步中所创建的用户态的内存区地址。sys_init_module()服务例程执行以下操作：

a. 检查是否允许该用户链接模块（当前进程必须具有CAP_SYS_MODULE能力）。

b. 调用 find_module()在module_list所指向的链表中查找相应的模块对象。

c. 使用用户态内存区中的相应对象的内容覆盖module对象。

d. 对模块对象中的地址执行一系列完整性（sanity）检查。

e. 把用户态内存区的其余部分拷贝到已经分配给模块的内存区中。

f. 扫描模块链表，并初始化该模块对象的ndeps和deps域。

g. 把该模块的引用计数器设置为1。

h. 如果该模块定义了init方法，就执行它并适当地初始化模块的数据结构。该方法的实现通常是由模块内所定义的init_module()函数完成的。

i. 把该模块的引用计数器设置成0并返回。

10. 释放用户态内存区并结束。

为了取消模块的链接，用户需要调用*/sbin/rmmod*外部程序，它执行以下操作：

1. 从命令行中读取要取消的模块的模块名。

2. 使用QM_MODULES子命令调用query_module()系统调用来取得已经链接的模块的链表。

3. 使用QM_REFS子命令多次调用query_module()系统调用来检索已链接的模块间的依赖关系。如果一个要删除的模块上面还链接有某一模块，就结束。

4. 调用delete_module()系统调用，向其传递模块名。相应的sys_delete_module()服务例程执行以下操作：

   a. 检查是否允许该用户删除该模块（当前进程具有CAP_SYS_MODUL能力）。

   b. 调用find_module()在module_list所指向的链表中查找相应的module对象。

   c. 检查这个module对象的refs域和uc.usecount域是否都为空；如果不是，就返回一个错误代码。

   d. 如果该模块定义了cleanup方法，就调用这个方法来执行彻底删除该模块所需要的操作。这个方法通常都是由模块内部定义的cleanup_module()函数实现的。

e. 扫描该模块的deps链表，把所找到的模块从refs链表中删除。

f. 把该模块从module_list所指向的链表中删除。

g. 调用vfree()来释放该模块所使用的内存区并返回0（成功）。

根据需要链接模块

模块可以在系统需要其所提供的功能时自动进行链接，之后也可以自动删除。

例如，假设现在既没有静态链接MS-DOS文件系统，也没有动态链接这个模块。如果一个用户试图装载MS-DOS文件系统，那么mount()系统调用通常就会失败，返回一个错误代码，因为MS-DOS没有被包含在已注册文件系统的file_system链表中。然而，如果在配置内核时已经指定了支持模块的动态链接，那么Linux就试图链接MS-DOS模块，然后再扫描已经注册过的文件系统。如果模块成功地被链接了，那么mount()系统调用就可以继续执行，就好像MS-DOS文件系统从一开始就存在。

modprobe程序

为了自动链接模块，内核要创建一个内核线程来执行*/sbin/modprobe*外部程序（注1），该程序要考虑由于模块依赖所引起的所有可能因素。模块依赖在前面已介绍过：一个模块可能需要一个或者多个其他模块，这些模块又可能需要其他模块。例如，MS-DOS模块需要另外一个名为*fat*的模块，该模块包含基于文件分配表（FAT）的所有文件系统所通用的一些代码。因此，如果*fat*模块还不在系统中，那么在系统请求MS-DOS模块时，*fat*模块也必须被动态链接到正在运行的内核中。对模块依赖进行解析以及对模块进行查找的操作最好都在用户态中实现，因为这需要装载和访问文件系统中的模块对象文件。

*/sbin/modprobe*外部程序和*insmod*类似，因为它也是链接在命令行中指定的一个模块。然而，*modprobe*还可以递归地链接命令行中指定的模块所使用的所有模块。例如，如果用户调用*modprobe*来链接MS-DOS模块，如果需要，*modprobe*就会在

注1： 这是内核依赖于外部程序的例子之一。

MS-DOS模块之后链接*fat*模块。实际上，*modprobe*只是检查模块依赖关系，每个模块的链接工作都是通过创建一个进程并执行*insmod*命令来实现的。

*modprobe*又是如何知道模块间的依赖关系的呢？是通过另外一个称为*/sbin/depmod*的外部命令，该命令是在系统启动时执行的。该程序查找为正在运行的内核而编译的所有模块，这些模块通常是存放在*/lib/modules*目录中。然后它就把所有的模块间依赖关系写入一个名为*modules.dep*的文件。这样，*modprobe*就可以简单地对该文件中所存放的信息和query_module()系统调用所产生的链接模块表进行比较来获得模块间的依赖关系。

request_module()函数

在某些情况下，内核可以调用request_module()函数来试图自动链接一个模块。

再次考虑一个用户试图装载MS-DOS文件系统的情况。如果get_fs_type()函数发现这个文件系统还没有注册，就调用request_module()函数，希望MS-DOS已经作为一个模块被编译。

如果request_module()函数成功地链接所请求的模块，get_fs_type()就可以继续执行，就仿佛这个模块一直都存在一样。当然，并不是所有的情况都是如此。在我们的例子中，MS-DOS模块可能根本就还没有被编译。在这种情况下，get_fs_type()返回一个错误代码。

request_module()函数接收要链接的模块名作为参数。该函数调用kernel_thread()来创建一个新的内核线程执行exec_modprobe()函数，然后简单地等待直到这个内核线程结束为止。

exec_modprobe()函数也会接收要链接的模块名作为参数。该函数调用execve()系统调用并执行*/sbin/modprobe*外部程序(注2)，向其传递模块名。然后，*modeprobe*程序真正地链接所请求的模块以及这个模块所依赖的任何模块。

每个自动链接到内核中的模块在module对象集合的flags域中都有MOD_AUTOCLEAN标记。该标记允许该模块在不再需要时自动取消链接。

注2：由exec_modprobe()执行的程序名和路径名可以通过向*/proc/sys/kernel/probe*文件写入而自定义。

为了自动取消模块的链接，一个系统进程（例如*crond*）会周期性地执行*rmmod*外部程序，向其传递一个*-a*选项。后面这个程序使用NULL参数执行delete_module()系统调用。相应的服务例程扫描这个module对象链表，把所有设置了MOD_AUTOCLEAN标志的未使用模块从链表中删除。

附录三

源代码结构

为了有助于你在源代码文件中找到你所需要的文件，我们来简要介绍一下内核目录树的组织结构。通常，所有的路径名都是指Linux内核的主目录，在大部分Linux发行版本中，这个目录是*/usr/src/linux*。

Linux用来支持各种体系结构的源代码包含大约4500个C语言程序，存放在270个左右汇编文件的子目录中，总共大约包含2百万行代码，大概占用58MB磁盘空间。

下面的列表显示了Linux源代码所在的目录树。请注意，只有那些在某种程度上和本书目标相关的子目录才给予了展开介绍。

`init`	内核初始化代码
`kernel`	内核核心部分：进程，定时，程序执行，信号，模块.....
`mm`	内存处理
`arch`	平台相关代码
`└i386`	IBM的PC体系结构
`└kernel`	内核核心部分
`└mm`	内存管理
`└math-emu`	浮点单元软件仿真
`.lib`	硬件相关工具函数

├boot	引导程序
│ ├compressed	压缩内核处理
│ └tools	生成压缩内核映像的程序
├alpha	康柏的 Alpha 体系结构
├s390	IBM 的 System/390 体系结构
└sparc	Sun 的 SPARC 体系结构
├sparc64	Sun 的 Ultra-SPARC 体系结构
└mips	SGI 的 MIPS 体系结构
└ppc	Motorola-IBM 的基于 PowerPC 的体系结构
└m68k	Motorola 的基于 MC680x0 的体系结构
└arm	基于 ARM 处理器的体系结构
fs	文件系统
├proc	*/proc* 虚拟文件系统
└devpts	*/dev/pts* 虚拟文件系统
└ext2	Linux 本地的 Ext2 文件系统
└isofs	ISO9660 文件系统（CD-ROM）
└nfs	网络文件系统（NFS）
└nfsd	集成的网络文件系统服务器
├fat	基于 FAT 的文件系统的通用代码
├msdos	微软的 MS-DOS 文件系统
└vfat	微软的 Windows 文件系统（VFAT）
├nls	本地语言支持
├ntfs	微软的 Windows NT 文件系统
└smbfs	微软的 Windows 服务器消息块（SMB）文件系统
└umsdos	UMSDOS 文件系统
├minix	MINIX 文件系统
└hpfs	IBM 的 OS/2 文件系统
└sysv	System V、SCO、Xenix、Coherent 和 Version 7 文件系统

├ncpfs	Novell 的 Netware 核心协议（NCP）
└ufs	Unix BSD、SunOs、FreeBSD、NetBSD、OpenBSD 和 NeXTStep 文件系统
├affs	Amiga 的快速文件系统（FFS）
├coda	Coda 网络文件系统
└hfs	苹果的 Macintosh 文件系统
└adfs	Acorn 磁盘填充文件系统
├efs	SGI IRIX 的 EFS 文件系统
├qnx4	QNX 4 OS 使用的文件系统
├romfs	只读小文件系统
└autofs	目录自动装载程序的支持
└lockd	远程文件锁定的支持
Net	网络代码
Ipc	System V 的进程间通信
Drivers	设备驱动程序
└block	块设备驱动程序
└pariде	从并口访问 IDE 设备的支持
└scsi	SCSI 设备驱动程序
├char	字符设备驱动程序
└joystick	游戏杆
ftape	磁带流设备
├hfmodem	无线电设备
├ip2	IntelliPort 的多端口串行控制器
└net	网卡设备
└sound	音频卡设备
└video	视频卡设备
├cdrom	专用 CD-ROM 设备（除 ATAPI 和 SCSI 之外）
└isd0n	ISDN 设备
└ap1000	富士的 AP1000 设备

├macintosh	苹果的 Macintosh 设备
├sgi	SGI 的设备
│ fc4	光纤设备
├acorn	Acorn 的设备
├misc	杂项设备
├pnp	即插即用的支持
├usb	通用串行总线（USB）的支持
├pci	PCI 总线的支持
├sbus	Sun 的 SPARC SBus 的支持
├nubus	苹果的 Macintosh Nubus 的支持
├zorro	Amiga 的 Zorro 总线的支持
├dio	惠普的 HP300 DIO 总线的支持
├tc	Sun 的 TurboChannel 支持（尚未完成）
Lib	通用内核函数
Include	头文件（.h）
├linux	内核核心部分
│ ├lockd	远程文件加锁
│ ├nfsd	集成的网络文件服务器
│ ├sunrpc	Sun 的远程过程调用
│ ├byteorder	字节交换函数
│ ├modules	模块支持
├asm-generic	平台无关低级头文件
├asm-i386	IBM 的 PC 体系结构
├asm-alpha	康柏的 Alpha 体系结构
├asm-mips	SGI 的 MIPS 体系结构
├asm-m68k	Motorola-IBM 的基于 PowerPC 的体系结构
├asm-ppc	Motorola-IBM 的 PowerPC 体系结构
├asm-s390	IBM 的 System/390 体系结构
├asm-sparc	Sun 的 SPARC 体系结构

asm-sparc64	Sun 的 Ultra-SPARC 体系结构
└asm-arm	基于 ARM 处理器的体系结构
├net	网络
└scsi	SCSI 支持
└video	视频卡支持
├config	定义内核配置的宏所在的头文件
scripts	生成内核映像的外部程序
Documentation	有关内核各个部分的通用解释和注释的文本文件

参考书目

本参考书目是根据主题进行划分的，并列出了作者认为最通用、最有用的一些有关内核的书籍和在线文档。

有关 Unix 内核的书籍

Bach, M. J. *The Design of the Unix Operating System.* London: Prentice-Hall International, Inc., 1986.

一本经典的描述 SVR2 内核的书。

Goodheart, B. and J. Cox. *The Magic Garden Explained: The Internals of the Unix System V Release 4.* London: Prentice-Hall International, Inc., 1994.

一本很好的有关 SVR4 内核的书。

Leffler, S. J., McKusick, M. K., Karels, M. J. and Quarterman, J. S. *The Design and Implementation of the 4.4 BSD Operating System.* Addison Wesley, 1986.

可能是最权威的有关 4.4 BSD 内核的书。

Vahalia, U. *Unix Internals: The New Frontiers*. Upper Saddle River: Prentice-Hall, Inc., 1996.

一本很有价值的书，对现代 Unix 内核的设计进行了相当深入的讨论。该书包括相当丰富的参考书目。

有关 Linux 内核的书籍

Beck, M., H. Böhme, M. Dziadzka, U. Kunitz, R. Magnus and D. Verworner. *Linux Kernel Internals (2d ed.)*. Edinburgh Gate: Addison Wesley Longman Limited, 1998.

一本介绍 Linux 2.0 内核的硬件无关的书。

Card, R., E. Dumas, and F. Mével. *The Linux Kernel Book*. New York: John Wiley & Sons, Inc., 1998.

另外一本介绍 Linux 2.0 内核的硬件无关的书。

Maxwell, S. *Linux Core Kernel Commentary*. Scottsdale, Ariz.: The Coriolis Group, LLC, 1999.

列出了部分 Linux 内核源代码，在该书最后有对相关内容的注释。

Rubini, A. *Linux Device Drivers*. Sebastopol, Calif.: O'Reilly & Associates, Inc., 1998.

在某种程度上对本书的内容有所补充，是一本很有价值的书。该书对如何在 Linux 上开发驱动程序给出了相当丰富的信息。

有关 PC 体系结构的书和有关 Intel 80x86 的技术手册

Intel, *Intel Architecture Software Developer's Manual, vol. 3: System Programming*. 1999.

描述Intel Pentium微处理器体系结构。这本指南可以从这里下载:

http://developer.intel.com/design/pentiumii/manuals/24319202.pdf

Intel, *MultiProcessor Specification, Version 1.4*, 1997.

描述 Intel 多处理器体系结构的规范。这份规范可以从这里下载:

http://www.intel.com/design/pentium/datashts/242016.htm

Messmer, H. P. *The Indispensable PC Hardware Book (3d ed.)* Edinburgh Gate: Addison Wesley Longman Limited, 1997.

完整介绍PC的各个部分的很好的一部参考手册。

其他在线文档资源

Linux 源代码

获得源代码的官方站点可以在这里找到:

http://www.kernel.org/

也可以使用世界各地的很多镜像站点。

GCC 手册

所有 GNU C 编译器的发行版都应该包括介绍其特点的完整文档，这些文档被存放在几个 info 文件中，可以使用 Emacs 程序或者其他 info 阅读器进行读取。顺便说一下，有关扩充内联汇编（Extended Inline Assembly）的信息很难跟踪，因此这里并不是特指某一种体系结构。有关 Intel 80x86 的 GCC 内联汇编可以在这里找到:

http://sag-www.ssl.berkeley.edu/~korpela/djgpp_asm.html
http://www.castle.net/~avly/djasm.html

Linux 文档项目

站点位于:

http://www.linuxdoc.org

其中包含了 Linux 文档项目的主页，而该主页中又包含了一些有用的链接、指南、FAQ 及 HOWTO。

Linux 内核开发论坛

新闻组:

comp.os.linux.development.system

专门用于讨论有关 Linux 系统开发的问题。

Linux 内核邮件列表

这份邮件列表内容丰富，令人着迷，其中包括很多对 Linux 当前的开发版本的一些中肯的解释，也包括是否要在内核中包括一些基本原理的提议，同时也充斥着一些无关的内容。这是一个逐渐成型的新思想的一个孵化实验室。邮件列表名为:

linux-kernel@vger.rutgers.edu

Linux 内核在线书籍

由 David A. Rusling 所著，这本 200 页的书可以在这里在线阅读:

http://www.linuxdoc.org/LDP/tlk/tlk.html

该书介绍 Linux 内核的一些基本内容。

Linux 虚拟文件系统

主页位于:

http://www.atnf.csiro.au/~rgooch/linux/vfs.txt

这是对 Linux 虚拟文件系统的一个介绍。作者是 Richard Gooch。

源代码索引

D

E

G

N

S

U

V

W

X

Z

词汇表

abort
异常结束

address bus
地址总线

address space
地址空间

addressing
寻址

advisory lock
劝告锁

alignment factor
对齐因子

anonymous mapping
匿名映射

API (Application Programming Interface)
应用编程接口

asynchronous
异步

atomic operation
原子操作

base time quantum
基本时间片

batch process
批处理进程

BIOS (Basic Input/Output System)
基本输入/输出系统

bitmask
位掩码

block clustering
块聚簇

block device request
块设备请求

block device
块设备

block group
块组

blocking
阻塞

block-oriented device file
块设备文件

boot loader
引导装入程序

bootstrap
启动

bottom half
下半部分

buffer cache
缓冲区高速缓存

buffer head
缓冲区首部

buffer page
缓冲区页

bus address
总线地址

cache hit
高速缓存命中

cache miss
高速缓存不命中

cache
高速缓存

capability
能力

chained list
链接表

character device
字符设备

character-oriented device file
字符设备文件

coloring
着色

command-line argument
命令行参数

constructor
构造函数

control bus
控制总线

control register
控制寄存器

COW (Copy On Write)
写时复制

CPL (Current Privilege Level)
当前特权级

CPU-bound
CPU 范围

credential
信任状

critical region
临界区

custom I/O interface
专用 I/O 接口

data bus
数据总线

demand paging
请求调页

dentry cache
目录项高速缓存

dentry object
目录项对象

dentry operation
目录项操作

destructor
析构函数

device controller
设备控制器

device driver
设备驱动程序

device file
设备文件

device plugging
设备插入

device unlugging
设备拔出

direct invocation
直接调用

directory
目录

disk cache
磁盘高速缓存

dispatch table
分派表

DMA (Direct Memory Access)
直接内存访问

DMAC (Direct Memory Access Control)
直接内存访问控制器

doubly lined circular list
双向循环链表

DPL (Descriptor Privilege Level)
描述符特权级

dynamic memory
动态内存

dynamic priority (base priority)
动态优先级（基本优先级）

dynamic timer
动态定时器

ELF (Executable and Linking Format)
可执行和链接格式

environment variable
环境变量

epoch
时期

exception table
异常表

exception
异常

exclusive access
互斥访问

executable file
可执行文件

execution context
执行上下文

execution domain descriptor
执行域描述符

execution tracing
执行跟踪

Ext2 (Second Extended Filesystem)
第二文件扩展系统

external device
外部设备

external fragmentation
外碎片

FAT (File Allocation Table)
文件分配表

fault
故障

FIFO (First In, First Out)
先进先出

file control block
文件控制块

file descriptor
文件描述符

file object
文件对象

file operation
文件操作

filesystem control block
文件系统控制块

fixup code
修正代码

flag
标志

frame buffer
显存

GDT (Global Descriptor Table)
全局描述符表

general-purpose I/O interface
通用I/O接口

GID (Group ID)
组标识符

global variable
全局变量

hard link
硬链接

hardware context
硬件上下文

hash
散列

heap
堆

hidden scheduling
隐含调度

I/O APIC (I/O Advanced Programmable Interrupt Controller)
I/O高级可编程中断控制器

I/O port
I/O端口

I/O shared memory
I/O共享内存

I/O-bound
I/O范围

ICC (Interrupt Controller Communication)
中断控制器通信

ID (identifier)
标识符

IDT (Interrupt Descriptor Table)
中断描述符表

inode cache
索引节点高速缓存

inode object
索引节点对象

inode operation
索引节点操作

input register
输入寄存器

interactive process
交互式进程

internal device
内部设备

internal fragmentation
内碎片

interpreted script
解释脚本

interrupt disabling
关中断

interrupt enabling
开中断

interrupt gate
中断门

interrupt mode
中断模式

Interrupt Redirection Table
中断重定向表

interrupt
中断

interval timer
间隔定时器

IPC (Interprocess Communication)
进程间通信

IPC identifier
IPC 标识符

IPC key
IPC 关键字

IPC resource
IPC 资源

IPI (interprocessor interrupt)
处理器间中断

IRQ (Interrupt ReQuest)
中断请求

ISR (Interrupt Service Routine)
中断服务例程

JFS (Journaling File System)
日志文件系统

kernel
内核

kernel semaphore
内核信号量

kernel control path
内核控制路径

Kernel Mode
内核态

kernel symbol table
内核符号表

kernel thread
内核线程

KMA (Kernel Memory Allocator)
内核内存分配器

lazy invocation
松散调用

LDT (Local Descriptor Table)
局部描述符表

linear address
线性地址

local variable
局部变量

locality principle
局部性原理

locking
加锁

logical address
逻辑地址

login session
登录会话

LRU (Least Recently Used)
最近最少使用

LWP (lightweight process)
轻量级进程

magic number
魔数

major number
主号

mandatory locking
强制加锁

MBR (Master Boot Record)
主引导记录

memory address
内存地址

memory arbiter
内存仲裁器

memory area
内存区

memory mapping
内存映射

memory region
线性区

message queue
消息队列

microkernel
微内核

minor number
次号

minor page fault
次级缺页

MMU (Memory Management Unit)
内存管理单元

module symbol table
模块符号表

module
模块

mount point
安装点

named pipe
命名管道

nonblocking
非阻塞

nonpreemptive
非抢占的

N-way set associative
N路组关联

object file
目标文件

object
对象

output register
输出寄存器

page cache
页高速缓存

page frame
页框

Page Global Directory
页全局目录

Page Middle Directory
页中间目录

page slot
页插槽

page swap-in
页换入

page swap-out
页换出

page table
页表

paging unit
分页单元

PCI (Peripheral Component Interconnect)
外部设备互连

pending signal
挂起信号

permission bitmap
访问权位图

permission
许可权

physical address
物理地址

PID (Process ID)
进程标识符

pipe buffer
管道缓冲区

pipe size
管道大小

PIT (Programmable Interval Timer)
可编程间隔定时器

polling mode
轮询模式

preemptive
抢占的

primitive semaphore
原始信号量

priority inversion
优先级倒置

privilege
特权

procedure
过程

process
进程

process descriptor
进程描述符

process group
进程组

process list
进程链表

process switching
进程切换

program interpreter
程序解释器

protected mode
保护模式

quantum
时间片（时限）

queue of pending request
挂起请求队列

race condition
竞争条件

read lock
读锁

read-ahead group
预读组

read-ahead operation
预读操作

read-ahead window
预读窗口

read-ahead
预读

real mode
实模式

real-time process
实时进程

reentrant
可重入的

regular file
正规文件

request descriptor
请求描述符

request queue
请求队列

reuse list
重用链表

root filesystem
根文件系统

RTC (Real Time Clock)
实时时钟

scheduling policy
调度策略

SCSI (Small Computer System Interface)
小型计算机系统接口

segment descriptor
段描述符

segment selector
段选择符

segmentation register
段寄存器

segmentation unit
段单元

semaphore
信号量

shared library
共享库

shared memory segment
共享内存段

slot index
位置索引

slot usage sequence number
位置使用序号

SMP (Symmetric Multiprocessor)
对称多处理器

socket
套接字

software interrupt
软中断

source code
源代码

spin lock
自旋锁

static priority
静态优先级

static timer
静态定时器

status register
状态寄存器

strategy routine
策略程序

superblock object
超级块对象

superblock operation
超级块操作

superuser
超级用户

swap area
交换区

swap cache
交换高速缓存

swapping
交换

symbolic link
符号链接

synchronous
同步

system call
系统调用

system gate
系统门

tag
标签

task priority register
任务优先级寄存器

task queue
任务队列

tick
节拍

time-out
超时

timer interrupt
定时中断

time-sharing
分时

timestamp
时间标记

timing measurement
定时测量

TLB (Translation Lookaside Buffer)
转换后援缓冲器

top half
上半部分

trap gate
陷阱门

TSC (Time Stamp Counter)
时间标记计数器

TSS (Task State Segment)
任务状态段

UID (User ID)
用户标识符

USB (Universal Serial Bus)
通用串行总线

User Mode
用户态

VFS (Virtual Filesystem)
虚拟文件系统

virtual address space
虚拟地址空间

virtual memory
虚拟内存

wait queue
等待队列

wrapper routine
封装例程

write lock
写锁

write-back
写回

write-through
写透

zero page
零页

zombie process
僵死进程

深入理解

LINUX 内核

DANIEL P. BOVET & MARCO CESATI 著

陈莉君 冯锐 牛欣源 译

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Paris • Sebastopol • Taipei • Tokyo

O'Reilly & Associates, Inc. 授权中国电力出版社出版

中国电力出版社

UNDERSTANDING THE LINUX KERNEL

深入理解 LINUX 内核

O'REILLY®
中国电力出版社

DANIEL P. BOVET & MARCO CESATI 著
陈莉君 冯锐 牛欣源 译

O'Reilly & Associates公司介绍

为了满足读者对网络和软件技术知识的迫切需求，世界著名计算机图书出版机构O'Reilly & Associates公司授权中国电力出版社，翻译出版一批该公司久负盛名的英文经典技术专著。

O'Reilly & Associates公司是世界上在UNIX、X、Internet和其他开放系统图书领域具有领导地位的出版公司，同时是联机出版的先锋。

从最畅销的《The Whole Internet Use's Guide & Catalog》（被纽约公共图书馆评为二十世纪最重要的50本书之一）到GNN（最早的Internet门户和商业网站），再到WebSite（第一个桌面PC的Web服务器软件），O'Reilly & Associates一直处于Internet发展的最前沿。

许多书店的反馈表明，O'Reilly & Associates是最稳定的计算机图书出版商——每一本书都一版再版。与大多数计算机图书出版商相比，O'Reilly & Associates公司具有深厚的计算机专业背景，这使得O'Reilly & Associates形成了一个非常不同于其他出版商的出版方针。O'Reilly & Associates所有的编辑人员以前都是程序员，或者是顶尖级的技术专家。O'Reilly & Associates还有许多固定的作者群体——他们本身是相关领域的技术专家、咨询专家，而现在编写著作，O'Reilly & Associates依靠他们及时地推出图书。因为O'Reilly & Associates紧密地与计算机业界联系着，所以O'Reilly & Associates知道市场上真正需要什么图书。

译者序

如果说Linus Torvalds所开发的Linux操作系统把我们带入了一个自由而开放的世界，那么，本书作者Daniel P. Bovet和Marco Cesati奉献给大家的这本书将把我们带进一个知识的王国。

在操作系统这样一个庞大而复杂的软件面前，多少想探其究竟的人只能望洋兴叹。不仅因为其庞大，更因为其错综复杂的关系如走迷宫一样易进而难以走出。

Linux的出现带给我们的不仅是源码开放的喜悦，更重要的是你我都可以走进操作系统的内部世界。但是，开放的源码仅仅给我们敞开了大门，要想在其中探究计算机的各种软硬件机制到底是如何协调而有效的工作，还需要有盏照亮探索之路的灯。可以说，呈现在你面前的这本书就是你探索操作系统内部世界的一盏指路灯。

操作系统不仅是一个软件，而且是充分挖掘硬件潜能的软件。在本书第二章的内存寻址中，你可以找到Linux是如何充分发挥硬件的分页单元和分段单元机制的。中断是操作系统中发生最频繁的一个活动，本书用第四、第五两章篇幅讲述了与中断相关的主题。你不仅可以从对中断的一无所知，到切切实实地了解中断，还能对中断的内在运作机制以及在操作系统中的灵活应用有切实的感受。读完这两章，会有豁然开朗的感觉，原来中断并不神秘，我们可以真正地了解并掌握它。在本书共19章的内容中，你还会找到许多你所期望了解的知识。

如果你自己去读Linux的源码，绊脚石会时时挡住你，尤其是一些关键的代码段，可能让你裹足不前。本书对很多关键代码的逐行解释会使你消除不少疑虑。

本书虽然讲述的是Linux的源码，但你从中获得的知识是广泛而深入的。每章开始部分对一般性原理的描述打破了知识的局限性，而其中每个知识点都落到实处的独到分析，又会使你体验到知识被灵活应用的喜悦。知识惟有被灵活应用，方能显出其魅力。

阅读本书，需要一份耐心，更需要一份执着。当你闯过一道道难关，阅读到本书的最后一章时，会有“蓦然回首，那人却在灯火阑珊处”的感觉！

感谢中国电力出版社给了我们翻译这样一本好书的机会，感谢一些网友在翻译过程中提出的宝贵意见。

本书的第十一、十三、十四、十五、十六、十八章及附录由冯锐翻译，郭新贺、李小丁也参加了部分工作，第十二章由牛欣源翻译。全书由陈莉君统稿。

在本书的翻译过程中，深深体会出作者对知识的执着追求和对技术的精益求精，我们尽力去反映作者本身写作的意图，但因为中英文本身的差异，有些句子的理解难免有偏差，恳请读者提出宝贵的意见，反馈信息请递如下电子信箱：Linuxkernelb@263.net，译者将不胜感谢！

译 者

2001 年 6 月

图书在版编目（CIP）数据

深入理解Linux内核：/（美）博伟特（Bovet, D. P.）等编著；陈莉君等译．－北京：中国电力出版社，2001

书名原文：Understanding the Linux Kernel

ISBN 7-5083-0719-4

Ⅰ．深… Ⅱ．①博… ②陈… Ⅲ．LINUX操作系统 Ⅳ．TP316.81

中国版本图书馆CIP数据核字（2001）第060450号

北京市版权局著作权合同登记

图字：01-2001-3298号

书　　名/ 深入理解Linux内核

书　　号/ ISBN 7-5083-0719-4

责任编辑/ 夏平

封面设计/ Edie Freedman，张健

出版发行/ 中国电力出版社

地　　址/ 北京三里河路6号（邮政编码100044）

经　　销/ 全国新华书店

印　　刷/ 北京市地矿印刷厂

开　　本/ 787毫米×1092毫米　16开本　46.25印张　684千字

版　　次/ 2001年10月第一版　2001年10月第一次印刷

印　　数/ 0001-5000册

定　　价/ 79.00元（册）