

第17章 内存映射文件

对文件进行操作几乎是所有应用程序都必须进行的，并且这常常是人们争论的一个问题。应用程序究竟是应该打开文件，读取文件并关闭文件，还是打开文件，然后使用一种缓冲算法，从文件的各个不同部分进行读取和写入呢？Microsoft提供了一种两全其美的方法，那就是内存映射文件。

与虚拟内存一样，内存映射文件可以用来保留一个地址空间的区域，并将物理存储器提交给该区域。它们之间的差别是，物理存储器来自一个已经位于磁盘上的文件，而不是系统的页文件。一旦该文件被映射，就可以访问它，就像整个文件已经加载内存一样。

内存映射文件可以用于3个不同的目的：

- 系统使用内存映射文件，以便加载和执行 .exe和DLL文件。这可以大大节省页文件空间 and 应用程序启动运行所需的时间。
- 可以使用内存映射文件来访问磁盘上的数据文件。这使你可以不必对文件执行 I/O操作，并且可以不必对文件内容进行缓存。
- 可以使用内存映射文件，使同一台计算机上运行的多个进程能够相互之间共享数据。

Windows确实提供了其他一些方法，以便在进程之间进行数据通信，但是这些方法都是使用内存映射文件来实现的，这使得内存映射文件成为单个计算机上的多个进程互相进行通信的最有效的方法。

本章将要介绍内存映射文件的各种使用方法。

17.1 内存映射的可执行文件和DLL文件

当线程调用CreateProcess时，系统将执行下列操作步骤：

1) 系统找出在调用CreateProcess时设定的 .exe文件。如果找不到这个 .exe文件，进程将无法创建，CreateProcess将返回FALSE。

2) 系统创建一个新进程内核对象。

3) 系统为这个新进程创建一个私有地址空间。

4) 系统保留一个足够大的地址空间区域，用于存放该 .exe文件。该区域需要的位置在 .exe文件本身中设定。按照默认设置，.exe文件的基地址是0x00400000（这个地址可能不同于在64位Windows 2000上运行的64位应用程序的地址），但是，可以在创建应用程序的 .exe文件时重载这个地址，方法是在链接应用程序时使用链接程序的 /BASE选项。

5) 系统注意到支持已保留区域的物理存储器是在磁盘上的 .exe文件中，而不是在系统的页文件中。

当.exe文件被映射到进程的地址空间中之后，系统将访问 .exe文件的一个部分，该部分列出了包含 .exe文件中的代码要调用的函数的 DLL文件。然后，系统为每个 DLL文件调用LoadLibrary函数，如果任何一个DLL需要更多的DLL，那么系统将调用LoadLibrary函数，以便加载这些DLL。每当调用LoadLibrary来加载一个DLL时，系统将执行下列操作步骤，它们均类似上面的第4和第5个步骤：

1) 系统保留一个足够大的地址空间区域，用于存放该 DLL文件。该区域需要的位置在DLL文件本身中设定。按照默认设置，Microsoft的Visual C++建立的DLL文件基地址是

0x10000000（这个地址可能不同于在 64 位 Windows 2000 上运行的 64 位 DLL 的地址）但是，你可以在创建 DLL 文件时重载这个地址，方法是使用链接程序的 /BASE 选项。Windows 提供的所有标准系统 DLL 都拥有不同的基地址，这样，如果加载到单个地址空间，它们就不会重叠。

2) 如果系统无法在该 DLL 的首选基地址上保留一个区域，其原因可能是该区域已经被另一个 DLL 或 .exe 占用，也可能是因为该区域不够大，此时系统将设法寻找另一个地址空间的区域来保留该 DLL。如果一个 DLL 无法加载到它的首选基地址，这将是非常不利的，原因有二。首先，如果系统没有再定位信息，它就无法加载该 DLL（可以在 DLL 创建时，使用链接程序的 /FIXED 开关，从 DLL 中删除再定位信息，这能够使 DLL 变得比较小，但是这也意味着该 DLL 必须加载到它的首选地址中，否则它就根本无法加载）。第二，系统必须在 DLL 中执行某些再定位操作。在 Windows 98 中，系统可以在页面被转入 RAM 时执行再定位操作。在 Windows 2000 中，这些再定位操作需要由系统的页文件提供更多的存储器，它们也增加了加载 DLL 所需要的时间量。

3) 系统会注意到支持已保留区域的物理存储器位于磁盘上的 DLL 文件中，而不是在系统的页文件中。如果由 DLL 无法加载到它的首选基地址，Windows 2000 必须执行再定位操作，那么系统也将注意到 DLL 的某些物理存储器已经被映射到页文件中。

如果由于某个原因系统无法映射 .exe 和所有必要的 DLL 文件，那么系统就会向用户显示一个消息框，并且释放进程的地址空间和进程对象。CreateProcess 函数将向调用者返回 FALSE，调用者可以调用 GetLastError 函数，以便更好地了解为什么无法创建该进程。

当所有的 .exe 和 DLL 文件都被映射到进程的地址空间之后，系统就可以开始执行 .exe 文件的启动代码。当 .exe 文件被映射后，系统将负责所有的分页、缓冲和高速缓存的处理。例如，如果 .exe 文件中的代码使它跳到一个尚未加载到内存的指令地址，那么就会出现一个错误。系统能够发现这个错误，并且自动将这页代码从该文件的映像加载到一个 RAM 页面。然后，系统将这个 RAM 页面映射到进程的地址空间中的相应位置，并且让线程继续运行，就像这页代码已经加载了一样。当然，这一切是应用程序看不见的。当进程中的线程每次试图访问尚未加载到 RAM 的代码或数据时，该进程就会重复执行。

17.1.1 可执行文件或 DLL 的多个实例不能共享静态数据

当为正在运行的应用程序创建新进程时，系统将打开用于标识可执行文件映像的文件映射对象的另一个内存映射视图，并创建一个新进程对象和（为主线程创建）一个新线程对象。系统还要将新的进程 ID 和线程 ID 赋予这些对象。通过使用内存映射文件，同一个应用程序的多个正在运行的实例就能够共享 RAM 中的相同代码和数据。

这里有一个小问题需要注意。进程使用的是一个平面地址空间。当编译和链接你的程序时，所有的代码和数据都被合并在一起，组成一个很大的结构。数据与代码被分开，但仅限于跟在 .exe 文件中的代码后面的数据而已^①。图 17-1 简单说明了应用程序的代码和数据究竟是如何加载到虚拟内存中，然后又被映射到应用程序的地址空间中的。

作为一个例子，假设应用程序的第二个实例正在运行。系统只是将包含文件的代码和数据的虚拟内存页面映射到第二个应用程序的地址空间，如图 17-2 所示。

如果应用程序的一个实例改变了驻留在数据页面中的某些全局变量，那么该应用程序的所有实例的内存内容都将改变。这种类型的改变可能带来灾难性的后果，因此是决不允许的。

^① 实际上，文件的内容被分割为不同的节。代码放在一个节中，全局变量放在另一个节中。各个节按照页面边界来对齐。通过调用 GetSystemInfo 函数，应用程序可以确定正在使用的页面的大小。在 .exe 或 DLL 文件中，代码节通常位于数据数据节的前面。

系统运用内存管理系统的 copy-on-write（写入时拷贝）特性来防止进行这种改变。每当应用程序尝试将数据写入它的内存映射文件时，系统就会抓住这种尝试，为包含应用程序尝试写入数据的内存页面分配一个新内存块，再拷贝该页面的内容，并允许该应用程序将数据写入这个新分配的内存块。结果，同一个应用程序的所有其他实例的运行都不会受到影响。图 17-3 显示了当应用程序的第一个实例尝试改变数据页面 2 时出现的情况。

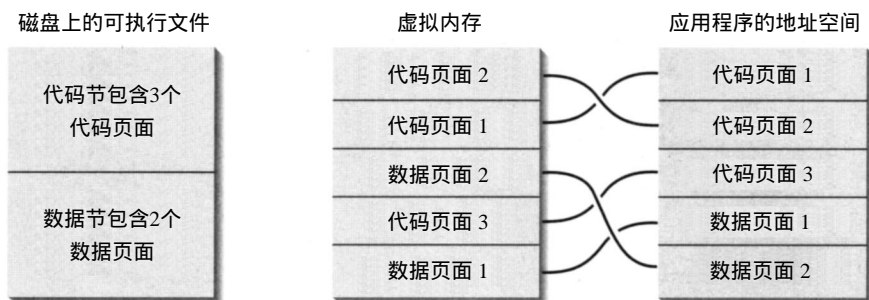


图 17-1 应用程序的代码和数据加载及映射示意图

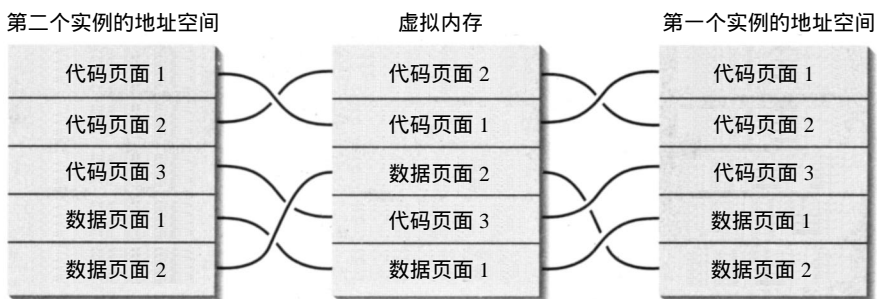


图 17-2 应用程序与虚拟内存地址空间之间的关系示意图



图 17-3 应用程序的第一个实例尝试改变数据页面 2 时的情况

系统分配一个新的虚拟内存页面，并且将数据页面 2 的内容拷贝到新页面中。第一个实例的地址空间发生了变更，这样，新数据页面就被映射到与原始地址页面相同位置上的地址空间中。这时系统就可以让进程修改全局变量，而不必担心改变同一个应用程序的另一个实例的数据。

当应用程序被调试时，将会发生类似的事件。比如说，你正在运行一个应用程序的多个实例，并且只想调试其中的一个实例。你访问调试程序，在一行源代码中设置一个断点。调试程序修改了你的代码，将你的一个汇编语言指令改为能使调试程序自行激活的指令。因此你再次

遇到了同样的问题。当调试程序修改代码时，它将导致应用程序的所有实例在修改后的汇编语言指令运行时激活该调试程序。为了解决这个问题，系统再次使用 copy-on-write 内存。当系统发现调试程序试图修改代码时，它就分配一个新内存块，将包含该指令的页面拷贝到新的内存页面中，并且允许调试程序修改页面拷贝中的代码。

Windows 98 当一个进程被加载时，系统要查看文件映像的所有页面。系统立即为通常用 copy-on-write 属性保护的那些页面提交页文件中的存储器。这些页面只是被提交而已，它们并不被访问。当文件映像中的页面被访问时，系统就加载相应的页面。如果该页面从来没有被修改，它就可以从内存中删除，并在必要时重新加载。但是，如果文件的页面被修改了，系统就将修改过的页面转到页文件中以前被提交的页面之一。

Windows 2000 与 Windows 98 之间的行为特性的唯一差别，是在你加载一个模块的两个拷贝并且可写入的数据尚未被修改的时候显示出来的。在这种情况下，在 Windows 2000 下运行的进程能够共享数据，而在 Windows 98 下，每个进程都可以得到它自己的数据拷贝。如果只加载模块的一个拷贝，或者可写入的数据已经被修改（这是通常的情况），那么 Windows 2000 与 Windows 98 的行为特性是完全相同的。

17.1.2 在可执行文件或 DLL 的多个实例之间共享静态数据

全局数据和静态数据不能被同一个 .exe 或 DLL 文件的多个映像共享，这是个安全的默认设置。但是，在某些情况下，让一个 .exe 文件的多个映像共享一个变量的实例是非常有用和方便的。例如，Windows 没有提供任何简便的方法来确定用户是否在运行应用程序的多个实例。但是，如果能够让所有实例共享单个全局变量，那么这个全局变量就能够反映正在运行的实例的数量。当用户启动应用程序的一个实例时，新实例的线程能够简单地查看全局变量的值（它已经被另一个实例更新）；如果这个数量大于 1，那么第二个实例就能够通知用户，该应用程序只有一个实例可以运行，而第二个实例将终止运行。

本节将介绍一种方法，它允许你共享 .exe 或 DLL 文件的所有实例的变量。不过在介绍这个方法之前，首先让我们介绍一些背景知识。

每个 .exe 或 DLL 文件的映像都由许多节组成。按照规定，每个标准节的名字均以圆点开头。例如，当编译你的程序时，编译器会将所有代码放入一个名叫 .text 的节中。该编译器还将所有未经初始化的数据放入一个 .bss 节，而已经初始化的所有数据则放入 .data 节中。

每一节都拥有与其相关的一组属性，这些属性如表 17-1 所示。

表 17-1 .exe 或 DLL 文件各节的属性

属 性	含 义
READ	该节中的字节可以读取
WRITE	该节中的字节可以写入
EXECUTE	该节中的字节可以执行
SHARED	该节中的字节可以被多个实例共享（本属性能够有效地关闭 copy-on-write 机制）

使用 Microsoft 的 Visual Studio 的 DumpBin 实用程序（带有 /Headers 开关），可以查看 .exe 或 DLL 映射文件中各个节的列表。下面选录的代码是在一个可执行文件上运行 DumpBin 程序而生成的：

```
SECTION HEADER #1
.text name
```

```
11A70 virtual size
1000 virtual address
12000 size of raw data
1000 file pointer to raw data
0 file pointer to relocation table
0 file pointer to line numbers
0 number of relocations
0 number of line numbers
60000020 flags
Code
Execute Read
```

SECTION HEADER #2

```
.rdata name
1F6 virtual size
13000 virtual address
1000 size of raw data
13000 file pointer to raw data
0 file pointer to relocation table
0 file pointer to line numbers
0 number of relocations
0 number of line numbers
40000040 flags
Initialized Data
Read Only
```

SECTION HEADER #3

```
.data name
560 virtual size
14000 virtual address
1000 size of raw data
14000 file pointer to raw data
0 file pointer to relocation table
0 file pointer to line numbers
0 number of relocations
0 number of line numbers
C0000040 flags
Initialized Data
Read Write
```

SECTION HEADER #4

```
.idata name
58D virtual size
15000 virtual address
1000 size of raw data
15000 file pointer to raw data
0 file pointer to relocation table
0 file pointer to line numbers
0 number of relocations
0 number of line numbers
C0000040 flags
Initialized Data
Read Write
```

SECTION HEADER #5

```
.didat name
```

```

    7A2 virtual size
16000 virtual address
    1000 size of raw data
16000 file pointer to raw data
    0 file pointer to relocation table
    0 file pointer to line numbers
    0 number of relocations
    0 number of line numbers
C0000040 flags
    Initialized Data
    Read Write
SECTION HEADER #6
.reloc name
    26D virtual size
17000 virtual address
    1000 size of raw data
17000 file pointer to raw data
    0 file pointer to relocation table
    0 file pointer to line numbers
    0 number of relocations
    0 number of line numbers
42000040 flags
    Initialized Data
    Discardable
    Read Only

Summary
    1000 .data
    1000 .didat
    1000 .idata
    1000 .rdata
    1000 .reloc
    12000 .text

```

表17-2显示了比较常见的一些节的名字，并且说明了每一节的作用。

除了编译器和链接程序创建的标准节外，也可以在使用下面的命令进行编译时创建自己的节：

表17-2 常见的节名及作用

节 名	作 用
.bss	未经初始化的数据
.CRT	C运行期只读数据
.data	已经初始化的数据
.debug	调试信息
.didata	延迟输入文件名表
.edata	输出文件名表
.idata	输入文件名表
.rdata	运行期只读数据
.reloc	重定位表信息
.rsrc	资源
.text	.exe或DLL文件的代码
.tls	线程的本地存储器
.xdata	异常处理表

```
#pragma data_seg("sectionname")
```

我可以创建一个称为“Shared”的节，它包含单个LONG值，如下所示：

```
#pragma data_seg("Shared")
LONG g_lInstanceCount = 0;
#pragma data_seg()
```

当编译器对这个代码进行编译时，它创建一个新节，称为Shared，并将它在编译指示后面看到的所有已经初始化（initialized）的数据变量放入这个新节中。在上面这个例子中，变量放入Shared节中。该变量后面的#pragma dataseg()一行告诉编译器停止将已经初始化的变量放入Shared节，并且开始将它们放回到默认数据节中。需要记住的是，编译器只将已经初始化的变量放入新节中。例如，如果我从前面的代码段中删除初始化变量（如下面的代码所示），那么编译器将把该变量放入Shared节以外的节中。

```
#pragma data_seg("Shared")
LONG g_lInstanceCount;
#pragma data_seg()
```

Microsoft 的Visual C++编译器提供了一个Allocate说明符，使你可以将未经初始化的数据放入你希望的任何节中。请看下面的代码：

```
// Create Shared section & have compiler place initialized data in it.
#pragma data_seg("Shared")

// Initialized, in Shared section
int a = 0;

// Uninitialized, not in Shared section
int b;

// Have compiler stop placing initialized data in Shared section.
#pragma data_seg()

// Initialized, in Shared section
__declspec(allocate("Shared")) int c = 0;
// Uninitialized, in Shared section
__declspec(allocate("Shared")) int d;

// Initialized, not in Shared section
int e = 0;

// Uninitialized, not in Shared section
int f;
```

上面的注释清楚地指明了指定的变量将被放入哪一节。若要使Allocate声明的规则正确地起作用，那么首先必须创建节。如果删除前面这个代码中的第一行#pragma data_seg，上面的代码将不进行编译。

之所以将变量放入它们自己的节中，最常见的原因也许是要在.exe或DLL文件的多个映像之间共享这些变量。按照默认设置，.exe或DLL文件的每个映像都有它自己的一组变量。然而，可以将你想在该模块的所有映像之间共享的任何变量组合到它自己的节中去。当给变量分组时，系统并不为.exe或DLL文件的每个映像创建新实例。

仅仅告诉编译器将某些变量放入它们自己的节中，是不足以实现对这些变量的共享的。还

必须告诉链接程序，某个节中的变量是需要加以共享的。若要进行这项操作，可以使用链接程序的命令行上的/SECTION开关：

```
/SECTION:name,attributes
```

在冒号的后面，放入你想要改变其属性的节的名字。在我们的例子中，我们想要改变Shared节的属性。因此应该创建下面的链接程序开关：

```
/SECTION:Shared,RWS
```

在逗号的后面，我们设定了需要的属性。用 R代表READ，W代表WRITE，E代表EXECUTE，S代表SHARED。上面的开关用于指明位于Shared节中的数据是可以读取、写入和共享的数据。如果想要改变多个节的属性，必须多次设定 /SECTION开关，也就是为你要改变属性的每个节设定一个/SECTION开关。

也可以使用下面的句法将链接程序开关嵌入你的源代码中：

```
#pragma comment(linker, "/SECTION:Shared,RWS")
```

这一行代码告诉编译器将上面的字符串嵌入名字为“.drectve”的节。当链接程序将所有的.obj模块组合在一起时，链接程序就要查看每个.obj模块的“.drectve”节，并且规定所有的字符串均作为命令行参数传递给该链接程序。我一直使用这种方法，因为它非常方便。如果将源代码文件移植到一个新项目中，不必记住在 Visual C++的Project Settings（项目设置）对话框中设置链接程序开关。

虽然可以创建共享节，但是，由于两个原因，Microsoft并不鼓励你使用共享节。第一，用这种方法共享内存有可能破坏系统的安全。第二，共享变量意味着一个应用程序中的错误可能影响另一个应用程序的运行，因为它没有办法防止某个应用程序将数据随机写入一个数据块。

假设你编写了两个应用程序，每个应用程序都要求用户输入一个口令。然而你又决定给应用程序添加一些特性，使用户操作起来更加方便些：如果在第二个应用程序启动运行时，用户正在运行其中的一个应用程序，那么第二个应用程序就可以查看共享内存的内容，以便获得用户的口令。这样，如果程序中的某一个已经被使用，那么用户就不必重新输入他的口令。

这听起来没有什么问题。毕竟没有别的应用程序而只有你自己的应用程序加载了DLL，并且知道到什么地方去查找包含在共享节中的口令。但是，黑客正在窥视着你的行动，如果他们想要得到你的口令，只需要编写一段很短的程序，加载到你的公司的DLL文件中，然后监控共享内存块。当用户输入口令时，黑客的程序就能知道该用户的口令。

黑客精心编制的程序也可能试图反复猜测用户的口令并将它们写入共享内存。一旦该程序猜测到正确的口令，它就能够将各种命令发送给两个应用程序中的一个。如果有一种办法只为某些应用程序赋予访问权，以便加载一个特定的DLL，那么这个问题也许是可以解决的。但是目前还不行，因为任何程序都能够调用LoadLibrary函数来显式加载DLL。

17.1.3 AppInst示例应用程序

清单17-1列出的AppInst示例应用程序（“17 AppInst.exe”）显示了应用程序如何能够知道每次有多少个应用程序的实例正在运行。该应用程序的源代码和资源文件位于本书所附光盘上的17-AppInst目录下。当运行AppInst程序时，就会出现它的对话框（见图17-4），指明该应用程序的一个实例正在运行。

如果运行该应用程序的第二个实例，那么第一和第二个实例的对话框都会发生变化，以反映目前两个实例都在运行（见图17-5，图17-6）。

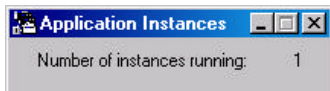


图17-4 运行AppInst时出现的对话框



图17-5 运行 AppInst 的第二个实例时，
第一个实例对话框的变化

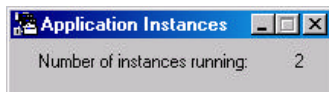


图17-6 运行 AppInst 的第二个实例时，
第二个实例对话框的变化

可以根据你的喜好，运行和撤消任意多个实例，实例的数量始终都能正确地反映在仍然保留的实例中。

在靠近AppInst.cpp应用程序的顶部，可以看到下面的代码行：

```
// Tell the compiler to put this initialized variable in its own
// section so it is shared by all instances of this application.
#pragma data_seg("Shared")
volatile LONG g_lApplicationInstances = 0;
#pragma data_seg()

// Tell the linker to make the Shared section
// readable, writable, and shared.
#pragma comment(linker, "/Section:Shared,RWS")
```

这些代码行用于创建一个称为 Shared 的节，该节拥有读取、写入和共享保护属性。在这个节中，有一个变量是 g_lApplicationInstances。该应用程序的所有实例均可以共享该变量。注意，该变量是个易失性变量，因此优化程序对我们不起多大的作用。

当每个实例的_tWinMain函数执行时，g_lApplicationInstances变量就递增1。在_tWinMain退出之前，该变量将递减1。我使用InterlockedExchangeAdd来改变这个变量，因为多个线程将要访问该共享资源。

当每个实例的对话框出现时，Dlg_OnInitDialog函数就被调用。该函数将一个注册窗口消息广播发送到所有的高层窗口（该消息的ID包含在g_aMsgAppInstCountUpdate变量中）：

```
PostMessage(HWND_BROADCAST, g_aMsgAppInstCountUpdate, 0, 0);
```

系统中的所有窗口将忽略这个注册窗口消息，但 AppInst的各个窗口例外。当我们的各个窗口中的一个接收到该消息时，Dlg_Proc中的代码将更新该对话框中的实例数量，以反映当前的实例数量（该数量在g_lApplicationInstances共享变量中进行维护）。

清单17-1 AppInst示例应用程序



AppInst.cpp

```
/*
Module: AppInst.cpp
Notices: Copyright (c) 2000 Jeffrey Richter
*/
```

```
#include "..\CmnHdr.h" /* See Appendix A. */
#include <windowsx.h>
#include <tchar.h>
#include "Resource.h"
```

```
////////////////////////////////////
```

```
// The system-wide unique window message
UINT g_uMsgAppInstCountUpdate = INVALID_ATOM;

////////////////////////////////////

// Tell the compiler to put this initialized variable in its own Shared
// section so it is shared by all instances of this application.
#pragma data_seg("Shared")
volatile LONG g_lApplicationInstances = 0;
#pragma data_seg()

// Tell the linker to make the Shared section readable, writable, and shared.
#pragma comment(linker, "/Section:Shared,RWS")

////////////////////////////////////

BOOL Dlg_OnInitDialog(HWND hwnd, HWND hwndFocus, LPARAM lParam) {

    chSETDLGICONS(hwnd, IDI_APPINST);

    // Force the static control to be initialized correctly.
    PostMessage(HWND_BROADCAST, g_uMsgAppInstCountUpdate, 0, 0);
    return(TRUE);
}

////////////////////////////////////

void Dlg_OnCommand(HWND hwnd, int id, HWND hwndCtl, UINT codeNotify) {

    switch (id) {
        case IDCANCEL:
            EndDialog(hwnd, id);
            break;
    }
}

////////////////////////////////////

INT_PTR WINAPI Dlg_Proc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {

    if (uMsg == g_uMsgAppInstCountUpdate) {
        SetDlgItemInt(hwnd, IDC_COUNT, g_lApplicationInstances, FALSE);
    }

    switch (uMsg) {
        case WM_INITDIALOG:
            Dlg_OnInitDialog(hwnd, hwnd, 0);
        case WM_COMMAND:
            Dlg_OnCommand(hwnd, LOWORD(wParam), HIWORD(wParam), LOWORD(wParam));
    }

    return(FALSE);
}
```

```
////////////////////////////////////
```

```
int WINAPI _tWinMain(HINSTANCE hinstExe, HINSTANCE, LPTSTR pszCmdLine, int) {

    // Get the numeric value of the systemwide window message used to notify
    // all top-level windows when the module's usage count has changed.
    g_uMsgAppInstCountUpdate =
        RegisterWindowMessage(TEXT("MsgAppInstCountUpdate"));

    // There is another instance of this application running.
    InterlockedExchangeAdd((PLONG) &g_lApplicationInstances, 1);

    DialogBox(hinstExe, MAKEINTRESOURCE(IDD_APPINST), NULL, Dlg_Proc);
    // This instance of the application is terminating.
    InterlockedExchangeAdd((PLONG) &g_lApplicationInstances, -1);

    // Have all other instances update their display.
    PostMessage(HWND_BROADCAST, g_uMsgAppInstCountUpdate, 0, 0);

    return(0);
}
```

```
//////////////////////////////////// End of File //////////////////////////////////////
```

Applnst.rc

```
//Microsoft Developer Studio generated resource script.
//
#include "Resource.h"

#define APSTUDIO_READONLY_SYMBOLS
////////////////////////////////////
//
// Generated from the TEXTINCLUDE 2 resource.
//
#include "afxres.h"

////////////////////////////////////
#undef APSTUDIO_READONLY_SYMBOLS

////////////////////////////////////
// English (U.S.) resources

#if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_ENU)
#ifdef _WIN32
LANGUAGE LANG_ENGLISH, SUBLANG_ENGLISH_US
#pragma code_page(1252)
#endif // _WIN32

#ifdef APSTUDIO_INVOKED
////////////////////////////////////
```

```
//
// TEXTINCLUDE
//
1 TEXTINCLUDE DISCARDABLE
BEGIN
    "Resource.h\0"
END

2 TEXTINCLUDE DISCARDABLE
BEGIN
    "#include ""afxres.h""\r\n"
    "\0"
END

3 TEXTINCLUDE DISCARDABLE
BEGIN
    "\r\n"
    "\0"
END

#endif    // APSTUDIO_INVOKED

////////////////////////////////////
//
// Dialog
//

IDD_APPINST_DIALOG DISCARDABLE 0, 0, 140, 21
STYLE WS_MINIMIZEBOX | WS_VISIBLE | WS_CAPTION | WS_SYSMENU
CAPTION "Application Instances"
FONT 8, "MS Sans Serif"
BEGIN
    LTEXT                "Number of instances running:", IDC_STATIC, 12, 4, 93, 8,
                        SS_NOPREFIX
    RTEXT                "#", IDC_COUNT, 112, 4, 16, 12, SS_NOPREFIX
END

////////////////////////////////////
//
// Icon
//

// Icon with lowest ID value placed first to ensure application icon
// remains consistent on all systems.
IDI_APPINST            ICON    DISCARDABLE    "AppInst.Ico"
////////////////////////////////////
//
// DESIGNINFO
//

#ifdef APSTUDIO_INVOKED
GUIDELINES DESIGNINFO DISCARDABLE
BEGIN
```

```

IDD_APPINST, DIALOG
BEGIN
    RIGHTMARGIN, 76
    BOTTOMMARGIN, 20
END
#endif // APSTUDIO_INVOKED

#endif // English (U.S.) resources
////////////////////////////////////

#ifndef APSTUDIO_INVOKED
////////////////////////////////////
//
// Generated from the TEXTINCLUDE 3 resource.
//
////////////////////////////////////
#endif // not APSTUDIO_INVOKED

```

17.2 内存映射数据文件

操作系统使得内存能够将一个数据文件映射到进程的地址空间中。因此，对大量的数据进行操作是非常方便的。

为了理解用这种方法来使用内存映射文件的功能，让我们看一看如何用 4 种方法来实现一个程序，以便将文件中的所有字节的顺序进行倒序。

17.2.1 方法1：一个文件，一个缓存

第一种方法也是理论上最简单的方法，它需要分配足够大的内存块来存放整个文件。该文件被打开，它的内容被读入内存块，然后该文件被关闭。文件内容进入内存后，我们就可以对所有字节的顺序进行倒序，方法是将第一个字节倒腾为最后一个字节，第二个字节倒腾为倒数第二个字节，依次类推。这个倒腾操作将一直进行下去直到文件的中间位置。当所有的字节都已经倒腾之后，就可以重新打开该文件，并用内存块的内容来改写它的内容。

这种方法实现起来非常容易，但是它有两个缺点。首先，必须分配一个与文件大小相同的内存块。如果文件比较小，那么这没有什么问题。但是如果文件非常大，比如说有 2GB 大，那该怎么办呢？一个 32 位的系统不允许应用程序提交那么大的物理内存块。因此大文件需要使用不同的方法。

第二，如果进程在运行过程的中间被中断，也就是说当倒序后的字节被重新写入该文件时进程被中断，那么文件的内容就会遭到破坏。防止出现这种情况的最简单的方法是在对它的内容进行倒序之前先制作一个原始文件的拷贝。如果整个进程运行成功，那么可以删除该文件的拷贝。这种方法需要更多的磁盘空间。

17.2.2 方法2：两个文件，一个缓存

在第二种方法中，你打开现有的文件，并且在磁盘上创建一个长度为 0 的新文件。然后分

配一个比较小的内部缓存，比如说 8 KB。你找到离原始文件结尾还有 8 KB 的位置，将这最后的 8 KB 读入缓存，将字节倒序，再将缓存中的内容写入新创建的文件。这个寻找、读入、倒序和写入的操作过程要反复进行，直到到达原始文件的开头。如果文件的长度不是 8 KB 的倍数，那么必须进行某些特殊的处理。当原始文件完全处理完毕之后，将原始文件和新文件关闭，并删除原始文件。

这种方法实现起来比第一种方法要复杂一些。它对内存的使用效率要高得多，因为它只需要分配一个 8 KB 的缓存块，但是它存在两个大问题。首先，它的处理速度比第一种方法要慢，原因是在每个循环操作过程中，在执行读入操作之前，必须对原始文件进行寻找操作。第二，这种方法可能要使用大量的硬盘空间。如果原始文件是 400 MB，那么随着进程的不断运行，新文件就会增大为 400 MB。在原始文件被删除之前，两个文件总共需要占用 800 MB 的磁盘空间。这比应该需要的空间大 400 MB。由于存在这个缺点，因此引来了下一个方法。

17.2.3 方法3：一个文件，两个缓存

如果使用这个方法，那么我们假设程序初始化时分配了两个独立的 8 KB 缓存。程序将文件的第一个 8 KB 读入一个缓存，再将文件的第二个 8 KB 读入另一个缓存。然后进程将两个缓存的内容进行倒序，并将第一个缓存的内容写回文件的结尾处，将第二个缓存的内容写回同一个文件的开始处。每个迭代操作不断进行（以 8 KB 为单位，从文件的开始和结尾处移动文件块）。如果文件的长度不是 16 KB 的倍数，并且有两个 8 KB 的文件块相重叠，那么就需要进行一些特殊的处理。这种特殊处理比上一种方法中的特殊处理更加复杂，不过这难不倒经验丰富的程序员。

与前面的两种方法相比，这种方法在节省硬盘空间方面有它的优点。由于所有内容都是从同一个文件读取并写入同一个文件，因此不需要增加额外的磁盘空间，至于内存的使用，这种方法也不错，它只需要使用 16 KB 的内存。当然，这种方法也许是最难实现的方法。与第一种方法一样，如果进程被中断，本方法会导致数据文件被破坏。

下面我们来看一看如何使用内存映射文件来完成这个过程。

17.2.4 方法4：一个文件，零缓存

当使用内存映射文件对文件内容进行倒序时，你打开该文件，然后告诉系统将虚拟地址空间的一个区域进行倒序。你告诉系统将文件的第一个字节映射到该保留区域的第一个字节。然后可以访问该虚拟内存的区域，就像它包含了这个文件一样。实际上，如果在文件的结尾处有一个单个 0 字节，那么只需要调用 C 运行期函数 `_strrev`，就可以对文件中的数据进行倒序操作。

这种方法的优点是，系统能够为你管理所有的文件缓存操作。不必分配任何内存，或者将文件数据加载到内存，也不必将数据重新写入该文件，或者释放任何内存块。但是，内存映射文件仍然可能出现因为电源故障之类的进程中断而造成数据被破坏的问题。

17.3 使用内存映射文件

若要使用内存映射文件，必须执行下列操作步骤：

- 1) 创建或打开一个文件内核对象，该对象用于标识磁盘上你想用作内存映射文件的文件。
- 2) 创建一个文件映射内核对象，告诉系统该文件的大小和你打算如何访问该文件。
- 3) 让系统将文件映射对象的全部或一部分映射到你的进程地址空间中。

当完成对内存映射文件的使用时，必须执行下面这些步骤将它清除：

- 1) 告诉系统从你的进程的地址空间中撤消文件映射内核对象的映像。
 - 2) 关闭文件映射内核对象。
 - 3) 关闭文件内核对象。
- 下面将详细介绍这些操作步骤。

17.3.1 步骤1：创建或打开文件内核对象

若要创建或打开一个文件内核对象，总是要调用 CreateFile 函数：

```
HANDLE CreateFile(  
    PCSTR pszFileName,  
    DWORD dwDesiredAccess,  
    DWORD dwShareMode,  
    PSECURITY_ATTRIBUTES psa,  
    DWORD dwCreationDisposition,  
    DWORD dwFlagsAndAttributes,  
    HANDLE hTemplateFile);
```

CreateFile 函数拥有好几个参数。这里只重点介绍前 3 个参数，即 pszFileName，dwDesiredAccess 和 dwShareMode。

你可能会猜到，第一个参数 pszFileName 用于指明要创建或打开的文件的名字（包括一个选项路径）。第二个参数 dwDesiredAccess 用于设定如何访问该文件的内容。可以设定表 17-3 所列的 4 个值中的一个。

表17-3 dwDesiredAccess 的值

值	含 义
0	不能读取或写入文件的内容。当只想获得文件的属性时，请设定 0
GENERIC_READ	可以从文件中读取数据
GENERIC_WRITE	可以将数据写入文件
GENERIC_READ GENERIC_WRITE	可以从文件中读取数据，也可以将数据写入文件

当创建或打开一个文件，将它作为一个内存映射文件来使用时，请选定最有意义的一个或多个访问标志，以说明你打算如何访问文件的数据。对内存映射文件来说，必须打开用于只读访问或读写访问的文件，因此，可以分别设定 GENERIC_READ 或 GENERIC_READ | GENERIC_WRITE。

第三个参数 dwShareMode 告诉系统你想如何共享该文件。可以为 dwShareMode 设定表 17-4 所列的 4 个值之一。

表17-4 dwShareMode 的值

值	含 义
0	打开文件的任何尝试均将失败
FILE_SHARE_READ	使用 GENERIC_WRITE 打开文件的其他尝试将会失败
FILE_SHARE_WRITE	使用 GENERIC_READ 打开文件的其他尝试将会失败
FILE_SHARE_READ FILE_SHARE_WRITE	打开文件的其他尝试将会取得成功

如果 CreateFile 函数成功地创建或打开指定的文件，便返回一个文件内核对象的句柄，否则返回 INVALID_HANDLE_VALUE。

注意 能够返回句柄的大多数 Windows 函数如果运行失败，那么就会返回 NULL。但是，CreateFile 函数将返回 INVALID_HANDLE_VALUE，它定义为 $((\text{HANDLE}) - 1)$ 。

17.3.2 步骤2：创建一个文件映射内核对象

调用 CreateFile 函数，就可以将文件映像的物理存储器的位置告诉操作系统。你传递的路径名用于指明支持文件映像的物理存储器在磁盘（或网络或光盘）上的确切位置。这时，必须告诉系统，文件映射对象需要多少物理存储器。若要进行这项操作，可以调用 CreateFileMapping 函数：

```
HANDLE CreateFileMapping(  
    HANDLE hFile,  
    PSECURITY_ATTRIBUTES psa,  
    DWORD fdwProtect,  
    DWORD dwMaximumSizeHigh,  
    DWORD dwMaximumSizeLow,  
    PCTSTR pszName);
```

第一个参数 hFile 用于标识你想要映射到进程地址空间中的文件句柄。该句柄由前面调用的 CreateFile 函数返回。psa 参数是指向文件映射内核对象的 SECURITY_ATTRIBUTES 结构的指针，通常传递的值是 NULL（它提供默认的安全特性，返回的句柄是不能继承的）。

本章开头讲过，创建内存映射文件就像保留一个地址空间区域然后将物理存储器提交给该区域一样。因为内存映射文件的物理存储器来自磁盘上的一个文件，而不是来自从系统的页文件中分配的空间。当创建一个文件映射对象时，系统并不为它保留地址空间区域，也不将文件的存储器映射到该区域（下一节将介绍如何进行这项操作）。但是，当系统将存储器映射到进程的地址空间中去时，系统必须知道应该将什么保护属性赋予物理存储器的页面。CreateFileMapping 函数的 fdwProtect 参数使你能够设定这些保护属性。大多数情况下，可以设定表 17-5 中列出的 3 个保护属性之一。

表 17-5 使用 fdwProtect 参数设定的部分保护属性

保护属性	含义
PAGE_READONLY	当文件映射对象被映射时，可以读取文件的数据。必须已经将 GENERIC_READ 传递给 CreateFile 函数
PAGE_READWRITE	当文件映射对象被映射时，可以读取和写入文件的数据。必须已经将 GENERIC_READ GENERIC_WRITE 传递给 CreateFile
PAGE_WRITECOPY	当文件映射对象被映射时，可以读取和写入文件的数据。如果写入数据，会导致页面的私有拷贝得以创建。必须已经将 GENERIC_READ 或 GENERIC_WRITE 传递给 CreateFile

Windows 98 在 Windows 98 下，可以将 PAGE_WRITECOPY 标志传递给 CreateFile Mapping，这将告诉系统从页文件中提交存储器。该页文件存储器是为数据文件的数据拷贝保留的，只有修改过的页面才被写入页文件。你对该文件的数据所作的任何修改都不会重新填入原始数据文件。其最终结果是，PAGE_WRITECOPY 标志的作用在 Windows 2000 和 Windows 98 上是相同的。

除了上面的页面保护属性外，还有 4 个节保护属性，你可以用 OR 将它们连接起来放入 CreateFileMapping 函数的 fdwProtect 参数中。节只是用于内存映射的另一个术语。

节的第一个保护属性是 SEC_NOCACHE，它告诉系统，没有将文件的任何内存映射页面放入高速缓存。因此，当将数据写入该文件时，系统将更加经常地更新磁盘上的文件数据。这个标志与 PAGE_NOCACHE 保护属性标志一样，是供设备驱动程序开发人员使用的，应用程序通常不使用。

Windows 98 Windows 98 将忽略 SEC_NOCACHE 标志。

节的第二个保护属性是 SEC_IMAGE，它告诉系统，你映射的文件是个可移植的可执行 (PE) 文件映像。当系统将该文件映射到你的进程的地址空间中时，系统要查看文件的内容，以确定将哪些保护属性赋予文件映像的各个页面。例如，PE 文件的代码节 (.text) 通常用 PAGE_EXECUTE_READ 属性进行映射，而 PE 文件的数据节 (.data) 则通常用 PAGE_READWRITE 属性进行映射。如果设定的属性是 SEC_IMAGE，则告诉系统进行文件映像的映射，并设置相应的页面保护属性。

Windows 98 Windows 98 将忽略 SEC_IMAGE 标志。

最后两个保护属性是 SEC_RESERVE 和 SEC_COMMIT，它们是两个互斥属性，当使用内存映射数据文件时，它们不能使用。这两个标志将在本章后面介绍。当创建内存映射数据文件时，不应该设定这些标志中的任何一个标志。CreateFileMapping 将忽略这些标志。

CreateFileMapping 的另外两个参数是 dwMaximumSizeHigh 和 dwMaximumSizeLow，它们是两个最重要的参数。CreateFileMapping 函数的主要作用是保证文件映射对象能够得到足够的物理存储器。这两个参数将告诉系统该文件的最大字节数。它需要两个 32 位的值，因为 Windows 支持的文件大小可以用 64 位的值来表示。dwMaximumSizeHigh 参数用于设定较高的 32 位，而 dwMaximumSizeLow 参数则用于设定较低的 32 位值。对于 4 GB 或小于 4 GB 的文件来说，dwMaximumSizeHigh 的值将始终是 0。

使用 64 位的值，意味着 Windows 能够处理最大为 16EB (10^{18} 字节) 的文件。如果想要创建一个文件映射对象，使它能够反映文件当前的大小，那么可以为上面两个参数传递 0。如果只打算读取该文件或者访问文件而不改变它的大小，那么为这两个参数传递 0。如果打算将数据附加给该文件，可以选择最大的文件大小，以便为你留出一些富裕的空间。如果当前磁盘上的文件包含 0 字节，那么可以给 CreateFileMapping 函数的 dwMaximumSizeHigh 和 dwMaximumSizeLow 传递两个 0。这样做就可以告诉系统，你要的文件映射对象里面的存储器为 0 字节。这是个错误，CreateFileMapping 将返回 NULL。

如果你对我们讲述的内容一直非常关注，你一定认为这里存在严重的问题。Windows 支持最大为 16EB 的文件和文件映射对象，这当然很好，但是，怎样将这样大的文件映射到 32 位进程的地址空间 (32 位地址空间是 4GB 文件的上限) 中去呢？下一节介绍解决这个问题的办法。当然，64 位进程拥有 16 EB 的地址空间，因此可以进行更大的文件的映射操作，但是，如果文件是个超大规模的文件，仍然会遇到类似的问题。

若要真正理解 CreateFile 和 CreateFileMapping 两个函数是如何运行的，建议你做一个下面的实验。建立下面的代码，对它进行编译，然后在一个调试程序中运行它。当你一步步执行每个语句时，你会跳到一个命令解释程序，并执行 C:\ 目录上的 “dir” 命令。当执行调试程序中的每个语句时，请注意目录中出现的变化。

```
int WINAPI _tWinMain(HINSTANCE hInstExe, HINSTANCE,
    PTSTR pszCmdLine, int nCmdShow) {

    // Before executing the line below, C:\ does not have
```

```

// Before executing the line below, C:\ does not have
// a file called "MMFTest.Dat."
HANDLE hfile = CreateFile("C:\\MMFTest.dat",
    GENERIC_READ | GENERIC_WRITE,
    FILE_SHARE_READ | FILE_SHARE_WRITE, NULL, CREATE_ALWAYS,
    FILE_ATTRIBUTE_NORMAL, NULL);

// Before executing the line below, the MMFTest.Dat
// file does exist but has a file size of 0 bytes.
HANDLE hfilemap = CreateFileMapping(hfile, NULL, PAGE_READWRITE,
    0, 100, NULL);

// After executing the line above, the MMFTest.Dat
// file has a size of 100 bytes.

// Cleanup
CloseHandle(hfilemap);
CloseHandle(hfile);

// When the process terminates, MMFTest.Dat remains
// on the disk with a size of 100 bytes.
return(0);
}

```

如果调用CreateFileMapping函数，传递PAGE_READWRITE标志，那么系统将设法确保磁盘上的相关数据文件的大小至少与dwMaximumSizeHigh和dwMaximumSizeLow参数中设定的大小相同。如果该文件小于设定的大小，CreateFileMapping函数将扩展该文件的大小，使磁盘上的文件变大。这种扩展是必要的，这样，当以后将该文件作为内存映射文件使用时，物理存储器就已经存在了。如果正在用PAGE_READONLY或PAGE_WRITECOPY标志创建该文件映射对象，那么CreateFileMapping特定的文件大小不得大于磁盘文件的物理大小。这是因为你无法将任何数据附加给该文件。

CreateFileMapping函数的最后一个参数是pszName。它是个以0结尾的字符串，用于给该文件映射对象赋予一个名字。该名字用于与其他进程共享文件映射对象（本章后面展示了它的一个例子。第3章详细介绍了内核对象的共享操作）。内存映射数据文件通常并不需要被共享，因此这个参数通常是NULL。

系统创建文件映射对象，并将用于标识该对象的句柄返回该调用线程。如果系统无法创建文件映射对象，便返回一个NULL句柄值。记住，当CreateFile运行失败时，它将返回INVALID_HANDLE_VALUE（定义为-1），当CreateFileMapping运行失败时，它返回NULL。请不要混淆这些错误值。

17.3.3 步骤3：将文件数据映射到进程的地址空间

当创建了一个文件映射对象后，仍然必须让系统为文件的数据保留一个地址空间区域，并将文件的数据作为映射到该区域的物理存储器进行提交。可以通过调用MapViewOfFile函数来进行这项操作：

```

PVOID MapViewOfFile(
    HANDLE hFileMappingObject,
    DWORD dwDesiredAccess,
    DWORD dwFileOffsetHigh,
    DWORD dwFileOffsetLow,
    SIZE_T dwNumberOfBytesToMap);

```

参数

FileMappingObject

用于标识文件映射对象的句柄，该句柄是前面调用CreateFile Mapping或OpenFileMapping（本章后面介绍）函数返回的。参数dwDesiredAccess用于标识如何访问该数据。不错，必须再次设定如何访问文件的数据。可以设定表17-6所列的4个值中的一个。

表17-6 值及其含义

值	含 义
FILE_MAP_WRITE	可以读取和写入文件数据。 CreateFileMapping函数必须通过传递PAGE_READWRITE标志来调用
FILE_MAP_READ	可以读取文件数据。CreateFileMapping函数可以通过传递下列任何一个保护属性来调用：PAGE_READONLY、PAGE_READWRITE或PAGE_WRITECOPY
FILE_MAP_ALL_ACCESS	与FILE_MAP_WRITE相同
FILE_MAP_COPY	可以读取和写入文件数据。如果写入文件数据，可以创建一个页面的私有拷贝。在Windows 2000中，CreateFileMapping函数可以用PAGE_READONLY、PAGE_READWRITE或PAGE_WRITECOPY等保护属性中的任何一个来调用。在Windows 98中，CreateFileMapping必须用PAGE_WRITECOPY来调用

Windows要求所有这些保护属性一次又一次地重复设置，这当然有些奇怪和烦人。我认为这样做可以使应用程序更多地对数据保护属性进行控制。

剩下的3个参数与保留地址空间区域及将物理存储器映射到该区域有关。当你将一个文件映射到你的进程的地址空间中时，你不必一次性地映射整个文件。相反，可以只将文件的一小部分映射到地址空间。被映射到进程的地址空间的这部分文件称为一个视图，这可以说明MapViewOfFile是如何而得名的。

当将一个文件视图映射到进程的地址空间中时，必须规定两件事情。首先，必须告诉系统，数据文件中的哪个字节应该作为视图中的第一个字节来映射。你可以使用 dwFileOffsetHigh和dwFileOffsetLow参数来进行这项操作。由于 Windows支持的文件最大可达16EB，因此必须用一个64位的值来设定这个字节的位移值。这个64位值中，较高的32位传递给参数dwFileOffsetHigh，较低的32位传递给参数dwFileOffsetLow。注意，文件中的这个位移值必须是系统的分配粒度的倍数（迄今为止，Windows的所有实现代码的分配粒度均为64 KB）。第14章介绍了如何获取某个系统的分配粒度。

第二，必须告诉系统，数据文件有多少字节要映射到地址空间。这与设定要保留多大的地址空间区域的情况是相同的。可以使用 dwNumberOfBytesToMap参数来设定这个值。如果设定的值是0，那么系统将设法把从文件中的指定位移开始到整个文件的结尾的视图映射到地址空间。

Windows 98 在Windows 98中，如果MapViewOfFile无法找到足够大的区域来存放整个文件映射对象，那么无论需要的视图是多大，MapViewOfFile均将返回NULL。

Windows 2000 在Windows 2000中，MapViewOfFile只需要为必要的视图找到足够大的一个区域，而不管整个文件映射对象是多大。

如果在调用MapViewOfFile函数时设定了FILE_MAP_COPY标志，系统就会从系统的页文件中提交物理存储器。提交的地址空间数量由 dwNumberOfBytesToMap参数决定。只要你不进行其他操作，只是从文件的映像视图中读取数据，那么系统将决不会使用页文件中的这些提交的页面。但是，如果进程中的任何线程将数据写入文件的映像视图中的任何内存地址，那么系统将从页文件中抓取已提交页面中的一个页面，将原始数据页面拷贝到该页交换文件中，然后

将该拷贝的页面映射到你的进程的地址空间。从这时起，你的进程中的线程就要访问数据的本地拷贝，不能读取或修改原始数据。

当系统制作原始页面的拷贝时，系统将把页面的保护属性从 `PAGE_WRITECOPY` 改为 `PAGE_READWRITE`。下面这个代码段就说明了这个情况：

```
// Open the file that we want to map.
HANDLE hFile = CreateFile(pszFileName, GENERIC_READ | GENERIC_WRITE, 0, NULL,
    OPEN_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);

// Create a file-mapping object for the file.
HANDLE hFileMapping = CreateFileMapping(hFile, NULL, PAGE_WRITECOPY,
    0, 0, NULL);

// Map a copy-on-write view of the file; the system will commit
// enough physical storage from the paging file to accommodate
// the entire file. All pages in the view will initially have
// PAGE_WRITECOPY access.
PBYTE pbFile = (PBYTE) MapViewOfFile(hFileMapping, FILE_MAP_COPY,
    0, 0, 0);

// Read a byte from the mapped view.
BYTE bSomeByte = pbFile[0];
// When reading, the system does not touch the committed pages in
// the paging file. The page keeps its PAGE_WRITECOPY attribute.

// Write a byte to the mapped view.
pbFile[0] = 0;
// When writing for the first time, the system grabs a committed
// page from the paging file, copies the original contents of the
// page at the accessed memory address, and maps the new page
// (the copy) into the process's address space. The new page has
// an attribute of PAGE_READWRITE.

// Write another byte to the mapped view.
pbFile[1] = 0;
// Because this byte is now in a PAGE_READWRITE page, the system
// simply writes the byte to the page (backed by the paging file).

// When finished using the file's mapped view, unmap it.
// UnmapViewOfFile is discussed in the next section.
UnmapViewOfFile(pbFile);
// The system decommits the physical storage from the paging file.
// Any writes to the pages are lost.

// Clean up after ourselves.
CloseHandle(hFileMapping);
CloseHandle(hFile);
```

Windows 98 前面讲过，Windows 98必须预先为内存映射文件提交页文件中的存储器。然而，它只有在必要时才将修改后的页面写入页文件。

17.3.4 步骤4：从进程的地址空间中撤消文件数据的映像

当不再需要保留映射到你的进程地址空间区域中的文件数据时，可以通过调用下面的函数

将它释放：

```
BOOL UnmapViewOfFile(PVOID pvBaseAddress);
```

该函数的唯一的参数 `pvBaseAddress` 用于设定返回区域的基地址。该值必须与调用 `MapViewOfFile` 函数返回的值相同。必须记住要调用 `UnmapViewOfFile` 函数。如果没有调用这个函数，那么在你的进程终止运行前，保留的区域就不会被释放。每当你调用 `MapViewOfFile` 时，系统总是在你的进程地址空间中保留一个新区域，而以前保留的所有区域将不被释放。

为了提高速度，系统将文件的数据页面进行高速缓存，并且在对文件的映射视图进行操作时不立即更新文件的磁盘映像。如果需要确保你的更新被写入磁盘，可以强制系统将修改过的数据的一部分或全部重新写入磁盘映像中，方法是调用 `FlushViewOfFile` 函数：

```
BOOL FlushViewOfFile(  
    PVOID pvAddress,  
    SIZE_T dwNumberOfBytesToFlush);
```

第一个参数是包含在内存映射文件中的视图的一个字节的地址。该函数将你在这里传递的地址圆整为一个页面边界值。第二个参数用于指明你想要刷新的字节数。系统将把这个数字向上圆整，使得字节总数是页面的整数。如果你调用 `FlushViewOfFile` 函数并且不修改任何数据，那么该函数只是返回，而不将任何信息写入磁盘。

对于存储器是在网络上的内存映射文件来说，`FlushViewOfFile` 能够保证文件的数据已经从工作站写入存储器。但是 `FlushViewOfFile` 不能保证正在共享文件的服务器已经将数据写入远程磁盘，因为服务器也许对文件的数据进行了高速缓存。若要保证服务器写入文件的数据，每当你为文件创建一个文件映射对象并且映射该文件映射对象的视图时，应该将 `FILE_FLAG_WRITE_THROUGH` 标志传递给 `CreateFile` 函数。如果你使用该标志打开该文件，那么只有当文件的全部数据已经存放在服务器的磁盘驱动器中的时候，`FlushViewOfFile` 函数才返回。

记住 `UnmapViewOfFile` 函数的一个特殊的特性。如果原先使用 `FILE_MAP_COPY` 标志来映射视图，那么你对文件的数据所作的任何修改，实际上是对存放在系统的页文件中的文件数据的拷贝所作的修改。在这种情况下，如果调用 `UnmapViewOfFile` 函数，该函数在磁盘文件上就没有什么可以更新，而只会释放页文件中的页面，从而导致数据丢失。

如果想保留修改后的数据，必须采用别的措施。例如，你可以用同一个文件创建另一个文件映射对象（使用 `PAGE_READWRITE`），然后使用 `FILE_MAP_WRITE` 标志将这个新文件映射对象映射到进程的地址空间。之后，你可以扫描第一个视图，寻找带有 `PAGE_READWRITE` 保护属性的页面。每当你找到一个带有该属性的页面时，可以查看它的内容，并且确定是否将修改了的数据写入该文件。如果不想用新数据更新该文件，那么继续对视图中的剩余页面进行扫描，直到视图的结尾。但是，如果你确实想要保存修改了的数据页面，那么只需要调用 `MoveMemory` 函数，将数据页面从第一个视图拷贝到第二个视图。由于第二个视图是用 `PAGE_READWRITE` 保护属性映射的，因此 `MoveMemory` 函数将更新磁盘上的实际文件内容。可以使用这种方法来确定文件的变更并保存你的文件的数据。

Windows 98 Windows 98 不支持 `copy-on-write`（写入时拷贝）保护属性，因此，当扫描内存映射文件的第一个视图时，无法测试用 `PAGE_READWRITE` 标志做上标记的页面。你必须设计一种方法来确定第一个视图中的哪些页面已经做了修改。

17.3.5 步骤5和步骤6：关闭文件映射对象和文件对象

不用说，你总是要关闭你打开了的内核对象。如果忘记关闭，在你的进程继续运行时会出现

现资源泄漏的问题。当然，当你的进程终止运行时，系统会自动关闭你的进程已经打开但是忘记关闭的任何对象。但是如果你的进程暂时没有终止运行，你将会积累许多资源句柄。因此你始终都应该编写清楚而又“正确的”代码，以便关闭你已经打开的任何对象。若要关闭文件映射对象和文件对象，只需要两次调用 `CloseHandle` 函数，每个句柄调用一次：

让我们更加仔细地观察一下这个进程。下面的伪代码显示了一个内存映射文件的例子：

```
HANDLE hFile = CreateFile(...);
HANDLE hFileMapping = CreateFileMapping(hFile, ...);
PVOID pvFile = MapViewOfFile(hFileMapping, ...);
```

```
// Use the memory-mapped file.
```

```
UnmapViewOfFile(pvFile);
CloseHandle(hFileMapping);
CloseHandle(hFile);
```

上面的代码显示了对内存映射文件进行操作所用的“预期”方法。但是，它没有显示，当你调用 `MapViewOfFile` 时系统对文件对象和文件映射对象的使用计数的递增情况。这个副作用是很大的，因为它意味着我们可以将上面的代码段重新编写成下面的样子：

```
HANDLE hFile = CreateFile(...);
HANDLE hFileMapping = CreateFileMapping(hFile, ...);
CloseHandle(hFile);
PVOID pvFile = MapViewOfFile(hFileMapping, ...);
CloseHandle(hFileMapping);
```

```
// Use the memory-mapped file.
```

```
UnmapViewOfFile(pvFile);
```

当对内存映射文件进行操作时，通常要打开文件，创建文件映射对象，然后使用文件映射对象将文件的数据视图映射到进程的地址空间。由于系统递增了文件对象和文件映射对象的内部使用计数，因此可以在你的代码开始运行时关闭这些对象，以消除资源泄漏的可能性。

如果用同一个文件来创建更多的文件映射对象，或者映射同一个文件映射对象的多个视图，那么就不能较早地调用 `CloseHandle` 函数——以后你可能还需要使用它们的句柄，以便分别对 `CreateFileMapping` 和 `MapViewOfFile` 函数进行更多的调用。

17.3.6 文件倒序示例应用程序

清单 17-2 列出的 `FileRev` 应用程序（“17-FileRev.exe”）显示了如何使用内存映射对象来对 ANSI 或 Unicode 文本文件的内容进行倒序。该应用程序的源代码和资源文件位于本书所附光盘上的 17-FileRev 目录下。当启动该程序时，会出现图 17-7 所示的窗口。

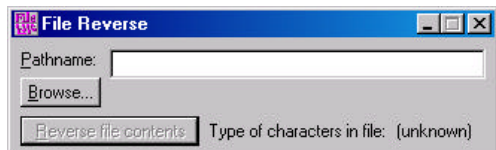


图17-7 运行FileRev 时出现的窗口

`FileRev` 应用程序首先允许选定一个文件，然后，当单击 `Reverse File Contents`（对文件内容进行倒序）按钮时，该函数就会将文件中使用的字符进行倒序。该程序只能对文本文件进行正确的倒序，对二进制文件不能正确地进行倒序操作。`FileRev` 能够确定文本文件是 ANSI 文件还是 Unicode 文件，方法是调用 `IsTextUnicode` 函数（第2章中介绍）。

Windows 98 在 Windows 98 中，`IsTextUnicode` 函数没有可以使用的实现代码，它只是返

回FALSE。如果调用GetLastError函数，则返回ERROR_CALL_NOT_IMPLEMENTED。这意味着FileRev示例应用程序总是认为，当它在Windows 98下运行时，它操作的是ANSI文本文件。

当单击Reverse File Contents按钮时，FileRev便制作指定文件的一个拷贝，称为FileRev.dat。它制作该拷贝的目的是，原始文件不会因为内容被倒序而变得无法使用。接着，FileRev调用FileReverse函数，该函数负责对文件进行倒序操作。FileReverse则调用CreateFile函数，打开FileRev.dat，以便进行读取和写入。

前面说过，对文件内容进行倒序的最容易的方法是调用C运行期函数_strrev。与所有C字符串一样，字符串的最后一个字符必须是个0结束符。由于文本文件不以0为结束符，因此FileRev必须给文件附加一个0。若要进行这样的附加操作，首先要调用GetFileSize函数：

```
dwFileSize = GetFileSize(hFile, NULL);
```

现在已经得到了文件的长度，你能够通过调用CreateFileMapping函数创建文件映射对象。创建的文件映射对象的长度是dwFileSize加一个宽字符的大小（对于0字符来说）。当文件映射对象创建后，该对象的视图就被映射到FileRev的地址空间。变量pvFile包含了MapViewOfFile函数的返回值，并指向文本文件的第一个字节。

下一步是在文件的结尾处写一个0，并对字符串进行倒序：

```
PSTR pchANSI = (PSTR) pvFile;  
pchANSI[dwFileSize / sizeof(CHAR)] = 0;
```

在文本文件中，每一行的结尾都是一个回车符（'\r'）后随一个换行符（'\n'）。但是，当调用_strrev对文件进行倒序时，这些字符也会被倒序。如果将已经倒序的文本文件加载到文本编辑器，那么出现的每一对“\n\r”字符都必须重新改为它的原始顺序。这个倒序操作是由下面的循环代码进行的：

```
while (pchANSI != NULL) {  
    // We have found an occurrence....  
    *pchANSI++ = '\r';    // Change '\n' to '\r'.  
    *pchANSI++ = '\n';    // Change '\r' to '\n'.  
    pchANSI = strchr(pchANSI, '\n'); // Find the next occurrence.  
}
```

当你观察这样一个简单的代码时，可能很容易忘记你实际上是在对磁盘驱动器上的文件内容进行操作（这显示出内存映射文件的功能是多么大）。

在文件被倒序后，FileRev便进行清除操作，撤消文件映射对象的视图映象，关闭所有的内核对象句柄。此外，FileRev必须删除附加给文件结尾处的0字符（记住_strrev并不对结尾的0字符进行倒序）。如果没有删除0字符，那么倒序的文件将会多出一个字符，如果再次调用FileRev函数，将无法使文件还原成它的原始样子。若要删除文件结尾处的0字符，必须后退一步，使用文件管理函数，而不是通过内存映射对文件进行操作。

如果要强制已经倒序的文件在某个位置上结束，就需要将文件指针定位在指定的位置（原始文件的结尾处）并调用SetEndOfFile函数：

```
SetFilePointer(hFile, dwFileSize, NULL, FILE_BEGIN);  
SetEndOfFile(hFile);
```

注意 SetEndOfFile函数必须在撤消视图的映象并且关闭文件映射对象之后调用，否则，该函数将返回FALSE，GetLastError则返回ERROR_USER_MAPPED_FILE。这个错误表示不能在与文件映射对象相关联的文件上执行文件末尾的操作。

FileRev函数做的最后一件事情是产生一个 Notepad实例，这样，就可以查看已经倒序的文件。图17-8显示了在FileRev.cpp文件上运行FileRev函数时产生的结果。

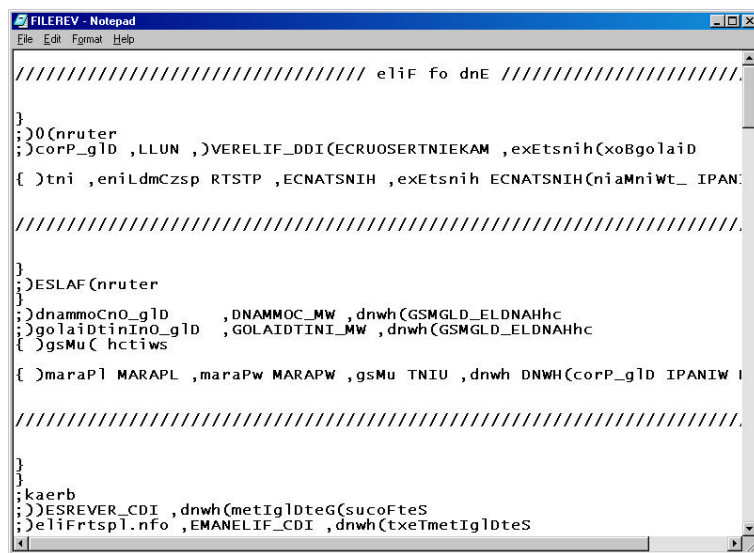


图17-8 FileRev 函数运行结果

清单17-2 FileRev示例应用程序



FileRev.cpp

```

/*****
Module: FileRev.cpp
Notices: Copyright (c) 2000 Jeffrey Richter
*****/

#include "..\CmnHdr.h" /* See Appendix A. */
#include <windowsx.h>
#include <tchar.h>
#include <commdlg.h>
#include <string.h> // For _strrev
#include "Resource.h"

////////////////////////////////////

#define FILENAME TEXT("FILEREV.DAT")

////////////////////////////////////

BOOL FileReverse(PCTSTR pszPathname, PBOOL pfIsTextUnicode) {

    *pfIsTextUnicode = FALSE; // Assume text is Unicode.

```

```

// Open the file for reading and writing.
HANDLE hFile = CreateFile(pszPathname, GENERIC_WRITE | GENERIC_READ, 0,
    NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);

if (hFile == INVALID_HANDLE_VALUE) {
    chMB("File could not be opened.");
    return(FALSE);
}

// Get the size of the file (I assume the whole file can be mapped).
DWORD dwFileSize = GetFileSize(hFile, NULL);

// Create the file-mapping object. The file-mapping object is 1 character
// bigger than the file size so that a zero character can be placed at the
// end of the file to terminate the string (file). Because I don't yet know
// if the file contains ANSI or Unicode characters, I assume worst case
// and add the size of a WCHAR instead of CHAR.
HANDLE hFileMap = CreateFileMapping(hFile, NULL, PAGE_READWRITE,
    0, dwFileSize + sizeof(WCHAR), NULL);

if (hFileMap == NULL) {
    chMB("File map could not be opened.");
    CloseHandle(hFile);
    return(FALSE);
}

// Get the address where the first byte of the file is mapped into memory.
PVOID pvFile = MapViewOfFile(hFileMap, FILE_MAP_WRITE, 0, 0, 0);

if (pvFile == NULL) {
    chMB("Could not map view of file.");
    CloseHandle(hFileMap);
    CloseHandle(hFile);
    return(FALSE);
}

// Does the buffer contain ANSI or Unicode?
int iUnicodeTestFlags = -1; // Try all tests.
*pfIsTextUnicode = IsTextUnicode(pvFile, dwFileSize, &iUnicodeTestFlags);
if (!*pfIsTextUnicode) {
    // For all the file manipulations below, we explicitly use ANSI
    // functions because we are processing an ANSI file.

    // Put a zero character at the very end of the file.
    PSTR pchANSI = (PSTR) pvFile;
    pchANSI[dwFileSize / sizeof(CHAR)] = 0;

    // Reverse the contents of the file.
    _strrev(pchANSI);

    // Convert all "\n\r" combinations back to "\r\n" to
    // preserve the normal end-of-line sequence.
    pchANSI = strchr(pchANSI, '\n'); // Find first '\n'.

    while (pchANSI != NULL) {
        // We have found an occurrence....
    }
}

```

```

        *pchANSI++ = '\r';    // Change '\n' to '\r'.
        *pchANSI++ = '\n';    // Change '\r' to '\n'.
        pchANSI = strchr(pchANSI, '\n'); // Find the next occurrence.
    }

} else {
    // For all the file manipulations below, we explicitly use Unicode
    // functions because we are processing a Unicode file.

    // Put a zero character at the very end of the file.
    PWSTR pchUnicode = (PWSTR) pvFile;
    pchUnicode[dwFileSize / sizeof(WCHAR)] = 0;

    if ((iUnicodeTestFlags & IS_TEXT_UNICODE_SIGNATURE) != 0) {
        // If the first character is the Unicode BOM (byte-order-mark),
        // 0xFEFF, keep this character at the beginning of the file.
        pchUnicode++;
    }

    // Reverse the contents of the file.
    _wcsrev(pchUnicode);

    // Convert all "\n\r" combinations back to "\r\n" to
    // preserve the normal end-of-line sequence.
    pchUnicode = wcschr(pchUnicode, L'\n'); // Find first '\n'.

    while (pchUnicode != NULL) {
        // We have found an occurrence....
        *pchUnicode++ = L'\r';    // Change '\n' to '\r'.
        *pchUnicode++ = L'\n';    // Change '\r' to '\n'.
        pchUnicode = wcschr(pchUnicode, L'\n'); // Find the next occurrence.
    }
}

// Clean up everything before exiting.
UnmapViewOfFile(pvFile);
CloseHandle(hFileMap);

// Remove trailing zero character added earlier.
SetFilePointer(hFile, dwFileSize, NULL, FILE_BEGIN);
SetEndOfFile(hFile);
CloseHandle(hFile);

return(TRUE);
}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

BOOL Dlg_OnInitDialog(HWND hwnd, HWND hwndFocus, LPARAM lParam) {

    chSETDLGICONS(hwnd, IDI_FILEREV);

    // Initialize the dialog box by disabling the Reverse button.
    EnableWindow(GetDlgItem(hwnd, IDC_REVERSE), FALSE);
    return(TRUE);
}

```

//

```
voidDlg_OnCommand(HWND hwnd, int id, HWND hwndCtl, UINT codeNotify) {

    TCHAR szPathname[MAX_PATH];

    switch (id) {
        case IDCANCEL:
            EndDialog(hwnd, id);
            break;

        case IDC_FILENAME:
            EnableWindow(GetDlgItem(hwnd, IDC_REVERSE),
                Edit_GetTextLength(hwndCtl) > 0);
            break;
        case IDC_REVERSE:
            GetDlgItemText(hwnd, IDC_FILENAME, szPathname, chDIMOF(szPathname));

            // Make a copy of input file so that we don't destroy it.
            if (!CopyFile(szPathname, FILENAME, FALSE)) {
                chMB("New file could not be created.");
                break;
            }

            BOOL fIsTextUnicode;
            if (FileReverse(FILENAME, &fIsTextUnicode)) {
                SetDlgItemText(hwnd, IDC_TEXTTYPE,
                    fIsTextUnicode ? TEXT("Unicode") : TEXT("ANSI"));

                // Spawn Notepad to see the fruits of our labors.
                STARTUPINFO si = { sizeof(si) };
                PROCESS_INFORMATION pi;
                TCHAR sz[] = TEXT("Notepad ") FILENAME;
                if (CreateProcess(NULL, sz,
                    NULL, NULL, FALSE, 0, NULL, NULL, &si, &pi)) {

                    CloseHandle(pi.hThread);
                    CloseHandle(pi.hProcess);
                }
            }
            break;

        case IDC_FILESELECT:
            OPENFILENAME ofn = { OPENFILENAME_SIZE_VERSION_400 };
            ofn.hwndOwner = hwnd;
            ofn.lpstrFile = szPathname;
            ofn.lpstrFile[0] = 0;
            ofn.nMaxFile = chDIMOF(szPathname);
            ofn.lpstrTitle = TEXT("Select file for reversing");
            ofn.Flags = OFN_EXPLORER | OFN_FILEMUSTEXIST;
            GetOpenFileName(&ofn);
            SetDlgItemText(hwnd, IDC_FILENAME, ofn.lpstrFile);
            SetFocus(GetDlgItem(hwnd, IDC_REVERSE));
            break;
    }
}
```

```

}

/////////////////////////////////////////////////////////////////
INT_PTR WINAPI Dlg_Proc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {

    switch (uMsg) {
        chHANDLE_DLGMSG(hwnd, WM_INITDIALOG, Dlg_OnInitDialog);
        chHANDLE_DLGMSG(hwnd, WM_COMMAND, Dlg_OnCommand);
    }
    return(FALSE);
}

/////////////////////////////////////////////////////////////////

int WINAPI _tWinMain(HINSTANCE hinstExe, HINSTANCE, PTSTR pszCmdLine, int) {

    DialogBox(hinstExe, MAKEINTRESOURCE(IDD_FILEREV), NULL, Dlg_Proc);
    return(0);
}

///////////////////////////////////////////////////////////////// End of File ///////////////////////////////////////////////////////////////////

```

FileRev.rc

```

//Microsoft Developer Studio generated resource script.
//
#include "Resource.h"

#define APSTUDIO_READONLY_SYMBOLS
/////////////////////////////////////////////////////////////////
//
// Generated from the TEXTINCLUDE 2 resource.
//
#include "afxres.h"

/////////////////////////////////////////////////////////////////
#undef APSTUDIO_READONLY_SYMBOLS

/////////////////////////////////////////////////////////////////
// English (U.S.) resources

#ifdef _WIN32
    #if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_ENU)
        #ifdef _WIN32
            LANGUAGE LANG_ENGLISH, SUBLANG_ENGLISH_US
            #pragma code_page(1252)
        #endif // _WIN32

        ///////////////////////////////////
        //

```



```

#ifdef APSTUDIO_INVOKED
GUIDELINES DESIGNINFO DISCARDABLE
BEGIN
    IDD_FILEREV, DIALOG
    BEGIN
        RIGHTMARGIN, 192
        BOTTOMMARGIN, 42
    END
END
#endif // APSTUDIO_INVOKED

#endif // English (U.S.) resources
////////////////////////////////////

#ifdef APSTUDIO_INVOKED
////////////////////////////////////
//
// Generated from the TEXTINCLUDE 3 resource.
//
////////////////////////////////////
#endif // not APSTUDIO_INVOKED

```

17.4 使用内存映射文件来处理大文件

上一节讲过我要告诉你如何将一个 16 EB 的文件映射到一个较小的地址空间中。当然，你是无法做到这一点的。你必须映射一个只包含一小部分文件数据的文件视图。首先映射一个文件的开头的视图。当完成对文件的第一个视图的访问时，可以取消它的映像，然后映射一个从文件中的一个更深的位移开始的新视图。必须重复这一操作，直到访问了整个文件。这使得大型内存映射文件的处理不太方便，但是，幸好大多数文件都比较小，因此不会出现这个问题。

让我们看一个例子，它使用一个 8 GB 的文件和一个 32 位的地址空间。下面是一个例程，它使用若干个步骤来计算一个二进制数据文件中的所有 0 字节的数目：

```

__int64 Count0s(void) {

    // Views must always start on a multiple
    // of the allocation granularity
    SYSTEM_INFO sinf;
    GetSystemInfo(&sinf);

    // Open the data file.
    HANDLE hFile = CreateFile("C:\\\\HugeFile.Big", GENERIC_READ,
        FILE_SHARE_READ, NULL, OPEN_EXISTING, FILE_FLAG_SEQUENTIAL_SCAN, NULL);

    // Create the file-mapping object.
    HANDLE hFileMapping = CreateFileMapping(hFile, NULL,
        PAGE_READONLY, 0, 0, NULL);

```

```

DWORD dwFileSizeHigh;
__int64 qwFileSize = GetFileSize(hFile, &dwFileSizeHigh);
qwFileSize += (((__int64) dwFileSizeHigh) << 32);

// We no longer need access to the file object's handle.
CloseHandle(hFile);

__int64 qwFileOffset = 0, qwNumOf0s = 0;

while (qwFileSize > 0) {

    // Determine the number of bytes to be mapped in this view
    DWORD dwBytesInBlock = sinf.dwAllocationGranularity;
    if (qwFileSize < sinf.dwAllocationGranularity)
        dwBytesInBlock = (DWORD) qwFileSize;
    PBYTE pbFile = (PBYTE) MapViewOfFile(hFileMapping, FILE_MAP_READ,
        (DWORD) (qwFileOffset >> 32),           // Starting byte
        (DWORD) (qwFileOffset & 0xFFFFFFFF),    // in file
        dwBytesInBlock);                        // # of bytes to map

    // Count the number of Js in this block.
    for (DWORD dwByte = 0; dwByte < dwBytesInBlock; dwByte++) {
        if (pbFile[dwByte] == 0)
            qwNumOf0s++;
    }

    // Unmap the view; we don't want multiple views
    // in our address space.
    UnmapViewOfFile(pbFile);

    // Skip to the next set of bytes in the file.
    qwFileOffset += dwBytesInBlock;
    qwFileSize -= dwBytesInBlock;
}

CloseHandle(hFileMapping);
return(qwNumOf0s);
}

```

这个算法用于映射 64 KB（分配粒度的大小）或更小的视图。另外，要记住，MapViewOfFile函数要求文件的位移是分配粒度大小的倍数。当每个视图被映射到地址空间时，对 0 的扫描不断进行。当每个 64 KB 的文件块已经映射和扫描完毕时，就要通过关闭文件映射对象来对每个文件块进行整理。

17.5 内存映射文件与数据视图的相关性

系统允许你映射一个文件的相同数据的多个视图。例如，你可以将文件开头的 10 KB 映射到一个视图，然后将同一个文件的头 4 KB 映射到另一个视图。只要你是映射相同的文件映射对象，系统就会确保映射的视图数据的相关性。例如，如果你的应用程序改变了一个视图中的文件内容，那么所有其他视图均被更新以反映这个变化。这是因为尽管页面多次被映射到进程的虚拟地址空间，但是系统只将数据放在单个 RAM 页面上。如果多个进程映射单个数据文件的视图，那么数据仍然是相关的，因为在数据文件中，每个 RAM 页面只有一个实例——正是

这个RAM页面被映射到多个进程的地址空间。

注意 Windows允许创建若干个由单个数据文件支持的文件映射对象。Windows不能保证这些不同的文件映射对象的视图具有相关性。它只能保证单个文件映射对象的多个视图具有相关性。

然而，当对文件进行操作时，没有理由使另一个应用程序无法调用 CreateFile函数以打开由另一个进程映射的同一个文件。这个新进程可以使用 ReadFile和WriteFile函数来读取该文件的数据和将数据写入该文件。当然，每当一个进程调用这些函数时，它必须从内存缓冲区读取文件数据或者将文件数据写入内存缓冲区。该内存缓冲区必须是进程自己创建的一个缓冲区，而不是映射文件使用的内存缓冲区。当两个应用程序打开同一个文件时，问题就可能产生：一个进程可以调用 ReadFile函数来读取文件的一个部分，并修改它的数据，然后使用 WriteFile函数将数据重新写入文件，而第二个进程的文件映射对象却不知道第一个进程执行的这些操作。由于这个原因，当你为将被内存映射的文件调用 CreateFile函数时，最好将dwShareMode参数的值设置为0。这样就可以告诉系统，你想要单独访问这个文件，而其他进程都不能打开它。

只读文件不存在相关性问题，因此它们可以作为很好的内存映射文件。内存映射文件决不应该用于共享网络上的可写入文件，因为系统无法保证数据视图的相关性。如果某个人的计算机更新了文件的内容，其他内存中含有原始数据的计算机将不知道它的信息已经被修改。

17.6 设定内存映射文件的基地址

正如你可以使用 VirtualAlloc函数来确定对地址空间进行倒序所用的初始地址一样，你也可以使用MapViewOfFileEx函数而不是使用MapViewOfFile函数来确定一个文件被映射到某个特定的地址。请看下面的代码：

```
PVOID MapViewOfFileEx(  
    HANDLE hFileMappingObject,  
    DWORD dwDesiredAccess,  
    DWORD dwFileOffsetHigh,  
    DWORD dwFileOffsetLow,  
    SIZE_T dwNumberOfBytesToMap,  
    PVOID pvBaseAddress);
```

该函数的所有参数和返回值均与 MapViewOfFile函数相同，唯一的差别是最后一个参数 pvBaseAddress有所不同。在这个参数中，你要映射的文件设定一个目标地址。与 VirtualAlloc一样，你设定的目标地址应该是分配粒度边界（64 KB）的倍数，否则 MapViewOfFileEx将返回NULL，表示出现了错误。

在Windows 2000下，如果设定的地址不是分配粒度的倍数，就会导致函数运行失败，同时 GetLastError将返回1132（ERROR_MAPPED_ALIGNMENT）。在Windows 98中，该地址将圆整为分配粒度边界值。

如果系统无法将文件映射到该位置上（通常由于文件太大并且与另一个保留的地址空间相重叠），那么该函数的运行就会失败并且返回NULL。MapViewOfFileEx并不设法寻找另一个地址空间来放置该文件。当然，你可以设定 NULL作为pvBaseAddress参数的值，这时，MapViewOfFileEx函数的运行特性与MapViewOfFile函数完全相同。

当你使用内存映射文件与其他进程共享数据时，你可以使用 MapViewOfFileEx函数。例如，当两个或多个应用程序需要共享包含指向其他数据结构的一组数据结构时，可能需要在某个特定地址上的内存映射文件。链接表是个极好的例子。在链接表中，每个节点或元素均包含列表

中的另一个元素的内存地址。若要遍历该列表，必须知道第一个元素的地址，然后参考包含下一个元素地址的元素成员。当使用内存映射文件时，这可能成为一个问题。

如果一个进程建立了内存映射文件中的链接表，然后与另一个进程共享该文件，那么另一个进程就可能将文件映射到它的地址空间中的一个完全不同的位置上。当第二个进程视图遍历该链接表时，它查看链接表的第一个元素，检索下一个元素的内存地址，然后设法引用下一个元素。然而，第一个节点中的下一个元素的地址并不是第二个进程需要查找的地址。

可以用两种办法来解决这个问题。首先，当第二个进程将包含链接表的内存映射文件映射到它自己的地址空间中去时，它只要调用 `MapViewOfFileEx` 函数而不是调用 `MapViewOfFile`。当然，这种方法要求第二个进程必须知道第一个进程原先在建立链接表时将文件映射到了什么地方。当两个应用程序打算互相进行交互操作时（这是非常可能的），这就不会出现任何问题，因为地址可以通过硬编码放入两个应用程序，或者一个进程可以通知另一个进程使用另一种进程间通信的方式，比如将消息发送到窗口。

第二个方法是创建链接表的进程将下一个节点所在的地址中的位移存放在每个节点中。这要求应用程序将该位移添加给内存映射文件的基地址，以便访问每个节点。这种方法并不高明，因为它的运行速度可能比较慢，它会使程序变得更大（因为编译器要生成附加代码来执行所有的计算操作），而且它很容易出错。但是，它仍然是个可行的方法，Microsoft 的编译器为使用 `__based` 关键字的基本指针提供了辅助程序。

Windows 98 当调用 `MapViewOfFileEx` 时，必须设定 `0x80000000` 与 `0xBFFFFFFF` 之间的一个地址，否则 `MapViewOfFileEx` 将返回 `ULL`。

Windows 2000 当调用 `MapViewOfFileEx` 时，必须设定在你的进程的用户方式分区中的一个地址，否则 `MapViewOfFileEx` 将返回 `NULL`。

17.7 实现内存映射文件的具体方法

Windows 98 和 Windows 2000 实现内存映射文件的方法是不同的。必须知道这些差别，因为它们会影响你编写代码的方法，也会影响其他应用程序对你的数据进行不利的操作。

在 Windows 98 下，视图总是映射到 `0x80000000` 至 `0xBFFFFFFF` 范围内的地址空间分区中。因此，对 `MapViewOfFile` 函数的成功调用都会返回这个范围内的一个地址。你也许还记得，所有进程都共享该分区中的数据。这意味着如果进程映射了文件映射对象的视图，那么该文件映射对象的数据实际上就可以被所有进程访问，而不管它们是否已经映射了该文件映射对象的视图。如果另一个进程调用使用同一个文件映射对象的 `MapViewOfFile` 函数，Windows 98 便将返回给第一个进程的同一个内存地址返回给第二个进程。这两个进程访问相同的数据，并且它们的视图具有相关性。

在 Windows 98 中，一个进程可以调用 `MapViewOfFile` 函数，并且可以使用某种进程间的通信方式将返回的内存地址传递给另一个进程的线程。一旦该线程收到这个内存地址，该线程就可以成功地访问文件映射对象的同一个视图。但是，不应该这样做，原因有二。

- 你的应用程序将无法在 Windows 2000 下运行，其原因将在下面说明。
- 如果第一个进程调用 `UnmapViewOfFile` 函数，地址空间区域将恢复为空闲状态，这意味着第二个进程的线程如果尝试访问视图曾经位于其中的内存，会引发一次访问违规。

如果第二个进程访问内存映射对象的视图，那么第二个进程中的线程应该调用 `MapViewOfFile` 函数。当第二个进程这样做的时候，系统将对内存映射视图的使用计数进行递增。因此，如果第一个进程调用 `UnmapViewOfFile` 函数，那么在第二个进程也调用 `UnmapViewOfFile` 之前，

系统将不会释放视图占用的地址空间区域。

当第二个进程调用 MapViewOfFile 函数时，返回的地址将与第一个进程返回的地址相同。这样，第一个进程就没有必要使用进程间的通信方式将内存地址传送给第二个进程。

Windows 2000 实现内存映射文件的方法要比 Windows 98 好，因为 Windows 2000 要求在进程的地址空间中的文件数据可供访问之前，该进程必须调用 MapViewOfFile 函数。如果一个进程调用 MapViewOfFile 函数，系统将为调用进程的地址空间中的视图进行地址空间区域的倒序操作，这样，其他进程都将无法看到该视图。如果另一个进程想要访问同一个文件映射对象中的数据，那么第二个进程中的线程就必须调用 MapViewOfFile，同时，系统将为第二个进程的地址空间中的视图进行地址空间区域的倒序操作。

值得注意的是，第一个进程调用 MapViewOfFile 函数后返回的内存地址，很可能不同于第二个进程调用 MapViewOfFile 函数后返回的内存地址。即使这两个进程映射了相同文件映射对象的视图，它们返回的地址也可能不同。在 Windows 98 下，MapViewOfFile 函数返回的内存地址是相同的，但是，如果你想让你的应用程序在 Windows 2000 下运行，那么绝对不应该指望它们也返回相同的地址。

让我们再来观察另一个实现代码的差别。下面是一个小程序，它映射了单个文件映射对象的两个视图：

```
#include <Windows.h>

int WINAPI WinMain (HINSTANCE hinstExe, HINSTANCE,
    PTSTR pszCmdLine, int nCmdShow) {

    // Open an existing file-it must be bigger than 64 KB.
    HANDLE hFile = CreateFile(pszCmdLine, GENERIC_READ | GENERIC_WRITE,
        0, NULL, OPEN_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);

    // Create a file-mapping object backed by the data file.
    HANDLE hFileMapping = CreateFileMapping(hFile, NULL,
        PAGE_READWRITE, 0, 0, NULL);

    // Map a view of the whole file into our address space.
    PBYTE pbFile = (PBYTE) MapViewOfFile(hFileMapping,
        FILE_MAP_WRITE, 0, 0, 0);

    // Map a view of the file (starting 64 KB in) into our address space
    PBYTE pbFile2 = (PBYTE) MapViewOfFile(hFileMapping,
        FILE_MAP_WRITE, 0, 65536, 0);

    if ((pbFile + 65536) == pbFile2) {
        // If the addresses overlap, there is one address
        // space region for both views: this must be Windows 98.
        MessageBox(NULL, "We are running under Windows 98", NULL, MB_OK);
    } else {
        // If the addresses do not overlap, each view has its own
        // address space region: this must be Windows 2000.
        MessageBox(NULL, "We are running under Windows 2000", NULL, MB_OK);
    }

    UnmapViewOfFile(pbFile2);
    UnmapViewOfFile(pbFile);
    CloseHandle(hFileMapping);
    CloseHandle(hFile);

    return(0);
}
```

在Windows 98中,当文件映射对象的视图被映射时,系统将为整个文件映射对象保留足够的地址空间。即使调用MapViewOfFile函数时它的参数指明你想要系统只映射文件映射对象的一小部分,系统也会为它保留足够的地址空间。这意味着即使你规定只映射文件映射对象的一个64 KB的部分,也不能将一个1 GB的文件映射对象映射到一个视图中。

每当进程调用MapViewOfFile时,该函数将返回一个为整个文件映射对象保留的地址空间区域中的地址。因此,在上面的代码段中,第一次调用MapViewOfFile函数时返回包含整个映射文件的区域的基地址,第二次调用MapViewOfFile函数时返回离同一个地址空间区域64 KB位置上的地址。

Windows 2000的实现代码在这里同样存在很大的差别。在上面的代码段中,两次调用MapViewOfFile函数将导致Windows 2000保留两个不同的地址空间区域。第一个区域的大小是文件映射对象的大小,第二个区域的大小是文件映射对象的大小减去64 KB。尽管存在两个不同的区域,但是它们的数据能够保证其相关性,因为两个视图都是从相同的文件映射对象映射而来的。在Windows 98下,各个视图具有相关性,因为它们位于同一个内存中。

17.8 使用内存映射文件在进程之间共享数据

Windows总是出色地提供各种机制,使应用程序能够迅速而方便地共享数据和信息。这些机制包括RPC、COM、OLE、DDE、窗口消息(尤其是WM_COPYDATA)、剪贴板、邮箱、管道和套接字等。在Windows中,在单个计算机上共享数据的最低层机制是内存映射文件。不错,如果互相进行通信的所有进程都在同一台计算机上的话,上面提到的所有机制均使用内存映射文件从事它们的烦琐工作。如果要求达到较高的性能和较小的开销,内存映射文件是举手可得的最佳机制。

数据共享方法是通过让两个或多个进程映射同一个文件映射对象的视图来实现的,这意味着它们将共享物理存储器的同一个页面。因此,当一个进程将数据写入一个共享文件映射对象的视图时,其他进程可以立即看到它们视图中的数据变更情况。注意,如果多个进程共享单个文件映射对象,那么所有进程必须使用相同的名字来表示该文件映射对象。

让我们观察一个例子,启动一个应用程序。当一个应用程序启动时,系统调用CreateFile函数,打开磁盘上的.exe文件。然后系统调用CreateFileMapping函数,创建一个文件映射对象。最后,系统代表新创建的进程调用MapViewOfFileEx函数(它带有SEC_IMAGE标志),这样,.exe文件就可以映射到进程的地址空间。这里调用的是MapViewOfFileEx,而不是MapViewOfFile,这样,文件的映像将被映射到存放在.exe文件映像中的基地址中。系统创建该进程的主线程,将该映射视图的可执行代码的第一个字节的地址放入线程的指令指针,然后CPU启动该代码的运行。

如果用户运行同一个应用程序的第二个实例,系统就认为规定的.exe文件已经存在一个文件映射对象,因此不会创建新的文件对象或者文件映射对象。相反,系统将第二次映射该文件的一个视图,这次是在新创建的进程的地址空间环境中映射的。系统所做的工作是将相同的文件同时映射到两个地址空间。显然,这是对内存的更有效的使用,因为两个进程将共享包含正在执行的这部分代码的物理存储器的同一个页面。

与所有内核对象一样,可以使用3种方法与多个进程共享对象,这3种方法是句柄继承性、句柄命名和句柄复制。关于这3种方法的详细说明,参见第3章的内容。

17.9 页文件支持的内存映射文件

到现在为止,已经介绍了映射驻留在磁盘驱动器上的文件视图的方法。许多应用程序在运

行时都要创建一些数据，并且需要将数据传送给其他进程，或者与其他进程共享。如果应用程序必须在磁盘驱动器上创建数据文件，并且将数据存储在磁盘上以便对它进行共享，那么这将是**非常不方便的**。

Microsoft公司认识到了这一点，并且增加了一些功能，以便创建由系统的页文件支持的内存映射文件，而不是由专用硬盘文件支持的内存映射文件。这个方法与创建内存映射磁盘文件所用的方法几乎相同，不同之处是它更加方便。一方面，它不必调用 CreateFile函数，因为你不是要创建或打开一个指定的文件，你只需要像通常那样调用 CreateFileMapping函数，并且传递INVALID_HANDLE_VALUE作为hFile参数。这将告诉系统，你不是创建其物理存储器驻留在磁盘上的文件中的文件映射对象，相反，你想让系统从它的页文件中提交物理存储器。分配的存储器的数量由 CreateFileMapping函数的dwMaximumSizeHigh和dwMaximumSizeLow两个参数来决定。

当创建了文件映射对象并且将它的一个视图映射到进程的地址空间之后，就可以像使用任何内存区域那样使用它。如果你想要与其他进程共享该数据，可调用 CreateFileMapping函数，并传递一个以0结尾的字符串作为pszName参数。然后，想要访问该存储器的其他进程就可以调用CreateFileMapping或OpenFileMapping函数，并传递相同的名字。

当进程不再想要访问文件映射对象时，该进程应该调用 CloseHandle函数。当所有句柄均被关闭后，系统将从系统的页文件中收回已经提交的存储器。

注意 下面是一个非常有意思的问题，它使单纯的程序员大吃一惊。你能猜到下面这个代码段错在哪里吗？

```
HANDLE hFile = CreateFile(...);
HANDLE hMap = CreateFileMapping(hFile, ...);
if (hMap == NULL)
    return(GetLastError());
...
```

如果上面这个对 CreateFile函数的调用失败，它将返回 INVALID_HANDLE_VALUE。但是，编写这个代码的程序员没有测试一下，看文件是否已经创建成功。当CreateFileMapping函数被调用时，在 hFile参数中传递了 INVALID_HANDLE_VALUE，这使得系统创建一个使用来自页文件而不是来自指定的磁盘文件存储器的文件映像。使用内存映射文件的任何辅助代码都能够正确地运行。但是，当文件映射对象被撤消时，写入文件映射存储器（页文件）的全部数据将被系统撤消。这时，程序员就坐在那里绞尽脑汁，不知道问题究竟出在哪里。因此，必须始终检查 CreateFile函数的返回值，以确定是否出现了错误，因为CreateFile运行失败的原因太多了。

共享内存映射文件的示例应用程序

清单17-3列出的MMFShare应用程序（“17 MMFShare.exe”）显示了如何使用内存映射文件在两个或多个独立的进程间传送数据。该应用程序的源代码和资源文件位于本书所附光盘上的17-MMFShare目录下。

至少需要执行MMFShare程序的两个实例。每个实例创建它自己的对话框，如图17-9所示。

若要将数据从MMFShare的一个实例传送到另一个实例，请将要传送的数据键入Data编辑框。然后单击Create Mapping Of Data（创建数据映像）按钮。当进行这项操作时，MMFShare调用CreateFileMapping函数，创建一个由系统的页文件支持的4 KB内存映射文件对象，并将该对象

命名为MMFSharedData。如果MMFShare发现已经存在一个带有这个名字的文件映射对象，它就显示一个消息框，告诉你它不能创建该对象。如果MMFShare成功地创建了该对象，那么它将进一步将文件的视图映射到进程的地址空间，并将数据从编辑控件拷贝到内存映射文件中。

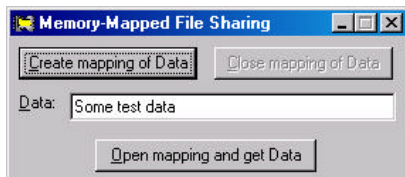


图17-9 运行MMFShare 时出现的对话框

当数据被拷贝后，MMFShare就撤消文件的视图，使Create Mapping Of Data按钮不起作用，并激活Close Mapping Of Data（关闭数据映像）按钮。这时，命名为MMFSharedData的内存映射文件仍然位于系统中的某个位置。没有任何进程映射了包含在文件中的数据视图。

如果这时转入MMFShare的另一个实例，并且单击该实例的 Open Mapping And Get Data（打开映像并获取数据）按钮，那么MMFShare将设法通过调用OpenFileMapping函数，寻找一个称为MMFSharedData的文件映射对象。如果无法找到带有该名字的对象，MMFShare就会显示另一个消息框，将这个情况通知你。如果MMFShare找到了这个对象，它将把该对象的视图映射到它的进程的地址空间，将数据从内存映射文件拷贝到对话框的编辑控件中，然后撤消它的映像，关闭文件映射对象。好极了，你已经成功地将数据从一个进程传送到另一个进程。

对话框中的Close Mapping Of Data（关闭数据映像）按钮用于关闭文件映射对象，它能够释放页文件中的存储器。如果不存在任何文件映射对象，那么MMFShare的其他实例将无法打开文件映像并从中取出数据。另外，如果一个实例已经创建了一个内存映射文件，那么其他实例均不得创建内存映射文件并改写文件中包含的数据。

清单17-3 MMFShare示例应用程序



MMFShare.cpp

```

/*****
Module: MMFShare.cpp
Notices: Copyright (c) 2000 Jeffrey Richter
*****/

#include "..\CmnHdr.h"      /* See Appendix A. */
#include <windowsx.h>
#include <tchar.h>
#include "Resource.h"

////////////////////////////////////

BOOL Dlg_OnInitDialog(HWND hwnd, HWND hwndFocus, LPARAM lParam) {

    chSETDLGICONS(hwnd, IDI_MMFSHARE);

    // Initialize the edit control with some test data.
    Edit_SetText(GetDlgItem(hwnd, IDC_DATA), TEXT("Some test data"));

    // Disable the Close button because the file can't

```

```

// be closed if it was never created or opened.
Button_Enable(GetDlgItem(hwnd, IDC_CLOSEFILE), FALSE);
return(TRUE);
}

////////////////////////////////////

void Dlg_OnCommand(HWND hwnd, int id, HWND hwndCtl, UINT codeNotify) {

    // Handle of the open memory-mapped file
    static HANDLE s_hFileMap = NULL;
    switch (id) {
        case IDCANCEL:
            EndDialog(hwnd, id);
            break;

        case IDC_CREATEFILE:
            if (codeNotify != BN_CLICKED)
                break;

            // Create a paging file-backed MMF to contain the edit control text.
            // The MMF is 4 KB at most and is named MMFSharedData.
            s_hFileMap = CreateFileMapping(INVALID_HANDLE_VALUE, NULL,
                PAGE_READWRITE, 0, 4 * 1024, TEXT("MMFSharedData"));

            if (s_hFileMap != NULL) {

                if (GetLastError() == ERROR_ALREADY_EXISTS) {
                    chMB("Mapping already exists - not created.");
                    CloseHandle(s_hFileMap);
                } else {

                    // File mapping created successfully.

                    // Map a view of the file into the address space.
                    PVOID pView = MapViewOfFile(s_hFileMap,
                        FILE_MAP_READ | FILE_MAP_WRITE, 0, 0, 0);

                    if (pView != NULL) {
                        // Put edit text into the MMF.
                        Edit_GetText(GetDlgItem(hwnd, IDC_DATA),
                            (LPTSTR) pView, 4 * 1024);

                        // Protect the MMF storage by unmapping it.
                        UnmapViewOfFile(pView);

                        // The user can't create another file right now.
                        Button_Enable(hwndCtl, FALSE);

                        // The user closed the file.
                        Button_Enable(GetDlgItem(hwnd, IDC_CLOSEFILE), TRUE);
                    } else {
                        chMB("Can't map view of file.");
                    }
                }
            }
        }
    }
}

```

```

        }
    }
    } else {
        chMB("Can't create file mapping.");
    }
    break;

case IDC_CLOSEFILE:
    if (codeNotify != BN_CLICKED)
        break;

    if (CloseHandle(s_hFileMap)) {
        // User closed the file; fix up the buttons.
        Button_Enable(GetDlgItem(hwnd, IDC_CREATEFILE), TRUE);
        Button_Enable(hwndCtl, FALSE);
    }
    break;

case IDC_OPENFILE:
    if (codeNotify != BN_CLICKED)
        break;

    // See if a memory-mapped file named MMFSharedData already exists.
    HANDLE hFileMapT = OpenFileMapping(FILE_MAP_READ | FILE_MAP_WRITE,
        FALSE, TEXT("MMFSharedData"));

    if (hFileMapT != NULL) {
        // The MMF does exist; map it into the process's address space.
        PVOID pView = MapViewOfFile(hFileMapT,
            FILE_MAP_READ | FILE_MAP_WRITE, 0, 0, 0);

        if (pView != NULL) {

            // Put the contents of the MMF into the edit control.
            Edit_SetText(GetDlgItem(hwnd, IDC_DATA), (LPTSTR) pView);
            UnmapViewOfFile(pView);
        } else {
            chMB("Can't map view.");
        }

        CloseHandle(hFileMapT);
    } else {
        chMB("Can't open mapping.");
    }
    break;
}

}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

INT_PTR WINAPI Dlg_Proc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {

    switch (uMsg) {
        case HANDLE_DLGMSG(hwnd, WM_INITDIALOG, Dlg_OnInitDialog);
        case HANDLE_DLGMSG(hwnd, WM_COMMAND, Dlg_OnCommand);
    }
}

```

```

    }
    return(FALSE);
}

////////////////////////////////////

int WINAPI _tWinMain(HINSTANCE hinstExe, HINSTANCE, PTSTR pszCmdLine, int) {

    DialogBox(hinstExe, MAKEINTRESOURCE(IDD_MMFSHARE), NULL,Dlg_Proc);
    return(0);
}

//////////////////////////////////// End of File //////////////////////////////////////

```

MMFShare.rc

```

//Microsoft Developer Studio generated resource script.
//
#include "Resource.h"

#define APSTUDIO_READONLY_SYMBOLS
////////////////////////////////////
//
// Generated from the TEXTINCLUDE 2 resource.
//
#include "afxres.h"

////////////////////////////////////
#undef APSTUDIO_READONLY_SYMBOLS

////////////////////////////////////
// English (U.S.) resources
#if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_ENU)
#ifdef _WIN32
LANGUAGE LANG_ENGLISH, SUBLANG_ENGLISH_US
#pragma code_page(1252)
#endif // _WIN32

////////////////////////////////////
//
// Dialog
//

IDD_MMFSHARE DIALOG DISCARDABLE 38, 36, 186, 61
STYLE WS_MINIMIZEBOX | WS_POPUP | WS_VISIBLE | WS_CAPTION | WS_SYSMENU
CAPTION "Memory-Mapped File Sharing"
FONT 8, "MS Sans Serif"
BEGIN
    PUSHBUTTON        "&Create mapping of Data", IDC_CREATEFILE, 4, 4, 84, 14,
                     WS_GROUP
    PUSHBUTTON        "&Close mapping of Data", IDC_CLOSEFILE, 96, 4, 84, 14

```

END

```

////////////////////////////////////
//
// Icon
//

```

```
// Icon with lowest ID value placed first to ensure application icon
// remains consistent on all systems.
IDI_MMFSHARE          ICON        DISCARDABLE        "MMFShare.Ico"
```

```
#ifndef APSTUDIO_INVOKED
////////////////////////////////////
//
// TEXTINCLUDE
//
```

```
1 TEXTINCLUDE DISCARDABLE
BEGIN
    "Resource.h\0"
END
```

```
2 TEXTINCLUDE DISCARDABLE
BEGIN
    "#include "afxres.h""\r\n"
    "\0"
END
```

```
3 TEXTINCLUDE DISCARDABLE
BEGIN
    "\r\n"
    "\0"
END
```

```
#endif // APSTUDIO_INVOKED
```

```
#endif // English (U.S.) resources
```

```
#ifndef APSTUDIO_INVOKED
////////////////////////////////////
//
// Generated from the TEXTINCLUDE 3 resource.
//
```

[illegible]

17.10 稀疏提交的内存映射文件

在迄今为止介绍的所有内存映射文件中，我们发现系统要求为内存映射文件提交的所有存储器必须是在磁盘上的数据文件中或者是在页文件中。这意味着我们不能根据我们的喜好来有效地使用存储器。让我们回到第15章中介绍电子表格的内容上来，比如说，你想要与另一个进程共享整个电子表格。如果我们使用内存映射文件，那么必须为整个电子表格提交物理存储器：

```
CELLDATA CellData[200][256];
```

如果CELLDATA结构的大小是128字节，那么这个数组需要6 553 600 (200 x 256 x 128)字节的物理存储器。第15章讲过，如果用页文件为电子表格分配物理存储器，那么这是个不小的数目了，尤其是考虑到大多数用户只是将信息放入少数的单元格中，而大多数单元格却空闲不用时，这就显得有些浪费。

显然，我们宁愿将电子表格作为一个文件映射对象来共享，而不必预先提交所有的物理存储器。CreateFileMapping函数为这种操作提供了一种方法，即可以在fdwProtect参数中设定SEC_RESERVE或SEC_COMMIT标志。

只有当创建由系统的页文件支持的文件映射对象时，这些标志才有意义。SEC_COMMIT标志能使CreateFileMapping从系统的页文件中提交存储器。如果两个标志都不设定，其结果也一样。

当调用CreateFileMapping函数并传递SEC_RESERVE标志时，系统并不从它的页文件中提交物理存储器，它只是返回文件映射对象的一个句柄。这时可以调用MapViewOfFile或MapViewOfFileEx函数，创建该文件映射对象的视图。MapViewOfFile和MapViewOfFileEx将保留一个地址空间区域，并且不提交支持该区域的任何物理存储器。对保留区域中的内存地址进行访问的任何尝试均将导致线程引发访问违规。

现在我们得到的是一个保留的地址空间区域和用于标识该区域的文件映射对象的句柄。其他进程可以使用相同的文件映射对象来映射同一个地址空间区域的视图。物理存储器仍然没有被提交给该区域。如果其他进程中的线程试图访问它们区域中的视图的内存地址，这些线程将会引发访问违规。

下面是令人感兴趣的一些事情。若要将物理存储器提交给共享区域，线程需要做的操作只是调用VirtualAlloc函数：

```
PVOID VirtualAlloc(  
    PVOID pvAddress,  
    SIZE_T dwSize,  
    DWORD fdwAllocationType,  
    DWORD fdwProtect);
```

第15章已经介绍了这个函数。调用VirtualAlloc函数将物理存储器提交给内存映射视图区域，就像是调用VirtualAlloc函数将存储器提交给开始时通过调用带有MEM_RESERVE标志的VirtualAlloc函数而保留的区域一样。而且，就像你可以提交稀疏地存在于用VirtualAlloc保留的区域中的存储器一样，你也可以提交稀疏地存在于用MapViewOfFile或MapViewOfFileEx保留的区域中的存储器。但是，当你将存储器提交给用MapViewOfFile或MapViewOfFileEx保留的区域时，已经映射了相同文件映射对象视图的所有进程这时就能够成功地访问已经提交的页面。

使用SEC_RESERVE标志和VirtualAlloc函数，就能够成功地与其他进程共享电子表格应用

程序的CellData数组，并且能够非常有效地使用物理存储器。

Windows 98 通常情况下，当给 VirtualAlloc函数传递的内存地址位于 0x00400000至 0x7FFFFFFF以外时，VirtualAlloc的运行就会失败。但是，当将物理存储器提交给使用 SEC_RESERVE标志创建的内存映射文件时，必须调用 VirtualAlloc函数，传递一个位于 0x80000000至 0xBFFFFFFF之间的内存地址。Windows 98知道你正在把存储器提交给一个保留的内存映射文件，并且让这个函数调用取得成功。

注意 在Windows 2000下，无法使用 VirtualFree函数从使用 SEC_RESERVE标志保留的内存映射文件中释放存储器。但是，Windows 98允许在这种情况下调用 VirtualFree函数来释放存储器。

NT文件系统（NTFS 5）提供了对稀疏文件的支持。这是个非常出色的新特性。使用这个新的稀疏文件特性，能够很容易地创建和使用稀疏内存映射文件，在这些稀疏内存映射文件中，存储器包含在通常的磁盘文件中，而不是在系统的页文件中。

下面是如何使用稀疏文件特性的一个例子。比如，你想要创建一个 MMF文件，以便存放记录的音频数据。当用户说话时，你想要将数字音频数据写入内存缓冲区，并且让该缓冲区得到磁盘上的一个文件的支持。稀疏 MMF当然是在你的代码中实现这个要求的最容易和最有效的方法。问题是你不知道用户在单击 Stop（停止）按钮之前讲了多长时间。你可能需要一个足够大的文件来存放 5分钟或 5小时的数据，这两个时间长度的差别太大了。但是，当使用稀疏 MMF时，数据文件的大小确实无关紧要。

稀疏内存映射文件示例应用程序

清单 17-4列出的 MMFSparse应用程序（“17 MMFSparse.exe”）显示了如何创建一个由 NTFS 5支持的内存映射文件。该应用程序的源代码和资源文件位于本书所附光盘上的 17-MMFSparse目录下。当启动该程序时，便会出现图 17-10所示的窗口。

当单击 Create a 1MB（1024 KB）Sparse MMF（创建一个 1MB（1024 KB）稀疏 MMF）按钮时，该程序将设法创建一个称为“C:\MMFSparse”的稀疏文件。如果你的 C驱动器不是个 NTFS 5卷，那么它的运行将会失败，并且该进程将终止运行。如果你的 NTFS 5卷在另一个驱动器名上，你必须修改源代码，并且将它重建，以了解应用程序运行的情况。

一旦稀疏文件创建完成，它就被映射到进程的地址空间中。底部的 Allocated Ranges（分配的范围）编辑框显示了文件的哪些部分实际上是由磁盘存储器支持的。开始时，该文件中没有任何存储器在里面，而编辑控件中则包含了“No allocated ranges in the file”（文件中没有分配范围）这一消息文本。

若要读取一个字节，只需将一个数字输入 Offset（位移）编辑框中，并单击 Read Byte（读取字节）按钮。输入的数字与 1024（1 KB）相乘，在该位置上的字节被读取并放入 Byte编辑框中。如果从没有支持存储器的任何部分中读取字节，将始终只能读取一个 0字节。如果从拥有支持存储器的文件部分读取字节，将能够读取那里的任何字节。

若要写入一个字节，请将一个数字输入 Offset编辑框，并且将一个字节值（0至255）输入

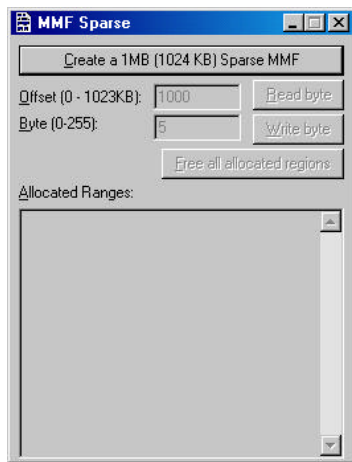


图17-10 MMF Sparse窗口

Byte编辑框。然后,当单击 Write Byte (写入字节)按钮时,位移数字就与 1024相乘,同时,该位置上的字节被修改以反映指定的字节值。这个写入操作可使系统为该部分文件提交支持的存储器。当读取或写入操作执行完成后,Allocated Ranges编辑框总是会得到更新,以便向你显示文件的哪些部分实际上得到存储器的支持。图 17-11显示了在1 024 000 (1000 x 1024) 这个位移上仅仅写入一个字节后对话框是个什么样子。

注意图 17-11中只存在一个分配范围,它从文件中的逻辑位移 983 040字节开始,并且支持存储器的65 536个字节已经被分配。也可以使用 Explorer来找出文件 C:\MMFSparse 并显示它的属性页,如图 17-12所示。

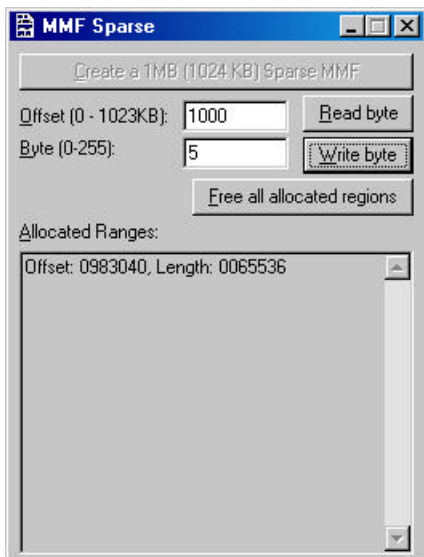


图17-11 写入一个字节后的 MMF Sparse 对话框

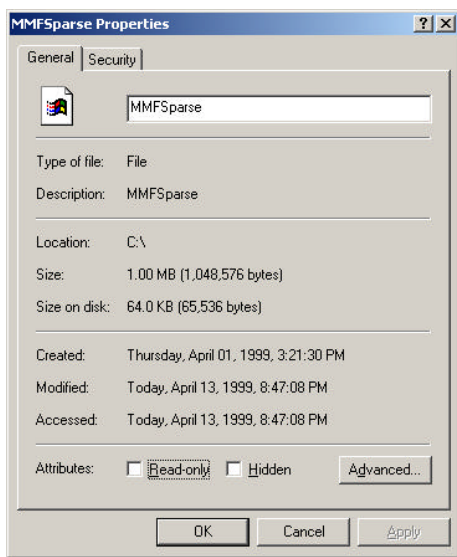


图17-12 MMF Sparse Properties 对话框

注意,该属性页显示了文件的大小是 1 MB (这是文件的虚拟大小),但是该文件实际上只占用 64 KB 磁盘空间。

最后一个按钮是 Free All Allocated Regions (释放所有分配的区域),该程序可以用它来释放用于文件的所有存储器。这个特性能够释放磁盘空间,使文件中的所有字节均显示为 0。

下面让我们来介绍一下该程序是如何运行的。为了简便起见,我创建了一个 CSparseStream 的 C++ 类 (在 SparseStream.h 文件中实现)。这个类封装了可以用稀疏文件或数据流执行的任务。然后,在 MMFSparse.cpp 文件中,我创建了另一个 C++ 类 CMMFSparse,它是由 CSparseStream 派生而来的。因此,CMMFSparse 对象将拥有 CSparseStream 的所有特性,并且要加上将稀疏数据流用作内存映射文件时特定的几个特性。该进程拥有 CMMFSparse 对象的单个全局实例,称为 g_mmf。应用程序在它的整个代码中都要引用这个全局变量,以便对稀疏内存映射文件进行操作。

当用户单击 Create a 1 MB (1024 KB) Sparse MMF (创建一个 1 MB (1024 KB) 的稀疏 MMF) 按钮时,CreateFile 函数被调用,以便在 NTFS 5 磁盘分区上创建一个新文件。这是个普通的常规文件。但是,这时我使用 g_mmf 对象并且调用它的 Initialize 方法,传递该文件的句柄和文件的最大长度 (1 MB)。在系统内部,Initialize 方法调用 CreateFileMapping 函数,按照指定的大小创建文件映射内核对象,然后调用 MapViewOfFile 函数,使得该稀疏文件出现在进程的地址空间中。

当Initialize方法返回时，Dlg_ShowAllocatedRanges函数被调用。该函数在内部调用各个Windows函数，以便枚举已经为之分配了存储器的稀疏文件的逻辑范围。每个分配范围的起始位移和长度显示在对话框底部的编辑控件中。当g_mmf对象首次被初始化时，已经为磁盘上的文件分配的物理存储器实际上是0，编辑控件将反映出这个情况。

这时，用户可以设法从稀疏内存映射文件中读取数据，或者将数据写入该文件。如果试图写入数据，用户的位移和字节值可以从它们各自的编辑控件中获得，并将数据写入g_mmf对象中的内存地址。如果将数据写入g_mmf，文件系统就会将存储器分配给文件的这个逻辑分区，不过分配情况对应用程序来说是透明的。

如果用户试图从g_mmf对象中读取一个字节，那么该读取操作将设法在已经分配了存储器的文件中读取一个字节，否则该字节也许会标识一个尚未分配存储器的字节。如果该字节尚未被分配存储器，那么读取该字节就会返回0。同样，这对于应用程序来说是透明的。如果存在供被读取的字节使用的存储器，当然就返回它的实际值。

该应用程序说明的最后一个问题是如何清除文件，使它的所有已分配的存储器范围被释放，文件不再需要磁盘存储器。若要释放所有的已分配范围，用户可以单击Free All Allocated Ranges按钮。Windows无法为内存映射文件释放所有已分配的范围，因此应用程序要做的第一件事情是调用g_mmf对象的ForceClose方法。ForceClose方法在内部调用UnmapViewOfFile函数，然后调用CloseHandle，传递文件映射内核对象的句柄。

接着DecommitPortionOfStream函数被调用，为文件中的0至1 MB的逻辑字节释放所有的存储器。最后，再次为g_mmf对象调用Initialize方法，将内存映射文件重新初始化到进程的地址空间中。为了证明文件已经释放它的全部分配范围，需调用Dlg_ShowAllocatedRanges函数，它在编辑控件中显示“No allocated ranges in the file”(文件中没有已分配范围)字符串。

最后需要说明的是，如果在实际的应用程序中使用稀疏内存映射文件，当关闭文件时，可能必须截断文件实际的逻辑长度。虽然将包含0字节的稀疏文件的结尾截去不会对磁盘空间产生任何影响，这确实是很好的事情——Explorer和命令外壳的DIR命令能够向用户报告比较准确的文件大小。若要为文件设置文件标记的结尾，可以在调用ForceClose方法后，调用SetFilePointer和SetEndOfFile函数。

清单17-4 MMFSparse示例应用程序



MMFSparse.cpp

```

/*****
Module: MMFSparse.cpp
Notices: Copyright (c) 2000 Jeffrey Richter
*****/

#include "..\CmnHdr.h"      /* See Appendix A. */
#include <tchar.h>
#include <WindowsX.h>
#include <WinIoctl.h>
#include "SparseStream.h"
#include "Resource.h"

```

```

/////////////////////////////////////////////////////////////////

// This class makes it easy to work with memory-mapped sparse files.
class CMMFSparse : public CSparseStream {
private:
    HANDLE m_hfilemap;        // File-mapping object
    PVOID m_pvFile;          // Address to start of mapped file

public:
    // Creates a Sparse MMF and maps it in the process's address space.
    CMMFSparse(HANDLE hstream = NULL, SIZE_T dwStreamSizeMax = 0);

    // Closes a Sparse MMF.
    virtual ~CMMFSparse() { ForceClose(); }

    // Creates a sparse MMF and maps it in the process's address space.
    BOOL Initialize(HANDLE hstream, SIZE_T dwStreamSizeMax);

    // MMF to BYTE cast operator returns address of first byte
    // in the memory-mapped sparse file.
    operator PBYTE() const { return((PBYTE) m_pvFile); }

    // Allows you to explicitly close the MMF without having
    // to wait for the destructor to be called.
    VOID ForceClose();
};

/////////////////////////////////////////////////////////////////

CMMFSparse::CMMFSparse(HANDLE hstream, SIZE_T dwStreamSizeMax) {

    Initialize(hstream, dwStreamSizeMax);
}

/////////////////////////////////////////////////////////////////

BOOL CMMFSparse::Initialize(HANDLE hstream, SIZE_T dwStreamSizeMax) {

    if (m_hfilemap != NULL)
        ForceClose();

    // Initialize to NULL in case something goes wrong.
    m_hfilemap = m_pvFile = NULL;
    BOOL fOk = TRUE; // Assume success.

    if (hstream != NULL) {
        if (dwStreamSizeMax == 0) {
            DebugBreak(); // Illegal stream size
        }

        CSparseStream::Initialize(hstream);
        fOk = MakeSparse(); // Make the stream sparse.
        if (fOk) {

```

```

        // Create a file-mapping object
        m_hfilemap = ::CreateFileMapping(hstream, NULL, PAGE_READWRITE,
            (DWORD) (dwStreamSizeMax >> 32i64), (DWORD) dwStreamSizeMax, NULL);

        if (m_hfilemap != NULL) {
            // Map the stream into the process's address space.
            m_pvFile = ::MapViewOfFile(m_hfilemap,
                FILE_MAP_WRITE | FILE_MAP_READ, 0, 0, 0);
        } else {
            // Failed to map the file; clean up
            CSparseStream::Initialize(NULL);
            ForceClose();
            fOk = FALSE;
        }
    }
}
return(fOk);
}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

VOID CMMFSparse::ForceClose() {

    // Clean up everything that was done successfully
    if (m_pvFile != NULL) {
        ::UnmapViewOfFile(m_pvFile);
        m_pvFile = NULL;
    }
    if (m_hfilemap != NULL) {
        ::CloseHandle(m_hfilemap);
        m_hfilemap = NULL;
    }
}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

#define STREAMSIZE      (1 * 1024 * 1024)    // 1 MB (1024 KB)
TCHAR szPathname[] = TEXT("C:\\MMFSparse.");
HANDLE g_hstream = INVALID_HANDLE_VALUE;
CMMFSparse g_mmf;

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

BOOL Dlg_OnInitDialog(HWND hwnd, HWND hwndFocus, LPARAM lParam) {

    chSETDLGICONS(hwnd, IDI_MMFSPARSE);

    // Initialize the dialog box controls.
    EnableWindow(GetDlgItem(hwnd, IDC_OFFSET), FALSE);
    Edit_LimitText(GetDlgItem(hwnd, IDC_OFFSET), 4);
    SetDlgItemInt(hwnd, IDC_OFFSET, 1000, FALSE);

    EnableWindow(GetDlgItem(hwnd, IDC_BYTE), FALSE);

```

```

Edit_LimitText(GetDlgItem(hwnd, IDC_BYTE), 3);
SetDlgItemInt(hwnd, IDC_BYTE, 5, FALSE);

EnableWindow(GetDlgItem(hwnd, IDC_WRITEBYTE), FALSE);
EnableWindow(GetDlgItem(hwnd, IDC_READBYTE), FALSE);
EnableWindow(GetDlgItem(hwnd, IDC_FREEALLOCATEDREGIONS), FALSE);

return(TRUE);
}

////////////////////////////////////

voidDlg_ShowAllocatedRanges(HWND hwnd) {

    // Fill in the Allocated Ranges edit control
    DWORD dwNumEntries;
    FILE_ALLOCATED_RANGE_BUFFER* pfarb =
        g_mmf.QueryAllocatedRanges(&dwNumEntries);

    if (dwNumEntries == 0) {
        SetDlgItemText(hwnd, IDC_FILESTATUS,
            TEXT("No allocated ranges in the file"));
    } else {
        TCHAR sz[4096] = { 0 };
        for (DWORD dwEntry = 0; dwEntry < dwNumEntries; dwEntry++) {
            wsprintf(_tcschr(sz, 0), TEXT("Offset: %7.7u, Length: %7.7u\r\n"),
                pfarb[dwEntry].FileOffset.LowPart, pfarb[dwEntry].Length.LowPart);
        }
        SetDlgItemText(hwnd, IDC_FILESTATUS, sz);
    }
    g_mmf.FreeAllocatedRanges(pfarb);
}

////////////////////////////////////

voidDlg_OnCommand(HWND hwnd, int id, HWND hwndCtl, UINT codeNotify) {

    switch (id) {
        case IDCANCEL:
            if (g_hstream != INVALID_HANDLE_VALUE)
                CloseHandle(g_hstream);
            EndDialog(hwnd, id);
            break;

        case IDC_CREATEMMF:
            // Create the file
            g_hstream = CreateFile(szPathname, GENERIC_READ | GENERIC_WRITE,
                0, NULL, CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
            if (g_hstream == INVALID_HANDLE_VALUE) {
                chFAIL("Failed to create file.");
            }

            // Create a 1 MB (1024 KB) MMF using the file

```

```

        if (!g_mmf.Initialize(g_hstream, STREAMSIZE)) {
            chFAIL("Failed to initialize Sparse MMF.");
        }
        Dlg_ShowAllocatedRanges(hwnd);

        // Enable/disable the other controls.
        EnableWindow(GetDlgItem(hwnd, IDC_CREATEMMF), FALSE);
        EnableWindow(GetDlgItem(hwnd, IDC_OFFSET), TRUE);
        EnableWindow(GetDlgItem(hwnd, IDC_BYTE), TRUE);
        EnableWindow(GetDlgItem(hwnd, IDC_WRITEBYTE), TRUE);
        EnableWindow(GetDlgItem(hwnd, IDC_READBYTE), TRUE);
        EnableWindow(GetDlgItem(hwnd, IDC_FREEALLOCATEDREGIONS), TRUE);
        // Force the Offset edit control to have the focus.
        SetFocus(GetDlgItem(hwnd, IDC_OFFSET));
        break;

case IDC_WRITEBYTE:
    {
        BOOL fTranslated;
        DWORD dwOffset = GetDlgItemInt(hwnd, IDC_OFFSET, &fTranslated, FALSE);
        if (fTranslated) {
            g_mmf[dwOffset * 1024] = (BYTE)
                GetDlgItemInt(hwnd, IDC_BYTE, NULL, FALSE);
            Dlg_ShowAllocatedRanges(hwnd);
        }
    }
    break;

case IDC_READBYTE:
    {
        BOOL fTranslated;
        DWORD dwOffset = GetDlgItemInt(hwnd, IDC_OFFSET, &fTranslated, FALSE);
        if (fTranslated) {
            SetDlgItemInt(hwnd, IDC_BYTE, g_mmf[dwOffset * 1024], FALSE);
            Dlg_ShowAllocatedRanges(hwnd);
        }
    }
    break;

case IDC_FREEALLOCATEDREGIONS:
    // Normally the destructor causes the file-mapping to close.
    // But, in this case, we wish to force it so that we can reset
    // a portion of the file back to all zeroes.
    g_mmf.ForceClose();

    // We call ForceClose above because attempting to zero a portion of
    // the file while it is mapped causes DeviceIoControl to fail with
    // error ERROR_USER_MAPPED_FILE ("The requested operation cannot
    // be performed on a file with a user-mapped section open.")
    g_mmf.DecommitPortionOfStream(0, STREAMSIZE);
    g_mmf.Initialize(g_hstream, STREAMSIZE);
    Dlg_ShowAllocatedRanges(hwnd);
    break;
}
}

```

```
////////////////////////////////////
```

```
INT_PTR WINAPI Dlg_Proc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {

    switch (uMsg) {
        case WM_INITDIALOG: Dlg_OnInitDialog();
        case WM_COMMAND:    Dlg_OnCommand();
    }
    return(FALSE);
}
```

```
////////////////////////////////////
```

```
int WINAPI _tWinMain(HINSTANCE hinstExe, HINSTANCE, PTSTR pszCmdLine, int) {

    chWindows2000Required();

    DialogBox(hinstExe, MAKEINTRESOURCE(IDD_MMFSPARSE), NULL, Dlg_Proc);
    return(0);
}
```

```
//////////////////////////////////// End of File //////////////////////////////////////
```

MMFSpase.rc

```
//Microsoft Developer Studio generated resource script.
```

```
//
```

```
#include "Resource.h"
```

```
#define APSTUDIO_READONLY_SYMBOLS
```

```
////////////////////////////////////
```

```
//
```

```
// Generated from the TEXTINCLUDE 2 resource.
```

```
//
```

```
#include "afxres.h"
```

```
////////////////////////////////////
```

```
#undef APSTUDIO_READONLY_SYMBOLS
```

```
////////////////////////////////////
```

```
// English (U.S.) resources
```

```
#if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_ENU)
```

```
#ifdef _WIN32
```

```
LANGUAGE LANG_ENGLISH, SUBLANG_ENGLISH_US
```

```
#pragma code_page(1252)
```

```
#endif // _WIN32
```

```
////////////////////////////////////
```

```
//
```

//

END

////////////////////////////////////

11

11

////////////////////////////////////

11

11

////////////////////////////////////

```
#ifndef APSTUDIO_INVOKED
```

```

////////////////////////////////////
//
// Generated from the TEXTINCLUDE 3 resource.
//

```

```

////////////////////////////////////
#endif // not APSTUDIO_INVOKED

```

SparseStream.h

```

/*****
Module: SparseStream.h
Notices: Copyright (c) 2000 Jeffrey Richter
*****/
#include "..\CmnHdr.h" /* See Appendix A. */
#include <WinIoCtl.h>

```

```

////////////////////////////////////

```

```
#pragma once
```

```

////////////////////////////////////

```

```

class CSparseStream {
public:
    static BOOL DoesFileSystemSupportSparseStreams(PCTSTR pszVolume);
    static BOOL DoesFileContainAnySparseStreams(PCTSTR pszPathname);

public:
    CSparseStream(HANDLE hstream = INVALID_HANDLE_VALUE) {
        Initialize(hstream);
    }

    virtual ~CSparseStream() { }

    void Initialize(HANDLE hstream = INVALID_HANDLE_VALUE) {
        m_hstream = hstream;
    }

public:
    operator HANDLE() const { return(m_hstream); }

public:
    BOOL IsStreamSparse() const;
    BOOL MakeSparse();
    BOOL DecommitPortionOfStream(
        __int64 qwFileOffsetStart, __int64 qwFileOffsetEnd);

    FILE_ALLOCATED_RANGE_BUFFER* QueryAllocatedRanges(PDWORD pdwNumEntries);

```

```

    BOOL FreeAllocatedRanges(FILE_ALLOCATED_RANGE_BUFFER* pfarb);

private:
    HANDLE m_hstream;
private:
    static BOOL AreFlagsSet(DWORD fdwFlagBits, DWORD fFlagsToCheck) {
        return((fdwFlagBits & fFlagsToCheck) == fFlagsToCheck);
    }
};

/////////////////////////////////////////////////////////////////

inline BOOL CSparseStream::DoesFileSystemSupportSparseStreams(
    PCTSTR pszVolume) {

    DWORD dwFileSystemFlags = 0;
    BOOL fOk = GetVolumeInformation(pszVolume, NULL, 0, NULL, NULL,
        &dwFileSystemFlags, NULL, 0);
    fOk = fOk && AreFlagsSet(dwFileSystemFlags, FILE_SUPPORTS_SPARSE_FILES);
    return(fOk);
}

/////////////////////////////////////////////////////////////////

inline BOOL CSparseStream::IsStreamSparse() const {

    BY_HANDLE_FILE_INFORMATION bhfi;
    GetFileInformationByHandle(m_hstream, &bhfi);
    return(AreFlagsSet(bhfi.dwFileAttributes, FILE_ATTRIBUTE_SPARSE_FILE));
}

/////////////////////////////////////////////////////////////////

inline BOOL CSparseStream::MakeSparse() {

    DWORD dw;
    return(DeviceIoControl(m_hstream, FSCTL_SET_SPARSE,
        NULL, 0, NULL, 0, &dw, NULL));
}

/////////////////////////////////////////////////////////////////

inline BOOL CSparseStream::DecommitPortionOfStream(
    __int64 qwOffsetStart, __int64 qwOffsetEnd) {

    // NOTE: This function does not work if this file is memory-mapped.
    DWORD dw;
    FILE_ZERO_DATA_INFORMATION fzdi;
    fzdi.FileOffset.QuadPart = qwOffsetStart;
    fzdi.BeyondFinalZero.QuadPart = qwOffsetEnd + 1;
    return(DeviceIoControl(m_hstream, FSCTL_SET_ZERO_DATA, (LPVOID) &fzdi,
        sizeof(fzdi), NULL, 0, &dw, NULL));
}

/////////////////////////////////////////////////////////////////

inline BOOL CSparseStream::DoesFileContainAnySparseStreams(

```

```
PCTSTR pszPathname) {

    DWORD dw = GetFileAttributes(pszPathname);
    return((dw == 0xffffffff)
        ? FALSE : AreFlagsSet(dw, FILE_ATTRIBUTE_SPARSE_FILE));
}

//////////////////////////////////////////////////////////////////////////////////////////////////////////////////

inline FILE_ALLOCATED_RANGE_BUFFER* CSparseStream::QueryAllocatedRanges(
    PDWORD pdwNumEntries) {

    FILE_ALLOCATED_RANGE_BUFFER farb;
    farb.FileOffset.QuadPart = 0;
    farb.Length.LowPart =
        GetFileSize(m_hstream, (PDWORD) &farb.Length.HighPart);

    // There is no way to determine the correct memory block size prior to
    // attempting to collect this data, so I just picked 100 * sizeof(*pfarb)
    DWORD cb = 100 * sizeof(farb);
    FILE_ALLOCATED_RANGE_BUFFER* pfarb = (FILE_ALLOCATED_RANGE_BUFFER*)
        HeapAlloc(GetProcessHeap(), HEAP_ZERO_MEMORY, cb);

    DeviceIoControl(m_hstream, FSCTL_QUERY_ALLOCATED_RANGES,
        &farb, sizeof(farb), pfarb, cb, &cb, NULL);
    *pdwNumEntries = cb / sizeof(*pfarb);
    return(pfarb);
}

//////////////////////////////////////////////////////////////////////////////////////////////////////////////////

inline BOOL CSparseStream::FreeAllocatedRanges(
    FILE_ALLOCATED_RANGE_BUFFER* pfarb) {

    // Free the queue entry's allocated memory
    return(HeapFree(GetProcessHeap(), 0, pfarb));
}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////// End Of File //////////////////////////////////////
```