

第三部分 内存管理

第13章 Windows的内存结构

操作系统使用的内存结构是理解操作系统如何运行的最重要的关键。当开始对一个新的操作系统进行操作时，你会想到一系列的问题。比如，“如何在两个应用程序之间共享数据呢？”“系统将要查找的信息存放在什么地方呢？”“如何使我的程序能够更加有效地运行呢？”等等。

很好地理解系统如何管理内存，可以帮助你更快和更准确地回答这些问题。本章将介绍Microsoft公司的Windows操作系统使用的内存结构。

13.1 进程的虚拟地址空间

每个进程都被赋予它自己的虚拟地址空间。对于32位进程来说，这个地址空间是4GB，因为32位指针可以拥有从0x00000000至0xFFFFFFFF之间的任何一个值。这使得一个指针能够拥有4 294 967 296个值中的一个值，它覆盖了一个进程的4GB虚拟空间的范围。对于64位进程来说，这个地址空间是16EB（ 10^{18} 字节），因为64位指针可以拥有从0x0000000000000000至0xFFFFFFFFFFFFFFFF之间的任何值。这使得一个指针可以拥有18 446 744 073 709 551 616个值中的一个值，它覆盖了一个进程的16EB虚拟空间的范围。这是相当大的一个范围。

由于每个进程可以接收它自己的私有的地址空间，因此当进程中的一个线程正在运行时，该线程可以访问只属于它的进程的内存。属于所有其他进程的内存则隐藏着，并且不能被正在运行的线程访问。

注意 在Windows 2000中，属于操作系统本身的内存也是隐藏的，正在运行的线程无法访问。这意味着线程常常不能访问操作系统的数据。Windows 98中，属于操作系统的内存是不隐藏的，正在运行的线程可以访问。因此，正在运行的线程常常可以访问操作系统的数据，也可以破坏操作系统（从而有可能导致操作系统崩溃）。在Windows 98中，一个进程的线程不可能访问属于另一个进程的内存。

前面说过，每个进程有它自己的私有地址空间。进程A可能有一个存放在它的地址空间中的数据结构，地址是0x12345678，而进程B则有一个完全不同的数据结构存放在它的地址空间中，地址是0x12345678。当进程A中运行的线程访问地址为0x12345678的内存时，这些线程访问的是进程A的数据结构。当进程B中运行的线程访问地址为0x12345678的内存时，这些线程访问的是进程B的数据结构。进程A中运行的线程不能访问进程B的地址空间中的数据结构。反之亦然。

当你因为拥有如此大的地址空间可以用于应用程序而兴高采烈之前，记住，这是个虚拟地址空间，不是物理地址空间。该地址空间只是内存地址的一个范围。在你能够成功地访问数据而不会出现违规访问之前，必须赋予物理存储器，或者将物理存储器映射到各个部分的地址空间。本章后面将要具体介绍这是如何操作的。

13.2 虚拟地址空间如何分区

每个进程的虚拟地址空间都要划分成各个分区。地址空间的分区是根据操作系统的基本实现方法来进行的。不同的 Windows内核，其分区也略有不同。表 13-1显示了每种平台是如何对进程的地址空间进行分区的。

表13-1 进程的地址空间如何分区

分 区	32位Windows 2000 (x86和 Alpha处理器)	32位Windows 2000(x86 w/3 GB用户方式)	64位Windows 2000 (Alpha和 IA-64处理器)	Windows 98
NULL指针分 配的分区	0x00000000 0x0000FFFF	0x00000000 0x0000FFFF	0x00000000 00000000 0x00000000 0000FFFF	0x00000000 0x000000FF
DOS/16位 Windows应用程 序兼容分区	无	无	无	0x000001000 0x003FFFFF
用户方式	0x00010000 0x7FFEFFFF	0x00010000 0xBFFEFFFFF	0x00000000 00010000 0x000003FF FFFEFFFF	0x00400000 0x7FFFFFFF
64-KB 禁止进入	0x7FFF0000 0x7FFFFFFF	0xBFFF0000 0xBFFFFFFF	0x000003FFFFFFF0000 0x000003FFFFFFFFF	无
共享内存映射 文件 (MMF) 内 核方式	无 0x80000000 0xFFFFFFFF	无 0xC0000000 0xFFFFFFFF	无 0x00000400 00000000 0xFFFFFFFF FFFFFFFF	0x80000000 0xBFFFFFFF 0xC0000000 0xFFFFFFFF

如你所见，32位Windows 2000的内核与64位Windows 2000的内核拥有大体相同的分区，差别在于分区的大小和位置有所不同。另一方面，可以看到 Windows 98下的分区有着很大的不同。下面让我们看一下系统是如何使用每一个分区的。

注意 Microsoft公司正在积极开发64位Windows 2000。但是当我撰写本书时，该系统仍在开发之中。应该使用本书中关于64位Windows 2000的信息，将它们用于你的当前项目的设计和实现中。不过应该知道，等到64位Windows 2000上市时，本章中介绍的一些详细信息很可能已经发生了变化。至于 IA-64（64位Intel结构）的内存管理，分区和系统页面大小的特定虚拟地址范围也有可能变更。

13.2.1 NULL指针分配的分区——适用于Windows 2000和Windows 98

进程地址空间的这个分区的设置是为了帮助程序员掌握 NULL指针的分配情况。如果你的进程中的线程试图读取该分区的地址空间的数据，或者将数据写入该分区的地址空间，那么CPU就会引发一个访问违规。保护这个分区是极其有用的，它可以帮助你发现 NULL指针的分配情况。

C/C++程序中常常不进行严格的错误检查。例如，下面这个代码就没有进行任何错误检查：

```
int* pnSomeInteger = (int*) malloc(sizeof(int));
*pnSomeInteger = 5;
```

如果malloc不能找到足够的内存来满足需要，它就返回 NULL。但是，该代码并不检查这种可能性，它认为地址的分配已经取得成功，并且开始访问 0x00000000地址的内存。由于这个分区的地址空间是禁止进入的，因此就会发生内存访问违规现象，同时该进程将终止运行。这个特性有助于程序员发现应用程序中的错误。

13.2.2 MS-DOS/16位Windows应用程序兼容分区——仅适用于Windows 98

进程地址空间的这个4MB分区是Windows 98需要的，目的是维护MS-DOS应用程序与16位应用程序之间的兼容性。不应该试图从32位应用程序来读取该分区的数据，或者将数据写入该分区。在理想的情况下，如果进程中的线程访问该内存，CPU应该产生一个访问违规，但是由于技术上的原因，Microsoft无法保护这个4MB的地址空间。

在Windows 2000中，16位MS-DOS与16位Windows应用程序是在它们自己的地址空间中运行的，32位应用程序不会对它们产生任何影响。

13.2.3 用户方式分区——适用于Windows 2000和Windows 98

这个分区是进程的私有（非共享）地址空间所在的地方。一个进程不能读取、写入、或者以任何方式访问驻留在该分区中的另一个进程的数据。对于所有应用程序来说，该分区是维护进程的大部分数据的地方。由于每个进程可以得到它自己的私有的、非共享分区，以便存放它的数据，因此，应用程序不太可能被其他应用程序所破坏，这使得整个系统更加健壮。

Windows 2000 在Windows 2000中，所有的.exe和DLL模块均加载这个分区。每个进程可以将这些DLL加载到该分区的不同地址中（不过这种可能性很小）。系统还可以在这个分区中映射该进程可以访问的所有内存映射文件。

Windows 98 在Windows 98中，主要的Win32系统DLL（Kernel32.dll，AdvAPI32.dll，User32.dll和GDI32.dll）均加载共享内存映射文件分区中。.exe和所有其他DLL模块则加载到这个用户方式分区中。所有进程的共享DLL均位于相同的虚拟地址中，但是其他DLL可以将这些DLL加载到用户方式分区的不同地址中（不过这种可能性不大）。另外，在Windows 98中，用户方式分区中决不会出现内存映射文件。

当我最初观察32位进程的地址空间的时候，我惊奇地发现可以使用的地址空间还不到我的进程的全部地址空间的一半。难道内核方式分区真的需要上面的一半地址空间吗？实际上回答是肯定的。系统需要这个地址空间，供内核代码、设备驱动程序代码、设备 I/O高速缓存、非页面内存池的分配和进程页面表等使用。实际上Microsoft将内核压缩到这个2GB空间之中。在64位Windows 2000中，内核终于得到了它真正需要的空间。

1. 在x86的Windows2000中获得3GB用户方式分区

多年来，编程人员一直强烈要求扩大用户方式的地址空间。为了满足这个需要，Microsoft允许x86的Windows 2000 Advanced Server版本和Windows 2000 Data Center版本将用户方式分区扩大为3GB。若要使所有进程都能够使用3GB用户方式分区和1GB内核方式分区，必须将/3GB开关附加到系统的BOOT.INI文件的有关项目中。表13-1中的“32位Windows 2000（x86 w/3GB用户方式）”这一列显示了使用3GB开关时它的地址空间是个什么样子。

在Microsoft添加/3GB开关之前，应用程序无法看到设置了高位的内存指针。一些有创意的程序员自己将这个高位用作一个标志，这个标志只对他们的应用程序具有意义。这时，当应用程序访问内存地址时，运行的代码将在内存地址被使用之前清除该指针的高位。可以想象，

当应用程序在3GB的用户方式环境中运行时，该应用程序转眼之间就会运行失败。

Microsoft不得不提出一个解决方案，以便使该应用程序能够在3GB环境中运行。当系统准备运行一个应用程序时，它要查看该应用程序是否与/LARGEADDRESSAWARE链接程序开关相链接。如果是链接的，那么应用程序就声称它并没有对内存地址执行什么特殊的操作，并且完全准备充分利用3GB 用户方式地址空间。另一方面，如果该应用程序没有与/LARGEADDRESSAWARE开关相链接，那么操作系统将保留 0x80000000至0xBFFFFFFF之间的1GB区域。这可以防止在已经设置了高位的内存地址上进行内存分配。

注意，内核已经被紧紧地压缩到了一个 2GB的分区中。当使用3GB的开关时，内核勉强地被放入一个1 GB的分区中。使用/3GB的开关，可以减少系统能够创建的线程、堆栈和其他资源的数量。此外，系统最多只能使用 16GB的RAM，而通常情况下最多可以使用 64GB的RAM，因为内核方式中没有足够的虚拟地址空间可以用来管理更多的 RAM。

注意 当操作系统创建进程的地址空间时，需要检查一个可执行的 LARGEADDRESSAWARE标志。对于DLL，系统则忽略该标志。在编写 DLL时，必须使之能够在 3 GB用户方式分区中正确地运行，否则它们的行为特性是无法确定的。

2. 在64位Windows 2000中获得2 GB用户方式分区

Microsoft发现许多编程人员需要尽可能迅速而方便地将现有的 32位应用程序移植到64位环境中去。但是，在许多源代码中，指针被视为 32位值。如果简单地重新编写应用程序，就会造成指针被截断的错误和不正确的内存访问。

然而，如果系统能够确保不对 0x000000007FFFFFFF以上的内存地址进行分配，那么应用程序就能很好地运行。当较高的 33位是0时，将64位地址截断为32位地址，不会产生任何问题。通过在地地址空间范围内运行应用程序，而这个地址空间范围将进程的可用地址空间限制为最低的GB，那么系统就能够确保这一点。

默认情况下，当启动一个 64位应用程序时，系统将保留从 0x0000000080000000开始的所有用户地址空间。这可以确保在底部的 2GB 64位地址空间中进行所有的内存分配。这就是地址空间的范围。对于大多数应用程序来说，这个地址空间足够了。若要使 64位应用程序能够访问它的全部4 TB(terabyte)用户方式分区，该应用程序必须使用 /LARGEADDRESSAWARE 链接开关来创建。

注意 当操作系统创建进程的 64位地址空间时，要检查一个可执行文件的 LARGEADDRESSAWARE标志。如果是DLL，那么系统将忽略该标志。编写DLL时，必须使之能够在整个4 TB用户方式分区中正确地运行，否则它们的行为特性将无法确定。

13.2.4 64 KB禁止进入的分区——仅适用于Windows 2000

这个位于用户方式分区上面的64 KB分区是禁止进入的，访问该分区中的内存的任何企图均将导致访问违规。Microsoft之所以保留该分区，是因为这样做将使得Microsoft能够更加容易地实现操作系统。当将内存块的地址和它的长度传递给Windows函数时，该函数将在执行它的操作之前使内存块生效。可以很容易创建类似下面这个代码（在32位Windows 2000系统上运行）：

```
BYTE bBuf[70000];
DWORD dwNumBytesWritten;
WriteProcessMemory(GetCurrentProcess(), (PVOID) 0x7FFEEE90, bBuf,
    sizeof(bBuf), &dwNumBytesWritten);
```

对于WriteProcessMemory这样的函数来说，写入的内存区是由内核方式代码来使之生效的，该代码能够成功地访问内核方式分区中的内存（32位系统上0x80000000以上的地址）。如果在0x80000000地址上存在内存，上面的函数调用就能成功地将数据写入只应该由内核方式代码访问的内存。为了防止出现这种情况，并使这个内存区迅速生效，Microsoft选择的办法是使该分区始终保持禁止进入的状态。只要试图读取或写入该分区中的内存，就一定会导致访问违规。

13.2.5 共享的MMF分区——仅适用于Windows 98

这个1GB分区是系统用来存放所有32位进程共享数据的地方。例如，系统的动态链接库Kernel32.dll、AdvAPI32.dll、User32.dll和GDI32.dll等，全部存放在这个地址空间分区中，因此，所有32位进程都能很容易同时访问它们。系统还为每个进程将DLL加载相同的内存地址。此外，系统将所有内存映射文件映射到这个分区中。内存映射文件将在第17章中详细介绍。

13.2.6 内核方式分区——适用于Windows 2000和Windows 98

这个分区是存放操作系统代码的地方。用于线程调度、内存管理、文件系统支持、网络支持和所有设备驱动程序的代码全部在这个分区加载。驻留在这个分区中的一切均可被所有进程共享。在Windows 2000中，这些组件是完全受到保护的。如果你试图访问该分区中的内存地址，你的线程将会产生访问违规，导致系统向用户显示一个消息框，并关闭你的应用程序。关于访问违规和如何处理这些违规的详细说明，请参见第23、24和25章的内容。

Windows 2000 在64位Windows 2000中，4 TB用户方式分区看上去与16,777,212 TB的内核方式分区非常不成比例。并不是内核方式分区需要使用该虚拟地址空间的全部空间，它只是说明64位地址空间是非常大的，而该地址空间的大部分是不用的。系统允许应用程序使用4 TB分区，并且允许内核使用它需要的东西，而内核方式分区的大部分是不用的。幸好系统并不需要任何内部数据结构来维护内核方式分区的不用部分。

Windows 98 不幸的是，在Windows 98中该分区中的数据是不受保护的。任何应用程序都可以从该分区读取数据，也可以写入数据，因此有可能破坏操作系统。

13.3 地址空间中的区域

当进程被创建并被赋予它的地址空间时，该可用地址空间的主体是空闲的，即未分配的。若要使用该地址空间的各个部分，必须通过调用VirtualAlloc函数（第15章介绍）来分配它里边的各个区域。对一个地址空间的区域进行分配的操作称为保留(reserving)。

每当你保留地址空间的一个区域时，系统要确保该区域从一个分配粒度的边界开始。对于不同的CPU平台来说，分配粒度是各不相同的。但是，截止到撰写本书时，所有的CPU平台（x86、32位Alpha、64位Alpha和IA-64）都使用64KB这个相同的分配粒度。

当你保留地址空间的一个区域时，系统还要确保该区域的大小是系统的页面大小的倍数。页面是系统在管理内存时使用的一个内存单位。与分配粒度一样，不同的CPU，其页面大小也是不同的。x86使用的页面大小是4 KB，而Alpha（当既能运行32位Windows 2000也能运行64位Windows 2000时）使用的页面大小则是8 KB。在撰写本书时，Microsoft预计IA-64也使用8 KB的页面。但是，如果测试显示使用更大的页面能够提高系统的总体性能，那么Microsoft可

以切换到更大的页面（16KB或更大）。

注意 有时系统能够代表你的进程来保留地址空间的区域。例如，系统可以分配一个地址空间区域，以便存放进程环境块（FEB）。FEB是由系统创建、操作和撤消的一个小型数据结构。当创建一个进程时，系统就为FEB分配一个地址空间区域。

系统也需要创建一个线程环境块（TEB），以便管理进程中当前存在的所有线程。用于这些TEB的区域将根据进程中的线程被创建和撤消等情况而保留和释放。

虽然系统规定，要求保留的地址空间区域均从分配粒度边界（目前所有平台上均为64KB）开始，但是系统本身并不受这个规定的限制。为你的进程的PEB和TEB保留的地址空间区域很可能不是从64KB这个边界开始的。不过这些保留区域仍然必须是CPU的页面大小的倍数。

如果想保留一个10KB的地址空间区域，系统将自动对你的请求进行四舍五入，使保留的地址空间区域的大小是页面大小的倍数。这意味着，在x86平台上，系统将保留一个12KB的区域，在Alpha平台上，系统将保留一个16KB的区域。

当你的程序算法不再需要访问已经保留的地址空间区域时，该区域应该被释放。这个过程称为释放地址空间的区域，它是通过调用VirtualFree函数来完成的。

13.4 提交地址空间区域中的物理存储器

若要使用已保留的地址空间区域，必须分配物理存储器，然后将该物理存储器映射到已保留的地址空间区域。这个过程称为提交物理存储器。物理存储器总是以页面的形式来提交的。若要将物理存储器提交给一个已保留的地址空间区域，也要调用VirtualAlloc函数。

当将物理存储器提交给地址空间区域时，不必将物理存储器提交给整个区域。例如，可以保留一个64KB的区域，然后将物理存储器提交给该区域中的第二和第四个页面。图13-1显示了进程的地址空间是个什么样子。注意，根据运行的CPU平台的不同，地址空间是各有差别的。左边的地址空间显示了x86计算机（它的页面大小是4KB）上的情况，而右边的地址空间则显示了Alpha计算机（它的页面大小是8KB）上发生的情况。

当你的程序算法不再需要访问保留的地址空间区域中已提交的物理存储器时，该物理存储器应该被释放。这个过程称为回收物理存储器，它是通过VirtualFree函数来完成的。

13.5 物理存储器与页文件

在较老的操作系统中，物理存储器被视为计算机拥有的RAM的容量。换句话说，如果计



图13-1 不同的CPU使用的示例进程地址空间

算机拥有16MB的RAM,那么加载和运行的应用程序最多可以使用16MB的RAM。今天的操作系统能够使得磁盘空间看上去就像内存一样。磁盘上的文件通常称为页文件,它包含了可供所有进程使用的虚拟内存。

当然,若要使虚拟内存能够运行,需要得到CPU本身的大量帮助。当一个线程试图访问一个字节的内存时,CPU必须知道这个字节是在RAM中还是在磁盘上。

从应用程序的角度来看,页文件透明地增加了应用程序能够使用的RAM(即内存)的数量。如果计算机拥有64MB的RAM,同时在硬盘上有一个100MB的页文件,那么运行的应用程序就认为计算机总共拥有164MB的RAM。

当然,实际上并不拥有164MB的RAM。相反,操作系统与CPU相协调,共同将RAM的各个部分保存到页文件中,当运行的应用程序需要时,再将页文件的各个部分重新加载到RAM。由于页文件增加了应用程序可以使用的RAM的容量,因此页文件的使用是视情况而定的。如果没有页文件,那么系统就认为只有较少的RAM可供应用程序使用。但是,我们鼓励用户使用页文件,这样他们就能够运行更多的应用程序,并且这些应用程序能够对更大的数据集进行操作。最好将物理存储器视为存储在磁盘驱动器(通常是硬盘驱动器)上的页文件中的数据。这样,当一个应用程序通过调用VirtualAlloc函数,将物理存储器提交给地址空间的一个区域时,地址空间实际上是从硬盘上的一个文件中进行分配的。系统的页文件的大小是确定有多少物理存储器可供应用程序使用时应该考虑的最重要的因素,RAM的容量则影响非常小。

现在,当你的进程中的一个线程试图访问进程的地址空间中的一个数据块时,将会发生两种情况之一,参见图13-2中的流程图。

在第一种情况中,线程试图访问的数据是在RAM中。在这种情况下,CPU将数据的虚拟内存地址映射到内存的物理地址中,然后执行需要的访问。

在第二种情况中,线程试图访问的数据不在RAM中,而是存放在页文件中的某个地方。这时,试图访问就称为页面失效,CPU将把试图进行的访问通知操作系统。这时操作系统就寻找RAM中的一个内存空页。如果找不到空页,系统必须释放一个空页。如果一个页面尚未被修改,系统就可以释放该页面。但是,如果系统需要释放一个已经修改的页面,那么它必须首先将该页面从RAM拷贝到页交换文件中,然后系统进入该页文件,找出需要访问的数据块,并将数据加载到空闲的内存页面。然后,操作系统更新它的用于指明数据的虚拟内存地址现在已经映射到RAM中的相应的物理存储器地址中的表。这时CPU重新运行生成初始页面失效的指令,但是这次CPU能够将虚拟内存地址映射到一个物理RAM地址,并访问该数据块。

系统需要将内存页面拷贝到页文件并反过来将页文件拷贝到内存页面的次数越多,你的硬盘倒腾的次数就越多,系统运行得越慢(倒腾意味着操作系统要花费更多的时间将页面从内存中转出转进,而不是将时间用于程序的运行)。因此,通过给你的计算机增加更多的RAM,就可以减少运行应用程序所需的倒腾次数,这就必然可以大大提高系统的运行速度。所以必须遵循一条基本原则,那就是要让你的计算机运行得更块,增加更多的RAM。实际上,在大多数情况下,若要提高系统的运行性能,增加RAM比提高CPU的速度所产生的效果更好。

不在页文件中维护的物理存储器

当阅读了上一节后,你必定会认为,如果同时运行许多文件的话,页文件就可能变得非常大,而且你会认为,每当你运行一个程序时,系统必须为进程的代码和数据保留地址空间的一些区域,将物理存储器提交给这些区域,然后将代码和数据从硬盘上的程序文件拷贝到页文件中已提交的物理存储器中。

实际上系统并不进行上面所说的这些操作。如果它进行这些操作的话,就要花费很长的时间来加载程序并启动它运行。相反,当启动一个应用程序的时候,系统将打开该应用程序的.exe文件,确定该应用程序的代码和数据的大小。然后系统要保留一个地址空间的区域,并指明与该区域相关联的物理存储器是在.exe文件本身中。即系统并不是从页文件中分配地址空间,而是将.exe文件的实际内容即映像用作程序的保留地址空间区域。当然,这使应用程序的加载非常迅速,并使页文件能够保持得非常小。

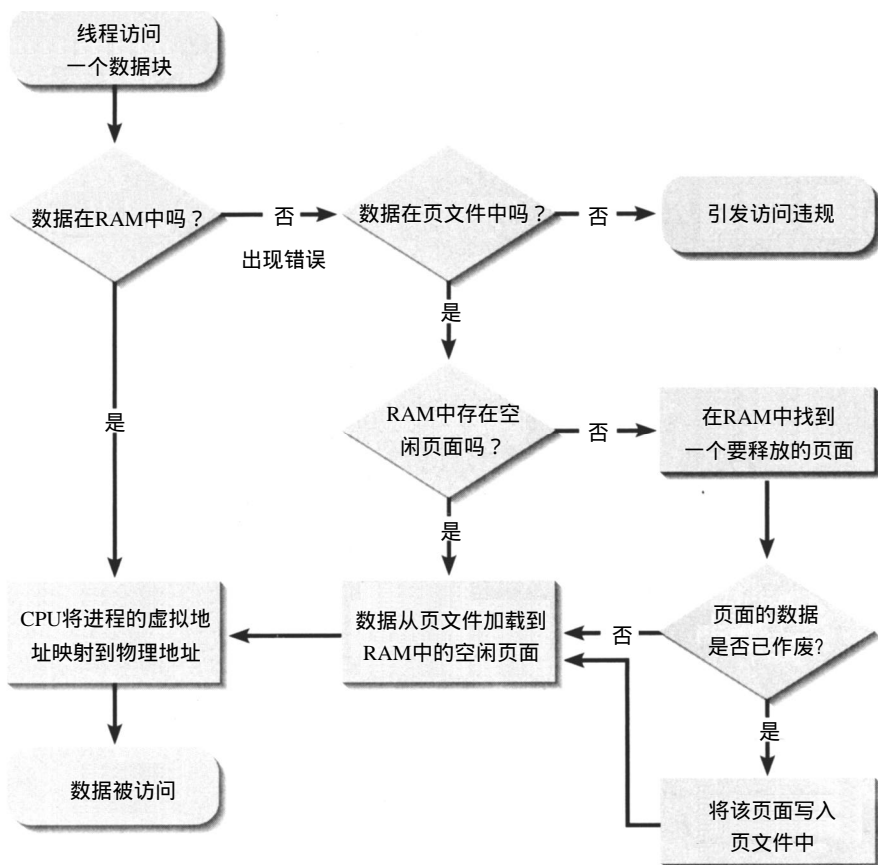


图13-2 将虚拟地址转换成物理存储器地址的流程图

当硬盘上的一个程序的文件映像（这是个.exe文件或DLL文件）用作地址空间的区域的物理存储器时,它称为内存映射文件。当一个.exe文件或DLL文件被加载时,系统将自动保留一个地址空间的区域,并将该文件映像映射到该区域中。但是,系统也提供了一组函数,使你能够将数据文件映射到一个地址空间的区域中。关于内存映射文件的详细说明,将在第17章中介绍。

Windows 2000 Windows 2000能够使用多个页文件。如果多个页文件存在于不同的物理硬盘驱动器上,系统的运行将能得快得多,因为它能够将数据同时写入多个驱动器。打开System Properties Control Panel（系统属性控制面板）小程序,再选择Advanced选项卡,单击Performance Options（性能选项）按钮,就能够添加或删除页文件。图13-3显示了该对话框的形式。

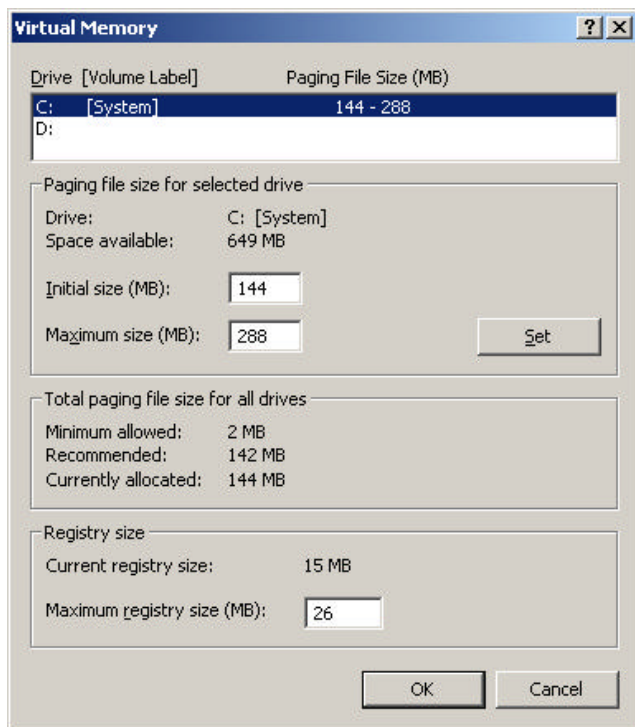


图13-3 Virtual Memory 对话框

注意 当.exe或DLL文件从软盘加载时，Windows 98和Windows 2000都能将整个文件从软盘拷贝到系统的RAM中。此外，系统将从页文件中分配足够的内存，以便存放该文件的映像。如果系统选择对当前包含该文件的一部分映像的RAM页面进行裁剪，那么该内存属于只能写入的内存。如果系统RAM上的负载比较小，那么文件始终都可以直接从RAM来运行。

Microsoft不得通过软盘来运行的映射文件，这样，安装应用程序才能正确运行。安装程序常常从一个软盘开始，然后用户将软盘从驱动器中取出来，再插入另一个软盘。如果系统需要回到第一个软盘，以便加载.exe或DLL文件的某些代码，当然该代码已经不再在软盘驱动器中了。然而，由于系统将文件拷贝到RAM（并且受页文件的支持），要访问安装程序是不会有问题的。

系统并不将RAM映射文件拷贝在其他可换式介质上，如光盘或网络驱动器，除非映射文件是用/SWAPRUN：CD或/SWAPRUN：NET开关链接的。注意，Windows 98不支持/SWAPRUN映像标志。

13.6 保护属性

已经分配的物理存储器的各个页面可以被赋予不同的保护属性。表 13-2显示了这些保护属性。

x86和Alpha CPU不支持“执行”保护属性，不过操作系统软件却支持这个属性。这些CPU将读访问视为执行访问。这意味着如果将PAGE_EXECUTE保护属性赋予内存，那么该内存也将拥有读优先权。当然，不应该依赖这个行为特性，因为在其他CPU上的Windows实现代码很

表13-2 页面的保护属性

保护属性	描述
PAGE_NOACCESS	如果试图在该页面上读取、写入或执行代码，就会引发访问违规
PAGE_READONLY	如果试图在该页面上写入或执行代码，就会引发访问违规
PAGE_READWRITE	如果试图在该页面上执行代码，就会引发访问违规
PAGE_EXECUTE	如果试图在该页面上对内存进行读取或写入操作，就会引发访问违规
PAGE_EXECUTE_READ	如果试图在该页面上对内存进行写入操作，就会引发访问违规
PAGE_EXECUTE_READWRITE	对于该页面不管执行什么操作，都不会引发访问违规
PAGE_WRITECOPY	如果试图在该页面上执行代码，就会引发访问违规。如果试图在该页面上写入内存，就会导致系统将它自己的私有页面（受页文件的支持）拷贝赋予该进程
PAGE_EXECUTE_WRITECOPY	对于该地址空间的区域，不管执行什么操作，都不会引发访问违规。如果试图在该页面上的内存中进行写入操作，就会将它自己的私有页面（受页文件的支持）拷贝赋予该进程

可能将“执行”保护视为“仅为执行”保护。

Windows 98 Windows 98只支持PAGE_NOACCESS、PAGE_READONLY和PAGE_READWRITE等保护属性。

13.6.1 Copy-On-Write 访问

表13-2列出的保护属性都是非常容易理解的，不过最后两个属性需要作一些说明。一个是PAGE_WRITECOPY，另一个是PAGE_EXECUTE_WRITECOPY。这两个属性的作用是为了节省RAM的使用量和页文件的空间。Windows支持一种机制，使得两个或多个进程能够共享单个内存块。因此，如果10个Notepad实例正在运行，那么所有实例可以共享应用程序的代码和数据页面。让所有实例共享同样的内存页面将能够大大提高系统的性能，但是这要求所有实例都将该内存视为只读或只执行的内存。如果一个实例中的线程将数据写入内存修改它，那么其他实例看到的这个内存也将被修改，从而造成一片混乱。

为了防止出现这种混乱，操作系统给共享内存块赋予了Copy-On-Write保护属性。当一个.exe或DLL模块被映射到一个内存地址时，系统将计算有多少页面是可以写入的（通常包含代码的页面标为PAGE_EXECUTE_READ，而包含数据的页面则标为PAGE_READWRITE）。然后，系统从页文件中分配内存，以适应这些可写入的页面的需要。除非该模块的可写入页面是实际的写入模块，否则这些页文件内存是不使用的。

当一个进程中的线程试图将数据写入一个共享内存块时，系统就会进行干预，并执行下列操作步骤：

- 1) 系统查找RAM中的一个空闲内存页面。注意，当该模块初次被映射到进程的地址空间时，该空闲页面将被页文件中已分配的页面之一所映射。当该模块初次被映射时，由于系统要分配所有可能需要的页文件，因此这一步不可能运行失败。

- 2) 系统将试图被修改的页面内容拷贝到第一步中找到的页面。该空闲页面将被赋予PAGE_READWRITE或PAGE_EXECUTE_READWRITE保护属性。原始页面的保护属性和数据不发生任何变化。

- 3) 然后系统更新进程的页面表，使得被访问的虚拟地址被转换成新的RAM页面。

当系统执行了这3个操作步骤之后，该进程就可以访问它自己的内存页面的私有实例。第

17章还要详细地介绍共享内存和Copy-On-Write保护属性。

此外，当使用 VirtualAlloc函数来保留地址空间或者提交物理存储器时，不应该传递 PAGE_WRITECOPY或PAGE_EXECUTE_WRITECOPY。如果传递的话，将会导致VirtualAlloc调用的失败。对 GetLastError的调用将返回 ERROR_INVALID_PARAMETER。当操作系统映射.exe或DLL文件映像时，这两个属性将被操作系统使用。

Windows 98 Windows 98不支持Copy-On-Write保护。当Windows 98发现需要Copy_On_Write保护时，它就立即进行数据的拷贝，而不是等待试图对内存进行写入操作。

13.6.2 特殊的访问保护属性的标志

除了上面介绍的保护属性外，还有 3 个保护属性标志，即 PAGE_NOCACHE，PAGE_WRITECOMBINE和PAGE_GUARD。可以用OR逐位将它们连接，以便将这 3 个标志用于任何一个保护属性（PAGE_NOCACHE除外）。

第一个保护属性标志PAGE_NOCACHE用于停用已提交页面的高速缓存。一般情况下最好不要使用该标志，因为它主要是供需要处理内存缓冲区的硬件设备驱动程序的开发人员使用的。

第二个保护属性PAGE_WRITECOMBINE也是供设备驱动程序开发人员使用的。它允许把单个设备的多次写入合并在一起，以便提高运行性能。

最后一个保护属性标志PAGE_GUARD可以在页面上写入一个字节时使用应用程序收到一个通知（通过一个异常条件）。该标志有一些非常巧妙的用法。Windows 2000在创建线程堆栈时使用该标志。关于该标志的详细说明，参见第16章。

Windows 98 Windows 98将忽略PAGE_NOCACHE、PAGE_WRITECOMBINE和PAGE_GUARD这3个保护属性标志。

13.7 综合使用所有的元素

本节要将地址空间、分区、区域、内存块和页面等元素综合起来加以使用。为了更好地说明问题，我们首先来看一看虚拟内存表，它可以显示单个进程中的所有地址空间的区域。该进程正好是第14章中介绍的VMMMap示例应用程序。为了全面了解进程的地址空间，首先要介绍一下当在32位x86计算机上的Windows 2000下运行VMMMap时，地址空间是个什么样子。表13-3显示了一个示例地址空间表。后面将要介绍 Windows 2000与Windows 98的地址空间之间的差别。

表13-3中的地址空间表显示了进程的地址空间中的各个不同区域。每行显示一个区域，每行包含6列。

第一列，即最左边的一列显示了区域的基地址。你会发现我们是从地址为 0x00000000的区域开始观察进程的地址空间的，并在可用地址空间的最后一个区域结束，该区域的起始地址是 0x7FFE0000。所有区域都是相邻的。你也会注意到，非空闲区域的所有基地址几乎都是从64KB的倍数上开始的。这是由系统采用的保留地址空间的分配粒度所决定的。不是从分配粒度边界开始的区域，表示该区域是由操作系统代码代表你的进程来分配的。

表13-3 显示了32位x86计算机上运行的Windows 2000
下的地址空间区域的示例地址空间表

基 地 址	类 型	大 小	块	保护属性	描 述
00000000	空闲	65 536			
00010000	私有	4096	1	-RW-	
00011000	空闲	61 440			
00020000	私有	4096	1	-RW-	
00021000	空闲	61 440			
00030000	私有	1 048 576	3	-RW-	线程堆栈
00130000	私有	1 048 576	2	-RW-	
00230000	映射	65 536	2	-RW-	
00240000	映射	90 112	1	-RW-	\Device\HarddiskVolume1\WINNT\system32\unicode.nls
00256000	空闲	40 960			
00260000	映射	208 896	1	-R--	\Device\HarddiskVolume1\WINNT\system32\locale.nls
00293000	空闲	53248			
002A0000	映射	266 240	1	-R--	\Device\HarddiskVolume1\WINNT\system32\sortkey.nls
002E1000	空闲	61 400			
002F0000	映射	16 384	1	-R--	\Device\HarddiskVolume1\WINNT\system32\sorttbls.nls
002F4000	空闲	49 152			
00300000	映射	819 200	4	ER--	
003C8000	空闲	229 376			
00400000	映像	106 496	5	ERWC	C:\CD\x86\Debug\14\VMMap.exe
0041A000	空闲	24 576			
00420000	映像	274 432	1	-R--	
00463000	空闲	53 248			
00470000	映像	3145 728	2	ER--	
00770000	私有	4096	1	-RW-	
00771000	空闲	61 440			
00780000	私有	4096	1	-RW-	
00781000	空闲	61 440			
00790000	私有	65 536	2	-RW-	
007A0000	映射	8192	1	-R--	\Device\HarddiskVolume1\WINNT\system32\ctype.nls
007A2000	空闲	1 763 893 248			
699D0000	映像	45 056	4	ERWC	C:\WINNT\System32\PSAPI.dll
699DB000	空闲	238 505 984			
77D50000	映像	450 560	4	ERWC	C:\WINNT\System32\RPCRT4.dll
77DBE000	空闲	8192			
77DC0000	映像	344 064	5	ERWC	C:\WINNT\System32\ADVAPI32.dll
77E14000	空闲	49152			
77E20000	映像	401 408	4	ERWC	C:\WINNT\System32\USER32.dll
77E82000	空闲	57 344			
77E90000	映像	720 896	5	ERWC	C:\WINNT\System32\KERNEL32.dll
77F40000	映像	241 664	4	ERWC	C:\WINNT\System32\GDI32.dll
77F7B000	空闲	20 480			

(续)

基 地 址	类 型	大 小	块	保护属性	描 述
77F80000	映像	483 328	5	ERWC	C:\WINNT\System32\ntdll.dll
77FF6000	空闲	40 960			
78000000	映像	290 816	6	ERWC	C:\WINNT\System32\MSVCRT.dll
78047000	空闲	124 424 192			
7F6F0000	映射	1 048 576	2	ER--	
7F7F0000	空闲	8 126 464			
7FFB0000	映射	147 456	1	-R--	
7FFD4000	空闲	40 960			
7FFDE000	私有	4096	1	ERW-	
7FFDF000	私有	4096	1	ERW-	
7FFE0000	私有	65 536	2	-R--	

第二列显示了区域的类型。区域类型共有 4 个值，即空闲，私有，映像或映射。表 13-4 对它们进行了介绍。

表13-4 区域类型说明

类 型	说 明
空闲	该区域的虚拟地址不受任何内存的支持。该地址空间没有被保留。应用程序既可以将一个区域保留在显示的基地址上，也可以保留在空闲区域中的任何位置上
私有	该区域的虚拟地址将受系统的页文件的支持。
映像	该区域的虚拟地址原先受内存映射的映像文件（如 .exe 或 DLL 文件）的支持，但也许不再受映像文件的支持。例如，当写入模块映像中的全局变量时，“写入时拷贝”的机制将由页文件来支持特定的页面，而不是受原始映像文件的支持
映射	该区域的虚拟地址原先是受内存映射的数据文件的支持，但也许不再受数据文件的支持。例如，数据文件可以使用“写入时拷贝”的保护属性来映射。对文件的任何写入操作都将导致页文件而不是原始数据支持特定的页面

我的VMMMap应用程序计算这一列的方法可能产生错误的结果。当地址空间区域不空闲时，VNNap示例应用程序就要猜测剩余的 3 个值中哪一个可以使用。没有一个函数可以调用，以便确定该区域的准确用途。这一列的值的方法是，对区域中的所有内存块进行扫描，然后进行合乎逻辑的推测。可以参考第14章中的代码，以便更好地了解计算方法。

第三列显示了为该区域保留的字节数量。例如，系统将 User.DLL 的映像映射到内存地址 0x77E20000。当系统为该映像保留地址空间时，它必须保留 401 408 个字节。第三列中的数字总是CPU的页面大小的倍数（x86 CPU 为 4096 字节）。

第四列显示了保留区域中的块的数量。所谓块是指一组相邻的页面，它们拥有相同的保护属性，并且都是受相同类型的物理存储器支持的。下一节将要详细介绍这个问题。对于空闲区域来说，这个值始终都是 0，因为在空闲区域中不能提交任何内存（在第四列中空闲区域不显示任何信息）。对于非空闲区域来说，这个值可以是 1 到区域大小/页面大小的最大数字之间的任何值。例如，从内存地址 0x77E20000 开始的区域，它的区域大小是 401 408 个字节。由于该进程是在 x86 CPU 上运行的（x86 CPU 的页面大小是 4096 个字节），因此提交的各种不同的块的最大数量是 98（401 408/4096）。表中显示该区域中的块的数量是 4。

第五列显示了区域的保护属性。各个字母所代表的含义是：E=执行，R=读取，W=写入，C=写入时拷贝。如果一个区域没有显示任何保护属性，那么该区域就没有访问保护。空闲区域没有显示任何保护属性，因为未保留区域不拥有与其相关联的保护属性。这里决不会出现

GUARD保护属性标志或NO-CACHE保护属性标志。只有当这些标志与物理存储器相关联，而不是与保留的地址空间相关联时，它们才具有意义。给一个区域赋予保护属性的目的只是为了提高效率，并且总是会被赋予物理存储器的保护属性所取代。

第六列即最后一列显示了区域中的信息的文字描述。如果是空闲区域，那么这一列总是空的。如果是私有区域，它通常也是空的，因为 VMMMap没有办法知道应用程序为什么要保留这个私有地址空间区域。但是，VMMMap能够识别包含线程堆栈的私有区域。VMMMap通常能够发现线程堆栈，因为它们通常拥有一个具有 GUARD保护属性的物理存储器块。不过，当线程堆栈满了的时候，它就不再拥有一个保护属性为 GUARD的物理存储器块，同时VMMMap将无法发现它。

对于映像区域来说，VMMMap将显示映射到该区域中的文件的全路径名。使用 ToolHelp函数，VMMMap就可以获得该信息。在 Windows 2000中，通过调用 GetMappedFileName函数 (Windows 98中没有这个函数)，VMMMap就能够显示受数据文件支持的区域。

13.7.1 区域的内部情况

我们还可以将区域划分得比表 13-3显示的情况更细一些。表 13-5显示的地址空间与表 13-3所示的地址空间相同，但是它也显示了每个区域中包含的内存块。

表13-5 显示了32位x86计算机上的Windows 2000下的
内存区域和块的示例地址空间表

基 地 址	类 型	大 小	块	保护属性	描 述
00000000	空闲	65 536			
00010000	私有	4096	1	-RW-	
00010000	私有	4096		-RW- ---	
00011000	空闲	61 440			
00020000	私有	4096	1	-RW-	
00020000	私有	4096		-RW- ---	
00021000	空闲	61 440			
00030000	私有	1 048 576	3	-RW-	线程堆栈
00030000	保留	905 216		-RW- ---	
0010D000	私有	4096		-RW- G--	
0010E000	私有	139 264		-RW- ---	
00130000	私有	1048576	2	-RW-	
00130000	私有	36 864		-RW- ---	
00139000	保留	1 011 712		-RW- ---	
00230000	映射	65 536	2	-RW-	
00230000	映射	4096		-RW- ---	
00231000	保留	61 440		-RW- ---	
00240000	映射	90 112	1	-R--	\Device\HarddiskVolume\WINNT\system32\unicode.nls
00240000	映射	90 112		-R-- ---	
00256000	空闲	40 960			
00260000	映射	208 896	1	-R--	\Device\HarddiskVolume\WINNT\system32\locale.nls
00260000	映射	208 896		-R-- ---	
00293000	空闲	53248			

(续)

基 地 址	类 型	大 小	块	保护属性	描 述
002A0000	映射	266240	1	-R--	\Device\HarddiskVolume1\WINNT\system32\sortkey.nls
002A0000	映射	266240		-R-- ---	
002E1000	空闲	61 440			
002F0000	映射	16 384	1	-R--	\Device\HarddiskVolume1\WINNT\system32\sorttbls.nls
002F0000	映射	16 384		-R-- ---	
002F4000	空闲	49 152			
00300000	映射	819 200	4	ER--	
00300000	映射	16 384		ER-- ---	
00304000	保留	770 048		ER-- ---	
003C0000	映射	8 192		ER-- ---	
003C2000	保留	24 576		ER-- ---	
003C8000	空闲	229 376			
00400000	映像	106 496	5	ERWC	C:\CD\x86\Debug\14_VMMMap.exe
00400000	映像	4096		-R-- ---	
00401000	映像	81 920		ER-- ---	
00415000	映像	4096		-R-- ---	
00416000	映像	8 192		-RW- ---	
00418000	映像	8 192		-R-- ---	
0041A000	空闲	24 576			
00420000	映射	274 432	1	-R--	
00420000	映射	274 432		-R-- ---	
00463000	空闲	53 248			
00470000	映射	3 145 728	2	ER--	
00470000	映射	274 432		ER-- ---	
004B3000	保留	2 871 296		ER-- ---	
00770000	私有	4096	1	-RW-	
00770000	私有	4096		-RW- ---	
00771000	空闲	61 440			
00780000	私有	4096	1	-RW-	
00780000	私有	4096		-RW- ---	
00781000	空闲	61 440			
00790000	私有	65 536	2	-RW-	
00790000	私有	20 480		-RW- ---	
00795000	保留	45 056		-RW- ---	
007A0000	映射	8 192	1	-R--	\Device\HarddiskVolume1\WINNT\system32\ctype.nls
007A0000	映射	8 192		-R-- ---	
007A2000	空闲	57 344			
007B0000	私有	524 288	2	-RW-	
007B0000	私有	4096		-RW- ---	
007B1000	保留	520 192		-RW- ---	
00830000	空闲	1 763 311 616			
699D0000	映像	45 056	4	ERWC	C:\WINNT\System32\PSAPI.dll
699D0000	映像	4096		-R-- ---	
699D1000	映像	16 384		ER-- ---	

(续)

基 地 址	类 型	大 小	块	保护属性	描 述
699D5000	映像	16 384		-RWC ---	
699D9000	映像	8192		-R-- ---	
699DB000	空闲	238 505 984			
77D50000	映像	450 560	4	ERWC	C:\WINNT\system32\RPCRT4.DLL
77D50000	映像	4096		-R-- ---	
77D51000	映像	421 888		ER-- ---	
77DB8000	映像	4096		-RW- ---	
77DB9000	映像	20 480		-R-- ---	
77DBE000	空闲	8192			
77DC0000	映像	344 064	5	ERWC	C:\WINNT\system32\ADVAPI32.dll
77DC0000	映像	4096		-R-- ---	
77DC1000	映像	307 200		ER-- ---	
77E0C000	映像	4096		-RW- ---	
77E0D000	映像	4096		-RWC ---	
77E0E000	映像	24 576		-R-- ---	
77E14000	空闲	49 152			
77E20000	映像	401 408	4	ERWC	C:\WINNT\system32\USER32.dll
77E20000	映像	4096		-R-- ---	
77E21000	映像	348 160		ER-- ---	
77E76000	映像	4096		-RW- ---	
77E77000	映像	45 056		-R-- ---	
77E82000	空闲	57 344			
77E90000	映像	720 896	5	ERWC	C:\WINNT\system32\KERNEL32.dll
77E90000	映像	4096		-R-- ---	
77E91000	映像	368 640		ER-- ---	
77EEB000	映像	8192		-RW- ---	
77EED000	映像	4096		-RWC ---	
77EEE000	映像	335 872		-R-- ---	
77F40000	映像	241 664	4	ERWC	C:\WINNT\system32\GDI32.DLL
77F40000	映像	4096		-R-- ---	
77F41000	映像	221 184		ER-- ---	
77F77000	映像	4096		-RW- ---	
77F78000	映像	12 288		--R-- ---	
77F7B000	空闲	20 480			
77F80000	映像	483 328	5	ERWC	C:\WINT\system32\ntdll.dll
77F80000	映像	4096		-R-- ---	
77F81000	映像	299 008		ER-- ---	
77FCA000	映像	8192		-RW- ---	
77FCC000	映像	4096		-RWC ---	
77FCD000	映像	167 936		-R-- ---	
77FF6000	空闲	40 960			
78000000	映像	290 816	6	ERWC	C:\WINNT\system32\MSVCRT.dll
78000000	映像	4096		-R-- ---	
78001000	映像	208 896		ER-- ---	
78034000	映像	32 768		-R-- ---	
7803C000	映像	12 288		-RW- ---	
7803F000	映像	16 384		RWC- ---	

(续)

基 地 址	类 型	大 小	块	保护属性	描 述
78043000	映像	16 384		-R-- ---	
78047000	空闲	124 424 192			
7F6F0000	映射	1 048 576	2	ER--	
7F6F0000	映射	28 672		ER-- ---	
7F6F7000	保留	1 019 904		ER-- ---	
7F7F0000	空闲	8 126 464			
7FFB0000	映射	147 456	1	-R--	
7FFB0000	映射	147 456		-R-- ---	
7FFD4000	空闲	40 960			
7FFDE000	私有	4096	1	ERW-	
7FFDE000	私有	4096		ERW- ---	
7FFDF000	私有	4096	1	ERW-	
7FFDF000	私有	4096		ERW- ---	
7FFE0000	私有	65 536	2	-R--	
7FFE0000	私有	4096		-R-- ---	
7FFE1000	保留	61 440		-R-- ---	

当然，空闲区域根本不会扩展，因为它们里面没有已经提交的内存页面。每个内存块的行显示4列，下面介绍它们的具体情况。

第一列显示一组页面的地址，这些页面具有相同的状态和保护属性。例如，具有只读保护属性的内存的单个页面（4096字节）被提交的地址是0x77E20000。在地址0x77E21000上，有一个85页（348 160字节）的已提交内存块，它具有执行和读保护特性。如果这两个内存块具有相同的保护属性，那么它们就被组合起来，在内存表中显示为一个86个页面（352 256字节）的项目。

第二列显示的是何种类型的物理存储器支持保留区域中的内存块。这一列中可以出现 5 个值中的一个。这 5 个值是空闲，私有，映射，映像和保留。如果这个值是私有、映射或映像，则表示内存块是分别受页文件、数据文件或加载的 .exe 或 DLL 文件中的物理存储器支持的。如果这个值是空闲或保留，那么该内存块根本没有任何物理存储器的支持。

大多数情况下，相同类型的物理存储器支持单个区域中的所有提交的内存块。但是单个区域中不同的已提交内存块可以受不同类型的物理存储器的支持。例如，内存映射的文件映像可以受 .exe 或 DLL 文件的支持。如果要在该区域中写入拥有 PAGE_WRITECOPY 或 PAGE_EXECUTE_WRITECOPY 保护属性的单个页面，那么系统就会使你的进程成为一个由页文件而不是文件映像支持的私有页面拷贝。这个新页面拥有的属性将与没有 copy_on_write 保护属性的原始页面相同。

第三列显示了地址空间块的大小。一个区域中的所有地址空间块都是相邻的，块与块之间没有任何空隙。

第四列显示保留区域中的块的数量。

第五列显示块的保护属性和保护属性标志。块的保护属性优先于包含该块的区域的保护属性。块使用的保护属性与区域的保护属性相同，但是，与区域不关联的保护属性标志 PAGE_GUARD、PAGE_NOCACHE 和 PAGE_WRITECOMBINE 可以与块相关联。

13.7.2 与Windows 98地址空间的差别

表13-6显示了在 Windows 98 下运行相同的 VMMAP 程序时的地址空间表。为了节省篇幅，

我们没有显示0x80018000至0x85620000之间的虚拟地址。

表13-6 显示Windows 98 下地址空间区域内块的示例地址空间表

基 地 址	类 型	大 小	块	保护属性	描 述
00000000	空闲	4 194 304			
00400000	私有	131 072	6	----	C:\CD\X86\DEBUG\14 VMMAPEX.E
00400000	私有	8192		-R- - -	
00402000	私有	8192		-RW- - -	
00404000	私有	73 728		-R- - -	
00416000	私有	8192		-RW- - -	
00418000	私有	8192		-R- - -	
0041A000	保留	24 576		-----	
00420000	私有	1 114 112	4	----	
00420000	私有	20 480		-RW- - -	
00425000	保留	1 028 096		-----	
00520000	私有	4096		-RW- - -	
00521000	保留	61440		-----	
00530000	私有	65 536	2	-RW- -	
00530000	私有	4096		-RW- - -	
00531000	保留	61 440		-RW- - -	
00540000	私有	1 179 648	6	----	线程堆栈
00540000	保留	942 080		-----	
00626000	私有	4096		-RW- - -	
00627000	保留	24576		-----	
0062D000	私有	4096		-----	
0062E000	私有	139 264		-RW- - -	
00650000	保留	65 536		-----	
00660000	私有	1 114 112	4	----	
00660000	私有	20 480		-RW- - -	
00665000	保留	1 028 096		-----	
00760000	私有	4096		-RW- - -	
00761000	保留	61 440		-----	
00770000	私有	1 048 576	2	-RW- -	
00770000	私有	32 768		-RW- - -	
00778000	保留	1 015 808		-RW- - -	
00870000	空闲	2 004 418 560			
7800000	私有	262114	3	----	C:\WINDOWS\SYSTEM\MSVCRT.DLL
78000000	私有	188 416		-R- - -	
7802E000	私有	57 344		-RW- - -	
7803C000	私有	16 384		-R- - -	
78040000	空闲	133 955 584			
80000000	私有	4096	1	----	
80000000	保留	4096		-----	
80001000	私有	4096	1	----	
80001000	私有	4096		-RW- - -	
80002000	私有	4096	1	----	
80002000	私有	4096		-RW- - -	
80003000	私有	4096	1	----	
80003000	私有	4096		-RW- - -	
80004000	私有	65 536	2	----	

(续)

基 地 址	类 型	大 小	块	保护属性	描 述
80004000	私有	32 768		-RW- ---	
8000C000	保留	32 768		---- ---	
80014000	私有	4096	1	----	
80014000	私有	4096		-RW- ---	
80015000	私有	4096	1	----	
80015000	私有	4096		-RW- ---	
80016000	私有	4096	1	----	
80016000	私有	4096		-RW- ---	
80017000	私有	4096	1	----	
80017000	私有	4096		-RW- ----	
85620000	空闲	9 773 056			
85F72000	私有	151 552	1	----	
85F72000	私有	151 552		-R-- ---	
85F97000	私有	327 680	1	----	
85F97000	私有	327 680		-R-- ---	
85FE7000	空闲	22 052 864			
874EF000	私有	4194304	1	----	
874EF000	保留	4 194 304		---- ---	
878EF000	空闲	679 219 200			
B00B0000	私有	880 640	3	----	
B00B0000	私有	233 472		-R-- ---	
B00E9000	私有	20 480		-RW- ---	
B00EE000	私有	626 288		-R-- ---	
B0187000	空闲	177 311 744			
BAAA0000	私有	315 392	7	----	
BAAA0000	私有	4096		-R-- ---	
BAAA1000	私有	4096		-RW- ---	
BAAA2000	私有	241 664		-R-- ---	
BAADD000	私有	4096		-RW- ---	
BAADE000	私有	4096		-R-- ---	
BAADF000	私有	32 768		-RW- ---	
BAAE7000	私有	24 576		-R-- ---	
BAAED000	空闲	86 978 560			
BFDE0000	私有	20 480	1	----	
BFDE0000	私有	20 480		-R-- ---	
BFDE5000	空闲	45 056			
BFD00000	私有	65 536	3	----	
BFD00000	私有	40 960		-R-- ---	
BFDFA000	私有	4096		-RW- ---	
BFD0FB000	私有	20 480		-R-- ---	
BFE00000	空闲	131 072			
BFE20000	私有	16 384	3	----	
BFE20000	私有	8192		-R-- ---	
BFE22000	私有	4096		-RW- ---	
BFE23000	私有	4096		-R-- ---	
BFE24000	空闲	245 760			
BFE60000	私有	24 576	3	----	

(续)

基 地 址	类 型	大 小	块	保护属性	描 述
BFE60000	私有	8192		-R-- ---	
BFE62000	私有	4096		-RW- ---	
BFE63000	私有	12 288		-R-- ---	
BFE66000	空闲	40 960			
BFE70000	私有	24 576	3	----	
BFE70000	私有	8192		-R-- ---	
BFE72000	私有	4096		-RW- ---	
BFE73000	私有	12 288		-R-- ---	
BFE76000	空闲	40 960			
BFE80000	私有	65 536	3	----	C:\WINDOWS\SYSTEM\ADVAPI32.DLL
BFE80000	私有	49 152		-R-- ---	
BFE8C000	私有	4096		-RW- ---	
BFE8D000	私有	12 288		-R-- ---	
BFE90000	私有	573 440	3	----	
BFE90000	私有	425 984		-R-- ---	
BFEF8000	私有	4096		-RW- ---	
BFEF9000	私有	143 360		-R-- ---	
BFF1C000	空闲	16 384			
BFF20000	私有	155 648	5	----	C:\WINDOWS\SYSTEM\GDI32.DLL
BFF20000	私有	126 976		-R-- ---	
BFF3F000	私有	8192		-RW- ---	
BFF41000	私有	4096		-R-- ---	
BFF42000	私有	4096		-RW- ---	
BFF43000	私有	12 288		-R-- ---	
BFF46000	空闲	40 960			
BFF50000	私有	69 632	3	----	C:\WINDOWS\SYSTEM\USER32.DLL
BFF50000	私有	53 248		-R-- ---	
BFF5D000	私有	4096		-RW- ---	
BFF5E000	私有	12 288		-R-- ---	
BFF61000	空闲	61 440			
BFF70000	私有	585 728	5	----	C:\WINDOWS\SYSTEM\KERNEL32.DLL
BFF70000	私有	352 256		-R-- ---	
BFFC6000	保留	12 288		----	
BFFC9000	私有	16 384		-RW- ---	
BFFCD000	私有	90 112		-R-- ---	
BFFE3000	保留	114 688		----	
BFFFF000	空闲	4096			

两个地址空间表的最大不同是在 Windows 98 下缺少了某些的信息。例如，每个区域和块能反映出地址空间的区域是空闲、保留还是私有的。你决不会看到映射或者映像之类的字样，因为 Windows 98 没有提供更多的信息来指明支持该区域的物理存储器的是个内存映射文件还是包含在 .exe 或 DLL 中的文件映像。

你会发现大多数地址空间区域的大小是分配粒度（64KB）的倍数。如果包含在地址空间区域中的块的大小不是分配粒度的倍数，那么在地址空间区域的结尾处常常有一个保留的地址空间块。这个地址空间块的大小必须使得地址空间区域能够符合分配粒度边界（64KB）倍数的要求。例如，从地址 0x00530000 开始的地址空间区域包含两个地址块，一个是 4 KB 的已提

交内存块，另一个是占用 60 KB 内存地址范围的已保留的地址块。

最后，保护标志从来不反映执行或 copy-on-write 访问权，因为 Windows 98 不支持这些标志。它也不支持 3 个保护属性标志，即 PAGE_NO_CACHE、PAGE_WRITECOMBINE 和 PAGE_GUARD。由于不支持 PAGE_GUARD 标志，因此 VMMMap 使用更加复杂的技术来确定是否已经为线程的堆栈保留了地址空间区域。

你将注意到，与 Windows 2000 不同，在 Windows 98 中，0x80000000 至 0xBFFFFFFF 之间的地址空间区域是可以查看的。这个分区包含了所有 32 位应用程序共享的地址空间。如你所见，有 4 个系统 DLL 被加载了这个地址空间区域，可以供所有进程使用。

13.8 数据对齐的重要性

本节不再讨论进程的虚拟地址空间问题，而是要介绍数据对齐的重要性。数据对齐并不是操作系统的内存结构的一部分，而是 CPU 结构的一部分。

当 CPU 访问正确对齐的数据时，它的运行效率最高。当数据大小的数据模数的内存地址是 0 时，数据是对齐的。例如，WORD 值应该总是从被 2 除尽的地址开始，而 DWORD 值应该总是从被 4 除尽的地址开始，如此等等。当 CPU 试图读取的数据值没有正确对齐时，CPU 可以执行两种操作之一。即它可以产生一个异常条件，也可以执行多次对齐的内存访问，以便读取完整的未对齐数据值。

下面是访问未对齐数据的某个代码：

```
VOID SomeFunc(PVOID pvDataBuffer) {  
  
    // The first byte in the buffer is some byte of information  
    char c = * (PBYTE) pvDataBuffer;  
  
    // Increment past the first byte in the buffer  
    pvDataBuffer = (PVOID)((PBYTE) pvDataBuffer + 1);  
  
    // Bytes 2-5 contain a double-word value  
    DWORD dw = * (DWORD *) pvDataBuffer;  
  
    // The line above raises a data misalignment exception on the Alpha  
    :  
    :  
}
```

显然，如果 CPU 执行多次内存访问，应用程序的运行速度就会放慢。在最好的情况下，系统访问未对齐的数据所需要的时间将是访问对齐数据的时间的两倍，不过在有些情况下，访问时间可能更长。为了使应用程序获得最佳的运行性能，编写的代码必须使数据正确地对齐。

下面让我们更加深入地说明 x86 CPU 是如何进行数据对齐的。X86 CPU 的 EFLAGS 寄存器中包含一个特殊的位标志，称为 AC（对齐检查的英文缩写）标志。按照默认设置，当 CPU 首次加电时，该标志被设置为 0。当该标志是 0 时，CPU 能够自动执行它应该执行的操作，以便成功地访问未对齐的数据值。然而，如果该标志被设置为 1，每当系统试图访问未对齐的数据时，CPU 就会发出一个 INT 17H 中断。x86 的 Windows 2000 和 Windows 98 版本从来不改变这个 CPU 标志位。因此，当应用程序在 x86 处理器上运行时，你根本看不到应用程序中出现数据未对齐的异常条件。

现在让我们来看一看 Alpha CPU 的情况。Alpha CPU 不能自动处理对未对齐数据的访问。

当未对齐的数据访问发生时，CPU就会将这一情况通知操作系统。这时，Windows 2000将会确定它是否应该引发一个数据未对齐异常条件。它也可以执行一些辅助指令，对问题默默地加以纠正，并让你的代码继续运行。按照默认设置，当在 Alpha 计算机上安装 Windows 2000 时，操作系统会对未对齐数据的访问默默地进行纠正。然而，可以改变这个行为特性。当引导 Windows 2000 时，系统就会在注册表中查找的这个关键字：

```
HKEY_LOCAL_MACHINE\CurrentControlSet\Control\Session Manager
```

在这个关键字中，可能存在一个值，称为 EnableAlignmentFaultExceptions。如果这个值不存在（这是通常的情况），Windows 2000 会默默地处理对未对齐数据的访问。如果存在这个值，系统就能获取它的相关数据值。如果数据值是 0，系统会默默地进行访问的处理。如果数据值是 1，系统将不执行默默的处理，而是引发一个未对齐异常条件。几乎从来都不需要修改该注册表值的数据值，因为如果修改有些应用程序能够引发数据未对齐的异常条件并终止运行。

为了更加容易地修改该注册表项。Alpha 处理器上运行的 Microsoft Visual C++ 版本包含了一个小型实用程序 AXPAlign.exe。AXPAlign 的用法如下面所示：

```
Alpha AXP alignment fault exception control
```

```
Usage: axpalign [option]
```

```
Options:
```

```
/enable   to enable alignment fault exceptions.  
/disable  to disable alignment fault exceptions.  
/show     to display the current alignment exception setting.
```

```
Enable alignment fault exceptions:
```

```
In this mode any aligned access to unaligned data will result in a  
data misalignment exception and no automatic operating system fixups  
will occur. The application may be terminated or a debugger can be  
used to locate the source of alignment faults in your code.
```

```
This setting applies to all running processes and thus care should  
be taken since older applications may get exceptions and terminate.
```

```
Note that SetErrorMode(SEM_NOALIGNMENTFAULTEXCEPT) can be used to  
suppress alignment exceptions even in this mode.
```

```
Disable alignment fault exceptions:
```

```
This is the default mode with Windows NT for Alpha AXP, versions 3.1  
and 3.5. In this mode the operating system will fixup any misaligned  
data accesses should they occur and applications or debuggers will  
not see them. This may lead to performance degradation if it occurs  
at a high rate. Perfmon or wperf may be used to monitor the rate.
```

该实用程序只是修改注册表值的状态，或者显示值的当前状态。当用该实用程序修改数据值后，必须重新引导操作系统，使所做的修改生效。

如果不使用 AXPAlign 实用程序，仍然可以让系统为进程中的所有线程默默地纠正对未对齐数据的访问，方法是让进程的线程调用 SetErrorMode 函数：

```
UINT SetErrorMode(UINT fuErrorMode);
```

就我们的讨论来说，需要说明的标志是 SEM_NOALIGNMENTFAULTEXCEPT 标志。当该标志设定后，系统会将自动纠正对未对齐数据的访问。当该标志重新设置时，系统将不纠正对未对齐数据的访问，而是引发数据未对齐异常条件。注意，修改该标志将会影响拥有调用该函

数的线程的进程中包含的所有线程。换句话说，改变该标志不会影响其他进程中的任何线程。还要注意，进程的错误方式标志是由所有的子进程继承的。因此，在调用 CreateProcess函数之前，必须临时重置该标志（不过通常不必这样做）。

当然，无论在哪个 CPU平台上运行，都可以调用 SetErrorMode函数，传递 SEM_NOALIGNMENTFAULTEXCEPT标志。但是，结果并不总是相同。如果是 x86系统，该标志总是打开的，并且不能被关闭。如果是 Alpha系统，那么只有当 EnableAlignmentFault Exceptions 注册表值被设置为 1 时，才能关闭该标志。

可以使用 Windows 2000 的 MMC Performance Monitor 来查看每秒钟系统执行多少次数据对齐的调整修改。图 13-4 显示了在将该计数器添加给图表之前，Add Counter（添加计数器）对话框是个什么样子。

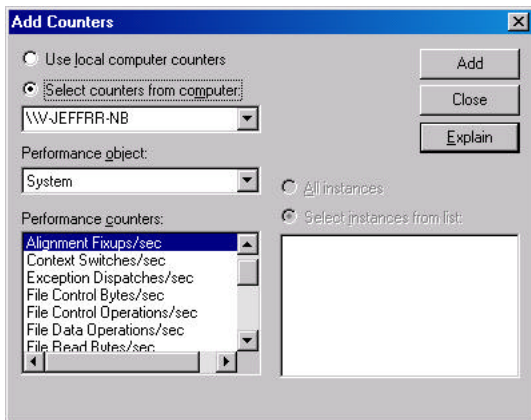


图13-4 Add Counters 对话框

该计数器显示的是每秒钟 CPU 通知操作系统的未对齐数据访问的次数。如果监视 x86 计算机上的这个计数器，你会看到它总是报告每秒钟为 0 次数据对齐的调整。这是因为 x86 CPU 本身正在进行调整，因此没有通知操作系统。由于是 x86 CPU 而不是操作系统来进行这种调整，因此访问 x86 计算机上的未对齐数据对性能产生的影响并不像需要用软件（Windows 2000 操作系统代码）来进行数据对齐调整的 CPU 那样大。

可以看到，只需要调用 SetErrorMode 函数便足以使你的应用程序能够正确运行。但是这个解决方案肯定不是效率最高的方案。实际上，Digital 出版社出版的《Alpha Architecture Reference Manual》手册上讲到，系统默默纠正未对齐数据访问的仿真代码运行时所花费的时间相当于普通情况下的 100 倍。这是个相当大的开销。不过有一种更加有效的解决方案可以解决你的问题。

Microsoft 用于 Alpha CPU 的 C/C++ 编译器支持一个特殊的关键字，称为 __unaligned。可以像使用 const 或 volatile 修改符那样使用 __unaligned 修改符，差别在于 __unaligned 修改符只有在用于指针变量时才起作用。当通过未对齐指针来访问数据时，编译器就会生成一个代码，该代码假设数据没有正确对齐，因此添加一些访问数据时必须使用的辅助 CPU 指令。下面显示的代码是前面已经讲过的代码的修改版。这个新版本利用了关键字 __unaligned。

```
VOID SomeFunc(PVOID pvDataBuffer) {  
  
    // The first byte in the buffer is some byte of information  
    char c = * (PBYTE) pvDataBuffer;
```

```
// Increment past the first byte in the buffer
pvDataBuffer = (PVOID)((PBYTE) pvDataBuffer + 1);

// Bytes 2-5 contain a double-word value
DWORD dw = * (__unaligned DWORD *) pvDataBuffer;

// The line above causes the compiler to generate additional
// instructions so that several aligned data accesses are performed
// to read the DWORD.
// Note that a data misalignment exception is not raised.
:
}

```

当我对Alpha计算机上运行的下面这行代码进行编译时，生成了7个CPU指令：

```
DWORD dw = * (__unaligned DWORD *) pvDataBuffer;
```

然而，如果我从这行代码中删除__unaligned关键字并进行编译，那么只生成3个CPU指令。可以看到，在Alpha CPU上使用__unaligned关键字，生成的CPU指令要多两倍以上。编译器添加的指令比CPU跟踪未对齐数据的访问并让操作系统来纠正未对齐数据的效率要高得多。实际上，如果监控Alignment Fixup/sec计数器，你将发现通过未对齐指针进行的访问对图表上显示的数据没有什么影响。

最后要说明的是，x86 CPU上运行的Visual C/C++编译器不支持__unaligned关键字。我想Microsoft公司也许认为这没有必要，因为CPU本身进行未对齐数据的纠正速度很快。但是，这也意味着x86编译器在遇到__unaligned关键字时就会产生错误。因此，如果打算为应用程序创建单个基本源代码，就必须使用UNALIGNED宏，而不是__unaligned关键字。在WinNT.h文件中，UNALIGNED宏定义为下面的形式：

```
#if defined(_M_MR000) || defined(_M_ALPHA) || defined(_M_IA64)
#define UNALIGNED __unaligned
#endif
#define UNALIGNED64 __unaligned
#else
#define UNALIGNED64
#endif
#define UNALIGNED
#define UNALIGNED64
#endif

```