Windows NT File System Internals
第一部分
第五章　NT 虚拟内存管理器

功能
进程地址空间
物理内存管理
虚拟地址支持
共享内存和内存映射文件支持
修改的页和映射页写入者
页错误处理
和文件系统驱动交互

今天的高级操作系统提供的一个重要功能就是管理节点上的物理内存。通常，机器上的非持久性随机访问存储器比其上的所有应用程序和操作系统需要的内存要少。因此，操作系统不得不干涉和影响这个有限的共享内存资源，节点上的所有组件在这个需求的地方常常争吵。另外，多个应用同时在一个机器上执行，操作系统有责任保证这些应用程序能够相互独立地执行他们自己的任务。因此，一个应用程序的代码和数据结构不能干涉其他的应用程序的代码和数据结构。操作系统还必须保护自己在内存中的用来管理系统的资源（代码和数据），不被执行在系统中的所有的应用程序破坏。这是机器自己的正确和安全运行的保证。最后，同一机器（有时候是网络连接的机器群）上的高级应用常常需要彼此共享内存中的数据。操作系统必须有秩序的共享数据，比如只有那些得到访问共享数据许可的应用程序才能访问数据等等。
虚拟内存管理器（VMM）有责任提供所有这些功能。VMM 叫这个名字是因为她为执行在系统上的每一个应用程序提供了一个抽象：每个执行任务的应用程序相信系统中的所有内存资源都是可以使用的。另外，应用程序相信自己有无限的内存资源可用。这个为特定的应用唯一的保留无限多可用内存的抽象叫做虚拟内存。VMM 是负责提供这个抽象的内核组件。

功能
NT VMM 为系统的其他组件提供下列的功能：

VMM 提供一个分页的虚拟内存系统。每个进程有一个相关的虚拟地址空间。这个虚拟地址空间由在需要的时候从机器的上的可用的总的物理页池中分配的物理页组成。
管理进程的虚拟地址空间和操作物理页分开。VMM 为应用进程控制虚拟地址空间的分配，约束，操作和释放的支持。
提供虚拟内存支持还得到本地文件系统的帮助。为了提供大量可用内存的假象（比实际 RAM 大），内存的内容被写入从辅助存储介质上分配的存储器中。内存在磁盘上的备份存储叫做“已提交内存”。“已提交内存”不是存放着能动态增长的页文件，就是辅助存储器上的数据/印象文件。
VMM 支持内存映射文件。这些文件可以任意大小；超过 2GB 的文件可以使用文件的部分“视图”。
支持在系统中不同的进程之间共享内存。这也用于进程间通信的一个方法。
VMM 实现每个进程定额
决定分配给进程的物理内存的工作集管理
另外，所有物理内存的分配/释放决定是由 NT VMM 执行的。不管这个内存是分配给用户空间的应用程序还是给内核模式的文件数据缓存。

使用访问控制列表（ACL）来保护内存。
支持POSIX的fork 和 exec操作，因此是符合POSIX标准的。
为页提供“写时复制”，他有能力建立保护页和设置页的等级保护。

进程地址空间
地址是一个指向一个包含在系统非持久性内存中的内存位置的值。每一个执行在 NT 系统上的进程可用的虚拟地址集。从执行进程的观点看，虚拟地址空间中的每个元素概念上指向每一个字节。WINDOWS NT 是 32 位系统，这就意味着所有指向内存的地址指针是 32 位值。这样的结果就是每个进程最多只能访问 4GB 虚拟地址空间的上限。
术语“虚拟地址空间”的原因很简单：虽然地址的范围从 0X00000000 到 0XFFFFFFFF，但是机器上的物理内存通常比 4GB 小很多。假设每一个虚拟地址空间中的单元都实际映射到物理内存上，那么系统将要给运行在系统上的每个进程安装 4GB 的内存。相反，VMM 欺骗每个进程相信他有 4GB 的内存可以使用。进程表面上接受这个建议，然后试图访问这个范围中的虚拟地址。VMM 要负责转换（或者映射）每一个虚拟地址到相应的物理地址上。
为什么进程需要访问内存？为了提供任何有用发功能，每个进程大多有下面的相关组件：

程序代码
进程栈（局部变量存储在这里）
已经初始化的全局数据
未初始化的全局数据
堆（动态分配的内存）
共享的内存
共享库

这些组件必须存储在物理内存的某个地方，虽然这些组件并不需要总是在物理内存中。如果这些组件在需要的时候被放置进了物理内存中，进程必须有某种方法来访问信息存储的地方。因此，每个进程必须有一个相关地址空间（或者一个地址的范围）。
有些物理内存必须被用于操作系统的代码和数据，因此现在，VMM 不仅要给进程提供进程相关的虚拟地址信息，还必须提供虚拟地址来引用操作系统组件，包括他自己用来管理系统中的内存的代码和数据。这是通过创建一个特别的进程来完成的，这个进程和其他的进程一样有 4GB 的虚拟内存可用。但是，对于他自己创建这个系统进程是不够的。
你知道系统中执行的进程常常要向操作系统请求系统服务。这些请求可能导致 I/O 操作，分配和操作内存，创建新的进程和其他类似的操作。NT 操作系统提供系统服务来接收用户的请求然后在内核中执行代码来处理这些请求。这就导致这样一种情况：执行在内核模式的代码必须在发起请求的某个用户进程的上下文中执行一些任务。
图 5-1 示例在 INTEL X86 硬件架构上的典型的进程地址空间
要执行这些任务，操作系统代码和数据必须能够从用户进程的上下文定位；也就是说，必须有一个地址范围的虚拟地址指向操作系统代码和数据，但是实际上位于由硬件设定的 4GB 的限制空间中。为了完成这个功能，NT VMM 把每个进程的 4GB 的地址范围分为两个部分，2GB 用于用户模式的虚拟地址叫做用户空间（执行在用户模式的进程只能访问这 2GB 范围），另外 2GB 范围包含内核模式的虚拟地址叫做内核空间。

注意：让我重复下面的概念：处理器可以在用户模式中或着内核模式中执行代码。因此用户模式或者内核模式状态是和处理器相关，而不是和执行在这个处理器上的

任何代码或者进程相关。所有内核开发人员必须理解这个区别。虽然微妙。

虽然 2GB 的用户模式虚拟地址指向进程私有的数据（不能被系统中其他进程访问），但是 2GB 的内核模式地址总是指向系统中相同的物理页（不管访问的是哪个线程上下文），包含操作系统的代码和数据。

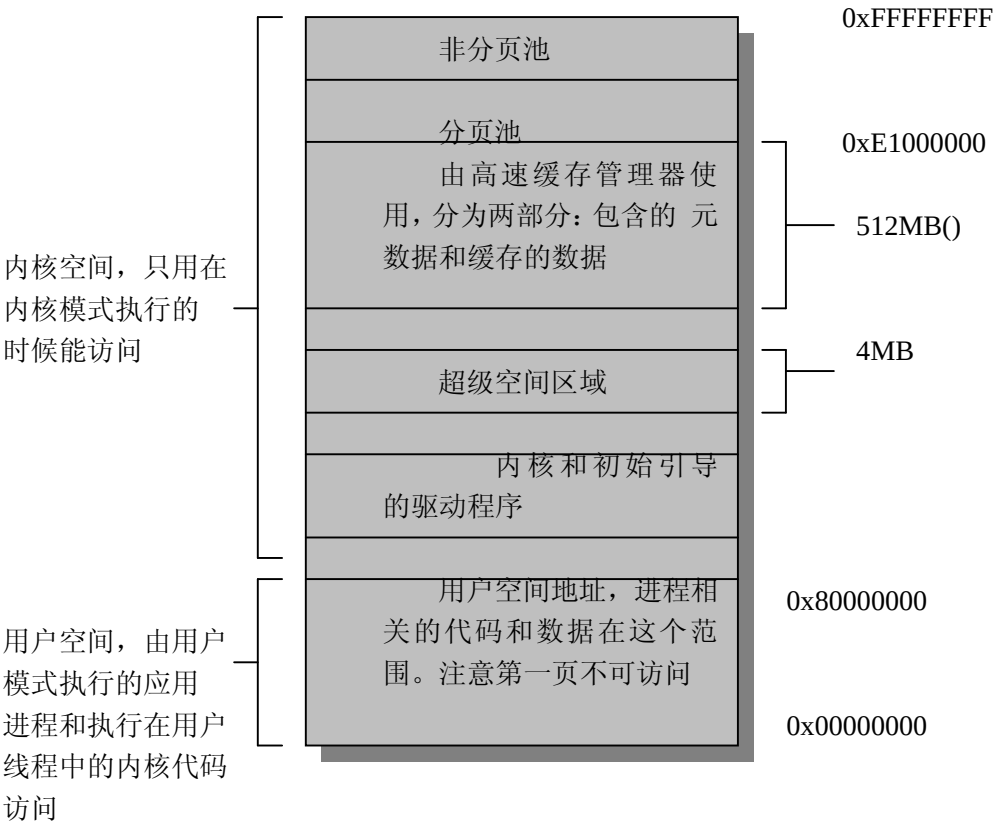

图 5-1 进程的虚拟地址空间

另一个应该了解的概念是和进程相关的 4GB 虚拟地址空间中的“超级空间区域”。这个“超级空间区域”是在内核 2GB 空间中保留的一个虚拟地址范围，特别要指出的是，因为这里面通常包含 NT VMM 维护的进程相关的内部数据结构。当一个上下文切换发生的时候，VMM 刷新这个区域来指出新进程相关的信息。这些数据结构包括分给进程的页表，以及其他的 VMM 数据结构。

如果你开发内核驱动程序，你必须始终关注代码运行时候的线程上下文（线程环境）。例如，如果你开发的是文件系统驱动，你的分配例程入口点通常在调用相应系统调用的用户进程的上下文中执行（但是情况并不总是这样，有时候子系统会在属于某个 WIN32 特定进程的工作者线程中调用文件系统的入口例程，另外，NT VMM 和 NT 高速缓存管理器常常也会在系统进程中的线程上下文中调用文件系统的读/写例程）。如果在这种情况下，你的驱动就可以使用底 2GB 的用户进程的地址空间来引用用户空间的内存（比如用户缓冲区）。但是，如果你写中间层或者底层驱动程序（比如设备驱动程序），你的分派例程通常在任意线程的上下文中被调用，在某个时候执行在处理器上的某个线程。这种情况下，你不能假设任何可能包含在 I/O 请求包中的用户空间虚拟地址还是有效的，因为你的代码不是执行在原来请求的用户线程的上下文中；所以底 2GB 虚拟地址空间现在映射的物理页已经属于其他的进程。

另一方面，如果你开发内核驱动，分配了内存，返回的内存指针通常指向一个内核空间的虚

拟地址。因为内核空间的虚拟地址对系统中的所有进程都是一样的，这个内存就可以在你的代码可能执行的任何线程上下文中引用（用分配的时候返回的指针）。
如果你的代码在任意的线程上下文中执行，你将怎样保证通过IRP传递的用户空间缓冲区可用呢？VMM正好提供支持例程来把用户空间的内存映射到内核虚拟地址空间（MmGetSystemAddressForMdl（））。
最后一点：你的内核驱动程序偶尔可能需要访问某个其他进程的虚拟地址空间。其中一条精通的方式是使用KeAttachProcess内核支持例程。这个例程没有在DDK中文档化，但是定义如下：

```
VOID
KeAttachProcess (
    IN PEPROCESS Process
) ;
```

参数：
Process
希望连接的进程的指针。可以使用IoGetCurrentProcess例程得到。
功能：
KeAttachProcess是你的内核驱动模式线程连接到一个目标进程。然后你的线程就在那个进程的上下文中执行，从而这个线程就能够访问属于那个进程的整个虚拟地址空间和所有的其他资源。
注意：你可能访问另一个进程的虚拟地址空间的原因是内存已经映射进目标进程而你需要访问。另一个原因是你需要使用任何属于目标进程的资源（比如，文件句柄）。
必须很慎重的使用，因为连接到其他的进程是非常昂贵的操作，在最坏的情况下，如果目标进程已经被换出物理页，将导致两次上下文切换。另外，在对称多处理器系统中还有可能引起“转换旁视缓冲区”（Translation Lookaside Buffers）的躁动，这将会降低系统性能。
不要在IRQL高于DISPATCH-LEVEL的时候调用这个函数。在这个函数的实现中在IRQL DISPATCH-LEVEL请求到一个执行自旋锁来保护内部数据结构；因此在一个更高的IRQL调用这个函数可能导致死锁。也不要在已经调用KeAttachProcess连接到一个进程的情况下试图连接到第二个进程，如果没有调用相应的解除连接就这样做的话回发生BUGCHECK。
从你的线程连接到的进程中脱离的相应的例程定义如下：

```
VOID
KeDetachProcess (
    VOID
);
```

参数：
无
功能：
这个函数使你的内核线程从一个先前连接到的进程中脱离。不要在 IRQL 高于 DISPATCH-LEVEL 的时候调用这个例程。

物理内存管理
要开发内核驱动（特别是文件系统驱动），深入理解内存管理器管理物理内存的方法是很有帮助的。一旦你理解了物理内存的操作，我将描述虚拟地址是怎样映射到物理地址上的。这些知识在调试 NT 系统和试图理解一些事情的工作方式的时候是很有价值的。
页帧和页帧数据库

NT VMM必须管理系统中的可用物理内存。VMM使用的是标准的基于页的规划，这种方法也是今天的现代商业操作系统如Solaris, HPUX, 或者 System V Revision 4 (SVR4)这些基于UNIX的实现所使用的。

NT VMM把可用的RAM分为固定大小的页帧。页帧的大小（页面尺寸）可以从4K到64K；在INTEL X86架构上现在是4K字节（注：WINDOWS NT和大多数商业操作系统现在都使用固定大小的页，但是已经有相当数量的关于在硬件架构和操作系统中实现可变大小页的研究在进行。支持可变大小页有一天可能在商业操作系统中实现，一个可能的推测是类UNIX平台会比NT平台更快实现。在NT4.0中，MICROSOFT在INTEL平台上使用4MB大小的页来存放内核代码。但是，现在的状态是，真正的可变尺寸页还没有被NT平台支持）。每个页帧使用一个叫做“页帧数据库（PFN数据库）”中的一个条目来表示。页帧数据库就是一个简单的在非分页系统内存中的条目的数组，每一个数组条目代表一个页帧的物理内存。对每一个页帧，要维护下面的信息：

PFN数据库中的条目代表的页帧的物理地址。这个物理地址现在限制在20位的域。当和12位的页面偏移量联合的时候，就形成32位量限制的支持4GB物理内存的系统。

一系列和页帧相联系的属性，他们是：

— 一个修改位指出这页的内容是否被修改过

— 状态，指出这个页帧在进行读操作还是写操作

— 页面相关的“页面颜色“（在某些平台上）

注意：在有一个物理索引的直接映射的缓冲的系统上，少数的为虚拟地址分配的页帧中的物理地址可能位于同一缓存线（例如，两个物理页被HASH到同一缓存线），如果这个页恰好是一个或者更多的正在执行的进程的工作集的一部分，将总导致缓存丢失。页面颜色试图用软件解决这个问题。但是NT VMM在X86的机器上不支持页面颜色，但是在另一些机器上面是支持的，比如MIPS R4000处理器。

— 表示这个页帧是一个进程的共享页还是私有页的信息。

一个反向的指针指向指向这一页的页表条目/原形页表条目（PTE/PPTE）。这个指针用于执行从物理地址到虚拟地址的映射。

一个页面的引用记数。这个记数向 VMM 指出是否有任何 PTE 引用页帧数据库中的这一页。

这个页帧可能被链接到的任何 HASH 链表的前向和后向指针。

一个事件指针在页面的 I/O 读操作正在进行的时候指向这个事件。

有效的页帧都有一个非 0 的引用记数。这些页帧包含了一页正被某个进程（或者操作系统）使用的信息。当页帧不再被一个 PTE 指向的时候，引用记数递减，当引用记数为 0 的时候，这个页帧就被认为是无用的。每个无用的页帧在反映页帧状态的五个不同列表中：

- 坏页帧列表，把有相同的错误（ECC）的页连接在一起的列表
- 自由列表，包含可以立即重新使用但是还没有被清 0 的页

  NT VMM 不会使用那些内容还没有清 0 的页帧（为了遵守 USDOD 定义的 C2 层安全定义）。但是，在对底消耗的关注中，页面在释放的时候没有被清 0。当一个关键段释放未清 0 的页面到达的时候，一个系统工作者线程会醒来异步的清 0 自由列表中的页。
- 已清 0 列表，连接那些可以立即重新使用的页帧
- 已修改列表，连接那些不再被引用但是还不能回收的页，直到页面的内容被写入辅助存储器为止。

  把已修改的页写入辅助存储器通常由“修改页面写入者/映射页面写入者”来异步的执行，将在后面讨论的一个组件。
- 等待列表，包含那些即将从进程的工作集中删除的页帧。

NT VMM 极力试图减少给一个进程分配的页帧的数量，基于进程的访问模式。这个分配给任何实例进程的页面的数量叫做进程的“工作集”。通过自动地无缝连接进程的工作集，NT VMM 试图更好地使用物理内存。但是，如果一个分配给进程页面在工作集清理中被清理出来，VMM 并不立即回收这个页面。相应的，把它放入等待列表中，VMM 延迟这个也面的重新使用，给予进程通过访问也面中的地址而重新得到这个页面的机会。当一个页帧在这个列表中，它就被标记为处于一个过渡状态，因为它还没有被释放，但是也不真正属于一个进程。

NT VMM 为系统中的等待和自由列表中的页帧的总量保持一个最小和最大值。每当一个页帧连接到自由或者等待列表，而且总的页帧量低于最小值或者对于最大值，一个适当的 VMM 全局事件被触发，这个事件是 VMM 用来查明系统中是否有足够数量的页面可以使用。经常，VMM调用一个内部例程来为某一个操作检查内存是否可用。例如，你的驱动可能调用一个叫MmAllocateNonCachedMemory的系统例程。这个例程需要自由的页面来分配给你的驱动，因此调用一个内部例程（对内核开发人员是不能直接使用的）叫做 MiEnsureAvailablePageOrWait的来检查自由页面列表或者等待页面列表中是否有请求的数量的页面可用。如果不可用，MiEnsureAvailablePageOrWait将会阻塞在这两个事件上等待足够的页面可用。如果在一个固定时间周期内这两个事件都没有信号，系统将调用恐怖的 KeBugCheck。

记住操作页帧数据库是一个经常性的操作。有相当多的研究在页帧数据库上使用更细密的锁来使VMM能够实现更多的并发运行。但是，NT VMM并不使用这些使用更细密的锁的模式。只有一个全局锁，一个执行自旋锁，用于整个页帧数据库。这个执行自旋锁在需要访问PFN数据库的时候在一个适当的IRQL（APC-LEVEL或者DISPATCH-LEVEL）请求得到。这可能减少并发，因为无论什么时候PFN数据库只能有一个线程可以访问。但是这简化了代码。记住在PFN锁得到的时候没有I/O执行（实际上是没有VMM模块以外的例程被调用）。但是，因为自旋锁是在 DISPATCH-LEVEL或者更底的IRQL等级上获得的，你现在完全可以相信你的代码在一个更高 IRQL上发生的任何页错误将导致系统崩溃。

虚拟地址支持

NT 虚拟内存管理器向系统的其他部分提供虚拟地址支持：

- 虚拟地址的范围可以独立于系统上的物理内存进行操作。
- 如果一个虚拟地址不管是由物理内存支持还是位于磁盘上的辅助存储器支持，NT VMM 帮助处理器硬件透明地把这个虚拟地址转换为相应的物理地址。
- 如果包含转换后的物理地址的页面需要从辅助存储器读入，NT VMM开始和管理这个 I/O操作。

  为了完成从磁盘到内存的数据转换，NT VMM使用适当的文件系统驱动程序的支持。
- VMM检查分页策略用来控制从磁盘到主内存的信息的转移来最大化系统的吞吐量。

在本章的前面提到，VMM为每个进程提供比系统中物理内存大很多的地址空间。但是，虚拟地址最终必须指向某个位于系统中物理RAM上的代码或者数据，VMM和系统硬件必须透明地把虚拟地址转换为物理地址。另外，因为执行在系统中的所有进程的总个内存需要量通常超过了可用的物理内存，VMM必须能够在需要是时候把数据和代码从辅助存储器移进移出。

NT VMM是一个核心组件，它决定了系统的性能和价值。RAM虽然每天都越来越便宜，但是还是不是不花钱的组件。同时，用户对他们的机器非常严格，一个VMM的糟糕的实现能够显著地降低整个系统的吞吐量。因此，VMM对于它强加于系统上的最小的内存需求是极度敏感的。这些情况是每一个设计必须决定的，某些权衡是不得不做的。在这一章的后面，

我会讨论NT VMM设计者作出的一个明显的权衡，在NT环境中实现分布式文件系统的时候导致的问题。

虚拟地址操作

为了给一个进程提供一个隔离的虚拟地址空间，NT VMM为每一个系统中的每一个进程维护一个虚拟地址描述符的（Virtual Address Descriptors (VADs)）自动平衡的二叉树。每一个分配给进程的内存块用这个树中的一个VAD结构来表示。这个树的根节点被插入进程结构中。一个虚拟地址描述符包含下面的信息：

- 这个VAD代表的虚拟地址范围的开始地址
- 这个VAD代表的虚拟地址范围的结束地址
- 一个指向树中其他 VAD 结构的指针
- 决定分配的虚拟地址范围的天然属性

  这些属性包含下列信息：

  — 分配的内存是否提交的信息

  对于已提交内存，VMM在已分配内存需要换出到磁盘的时候从页文件中分配存储空间来备份内存信息。

  — 指出分配的虚拟地址范围是进程私有的还是共享的信息

  — 描述和内存虚拟地址范围相关的保护特性的位

  这个保护位由原始保护属性联合组成：PAGE-NOACCESS，PAGE-READONLY，PAGE-READWRITE，PAGE-WRITECOPY，PAGE-EXECUTE，PAGE-EXECUTE-READ PAGE-EXECUTE-READWRITE，PAGE-EXECUTE-WRITECOPY，PAGE-GUARD，和PAGE-NOCACHE。

  — 虚拟地址范围中的页的“写时复制”是否打开

  “写时复制”这个特性能够有效地支持POSIX风格的fork操作，在这种操作中地址空间最初是被父进程和子进程共享的。但是，如果父进程或者是子进程试图修改一个页面的时候，一个私有的复制页会创建出来交给执行修改的进程。

  — 在fork发生的时候虚拟地址范围是否被子进程共享（VIEW-UNMAP=不共享，VIEW-SHARE=被父进程和子进程共享）

  这个信息只在一个文件的映射试图中有效，将在本章后面讨论。

  — VAD是否代表的是一个段对象（section object）的映射视图

  — 这个 VAD 相关的提交的内存数量

每当为进程分配了内存或者进程映射一个文件的视图到他的虚拟地址空间，NT VMM 分配一个 VAD 结构插入自动平衡的二叉树中。在分配内存的时候，进程可以指定他需要是提交内存还是仅仅是保留一个虚拟地址范围。分配提交内存的结果是和请求的数量的相应的内存被分配给进程。但是，保留一个虚拟地址范围仅仅是创建一个 VAD 结构然后插入自动平衡的二叉树中，然后虚拟地址范围的起始地址返回给请求的进程。注意在真正使用这些内存前必须提交。

NT VMM 允许进程分配和释放纯粹的虚拟地址空间，即是说，内存永远不需要提交。如果进程分配了一个虚拟地址范围然后发现他仅仅只需要提交这个范围中的一部分，NT VMM 也允许进程这样做。

有一个NT VMM提供的本地分配例程叫NtAllocateVirtualMemory，但是内核驱动开发人员不可用。内核模式驱动使用下面的例程代替：

```
NTSTATUS
ZwAllocateVirtualMemory(
```

```
    IN HANDLE ProcessHandle,
    IN OUT PVOID *BaseAddress,
    IN ULONG ZeroBits,
    IN OUT PULONG RegionSize,
    IN ULONG AllocationType,
    IN ULONG Protect
);
```

参数：

ProcessHandle

要被分配的内存所在的上下文所在的进程的句柄。对于内核驱动调用这个例程，是在系统进程的上下文中（例如，在驱动初始化的时候）。你可以使用宏NtCurrentProcess（只是简单地返回一个特定句柄值（-1）来标志当前进程是系统进程）。注意如果你要在不是当前进程的上下文中分配内存，NtAllocateVirtualMemory例程使用KeAttachProcess例程来把你的进程连接到目标进程然后分配虚拟地址空间。

BaseAddress

如果例程成功返回, BaseAddress 参数将包含分配内存的起始虚拟地址.如果你提供一个初始化的非 NULL 值,VMM 将把这个值舍入到页面尺寸的倍数后再试图在你提供的地址处为你分配内存.但是,如果你提供的是一个 NULL 初始化值,VMM 将会随便挑拣一个基地址.

注意如果VMM不能在你指定的基地址处为你分配内存(地址已经被使用或者从这个地址开始没有连续的可用内存),如果你已经指定了MEM_RESERVE作为AllocationType参数,一个错误将会返回(STATUS-CONFLICTING-ADDRESSES).如果你提供一个预先没有保留的地址将会返回同样的错误.

最后,你不能指定一个超过2GB的基地址,而且你指定的范围不能超过2GB虚拟地址限制.然后,重要的一点要注意,如果你使用这个调用你将得到一个内核模式地址,但是这个地址只有在传进来的(通过第一个进程句柄参数)进程的上下文里有效.因此,这个调用不是你的驱动得到内核内存的好方式(可以使用ExAllocatePool例程代替).

ZeroBits

这个参数只有在BaseAddress初始化为NULL的时候有效.你可以指定要分配内存的基地址的高位的0的个数.通过这样,你能够保证返回的的起始地址在一个特定值以下.这个参数不能大于21(因为那会使起始地址小于4096字节).0会被解释为2,这样返回的虚拟地址将总是在用户空间可寻址的2GB的虚拟地址空间.

RegionSize

这是一个指针参数.你必须指定要分配的字节数.你会收到实际分配的字节数,这个数字是把你提供的数字舍入页面尺寸的倍数后所得.

AllocationType

可以从MEM-COMMIT,MEM-RESERVE,MEM-TOP-DOWN中选择.第一个表示希望空间被保留在页文件中(内存已提交,是可用的).第二个是说你仅仅想要虚拟地址范围,可能在以后提交内存.因此这两个选项是相互排斥的.第三个选项可以和这两个中的联合,来说明你希望尽可能的从你用ZeroBits参数约束的起始地址分配.

Protect

这个参数可以是一个或者多个下面的原始保护选项:PAGE-NOACCESS,PAGE-READONLY,PAGE-READWRITE,PAGE-NOCACHE(不能放进数据缓存中,对已经映射的页面是不允许的),和 PAGE-EXECUTE.

功能提供:

这个函数只用于在低于 2GB 的进程虚拟地址空间中分配内存.因此,通常不被内核驱动使用,除非你非常确信你将只在指定进程的上下文中访问内存.如果你需要分配在任何进程的上下文中都能够访问的内存,就使用 ExAllocatePool 例程来代替.
这个例程允许你做下面三者之一的事情:

- 保留一段虚拟地址范围但是不提交
- 保留和提交一段虚拟地址范围(在一次调用中)
- 提交预先保留的一段虚拟地址范围

相应的用于释放分配的地址范围的例程定义如下:

```
NTSTATUS
NTAPI
ZwFreeVirtualMemory(
    IN HANDLE ProcessHandle,
    IN OUT PVOID *BaseAddress,
    IN OUT PULONG RegionSize,
    IN ULONG FreeType
);
```

参数

`ProcessHandle`

先前在其上下文中分配内存的进程的打开句柄.

`BaseAddress`

要释放的虚拟地址范围的第一个地址.这个值是舍入到页面尺寸的倍数的值.

`RegionSize`

这是一个指针参数.必须指定要释放的字节数.你会接收到实际释放的字节数,圆整到页面尺寸的倍数的值.

`FreeType`

可以是下列选择中的一个:MEM-DECOMMIT 或者 MEM-RELEASE.这两个是排斥的.

功能提供:
可以使用这个例程做下面的事情:

- 回收先前提交的页(但是保留分配的虚拟地址范围)
- 释放先前分配的虚拟地址范围和提交的内存

这个例程是相当灵活的,他可以修改先前分配的虚拟地址范围的子集.但是,不能期望能够释放分两次调用 `ZwAllocateVirtualMemory` 来分配的地址范围;也就是说,你指定整个地址范围必须包含在一个 VAD 中.如果你指定 `RegionSize` 值为 `0`,`VMM` 就解释为整个 `VAD` 必须被释放/回收.但是,在这种情况下你必须指定正确的 `BaseAddress`(等于 `VAD` 的 `BaseAddress`,或者等于你前面分配地址范围的时候指定的 `BaseAddress`).
这可能听起来奇怪但是有一种可能就是你可能得到一个错误来指出你超出了目标进程的定额,如果你试图释放一个先前分配的地址范围的子集.原因是 `VMM` 为了满足你释放原来分配的地址范围中的一部分的时候会分裂一个 `VAD` 为两个.当然,这会创建一个新的 `VAD` 结构,使他的定额等于目标进程的要求.如果把为这个进程分配的内存压缩到指定的值比允许的值大,你就会得到一个错误返回.

虚拟地址转换
在这一节中,我回为你讨论虚拟地址到物理地址的转换.这个主题在在很多文献中进行了深入

的讨论.我建议你查阅附录 E,推荐阅读和参考,来得到更多的信息.

每个 WINDOWS NT 中的虚拟地址都是一个 32 位量.这个虚拟地址必须被转换到指向内存中的某个物理字节(注:内存映射 I/O 设备注册器也能够通过虚拟地址空间寻址.因此,虚拟地址也能够被转换为实际上是 I/O 总线上的映射注册器相应的物理地址).两个系统组件共同工作来完成这个转换:

- 处理器提供的硬件的内存管理单元(MMU)
- 操作系统实现的虚拟内存管理软件

转换不仅仅在一个方向上是必要的,例如从虚拟内存地址到物理内存地址.VMM 还必须能够进行反向的转换,从物理地址转换到相应的虚拟内存地址(注:一个或多个虚拟地址可能指向同一个物理地址).当为了给其他数据腾空间而把物理页的内容写出到辅助存储器的时候,相应的虚拟地址必须被标记为"在内存中不再有效".这要求把物理地址被转换回他对应的虚拟地址.

虚拟地址转换通常在硬件中的 MMU 中执行.VMM 负责维护适当的"转换视图"或者"页表",以便 MMU 做实际转换的时候使用.大体上说,转换一个物理地址通常执行下面的操作序列:

1,在一个上下文切换过程导致一个进程开始执行的时候,VMM 构建适当的页表,其中包含特定于那个进程的虚拟地址到物理地址转换的信息.

2,当执行进程访问一个虚拟地址的时候,MMU 试图执行虚拟地址到物理地址的转换,使用一个叫做的"转换旁视缓冲区(TLB)"的缓存,或者,如果在 TLB 中没有发现条目,就使用 VMM 构建的页表,接下来,可能在系统中的页帧中的某一页中发现这个地址.

注意:从虚拟地址到物理地址的转换是消耗时间的操作.因为必须在每一次内存访问的时候执行这个操作,大多数的体系结构提供高效率的转换.提高这个操作的速度的一中方式是使用相关的缓存比如"转换旁视缓冲区(TLB)".TLB 包含最近执行的转换的列表,用进程 ID 做为标签.因此如果一个虚拟地址位于 TLB 中,相应的物理地址就能够立即获得,那个地址的内容也会保证处于主内存中.软件操作 TLB 是架构相关的;一些架构允许 VMM 显式地加载,卸载,刷新 TLB 条目(刷新其中的某个条目或者整个 TLB),但是其他的一些架构仅仅把加载或卸载 TLB 作为某些执行序列的副产品.

3,如果转换后的物理地址引用的字节在主内存中,进程就被允许访问数据

4,但是,如果这一页不包含在主内存中的页帧里,一个异常就发生了,一个页错误发生,控制交给 VMM 页错误处理器来把适当的数据放到主内存中.如果页面的保护属性和试图访问的模式冲突或者其他类似的原因硬件也可能发起一个异常.

注意 MMU 的设计对 VMM 子系统的设计有深远的影响.天生的, VMM 子系统和 MMU 接口的部分是非常依赖于特定的架构的,而且天生是不可移植的.

在前面描述的,VMM 在非分页池中维护一个页帧数据库来管理系统中可用的物理内存.这个数据库由页帧条目组成,每个页帧代表一段连续物理内存块.因为系统中的每一个物理页是一个有限的序列(有 N 个页帧的物理 RAM 是从页帧 0 到页帧 N-1),计算一个页帧在 PFN 数据库中的索引位置就是相当平常的.一旦一个虚拟地址被转换成物理地址(由页帧号和在页帧中的偏移量组成),用页帧号乘上 PFN 数据库项的尺寸,然后把结果地址和指派给 PFN 数据库的物理基地址相加.最终的结果是一个物理地址指向这个物理地址在 PFN 数据库中的页帧项的开始.

考虑 NT 平台上的 32 位虚拟地址值.因为页的尺寸是 4096 字节,计算一页中的偏移需要 12 位(最底(最不重要的)12 位).这就给 MMU 留下 20 位来唯一标志一个页帧.页帧是通过页表中的页表项(PTE)来唯一标志的,页表仅仅是 PTE 的数组.注意许多架构(包括 INTEL X86 架构)清楚地定义了 PTE 结构

在INTEL平台上,每个PTE必须是32位宽(4字节).一共可能有2的20次方(1M)个可能的PTE,而每个PTE有2的12次方(4K)个字节.存储一个4GB虚拟地址空间的转换信息需要2的22次方

(4MB)个字节.因为一个页面尺寸为4K,那么仅仅存储所有这些PTE就需要1024个页帧.这还是对一个进程来说.(注:记住INTEL X86架构是分段的,这里一个虚拟地址实际上由段地址和偏移量组成,INTEL硬件转换这个虚拟地址为32位的线性地址,然后这个线性地址被这一节描述的方法转换为一个物理地址,因此WINDOWS NT VMM提供一个平滑的内存模式给系统,隐藏段的细节,我们这里忽略虚拟地址到线性地址的转换处理,假设用户寻址虚拟地址仅仅需要一步就转换到物理地址)

为了避免消耗这么大量的内存来存储转换信息(注:记住很少需要4GB的虚拟地址空间完全被转换的,因此大多数地址空间是没有使用的实际上,也就是说,在寻址的虚拟地址空间中存在从来不使用的空洞,因此为那些可能从来不可能被利用的PTE保留内存是很不必要的),页表也可以被换进和换出内存.为了这样,X86处理器定义了两级页表策略.每个进程有一个包含页表的PTE的页目录.这个页目录是一页因此能够包含1024个PTE,每个PTE又引用了一个页表.一个典型的进程虚拟地址有10位用来标志进程相关的页目录中的页表,10位标志给定页表中的页帧,还有12位为页面中的偏移量.

图5-2图形显示NT系统上虚拟地址怎样转换为物理地址.记住在一些象MIPS R3000这样的系统上,架构本身没有PTE结构这样的限制(相应的没有提供硬件的虚拟地址转换支持,除了TLB查找外),VMM维护一集类似的数据结构来简化VMM子系统的设计和实现.

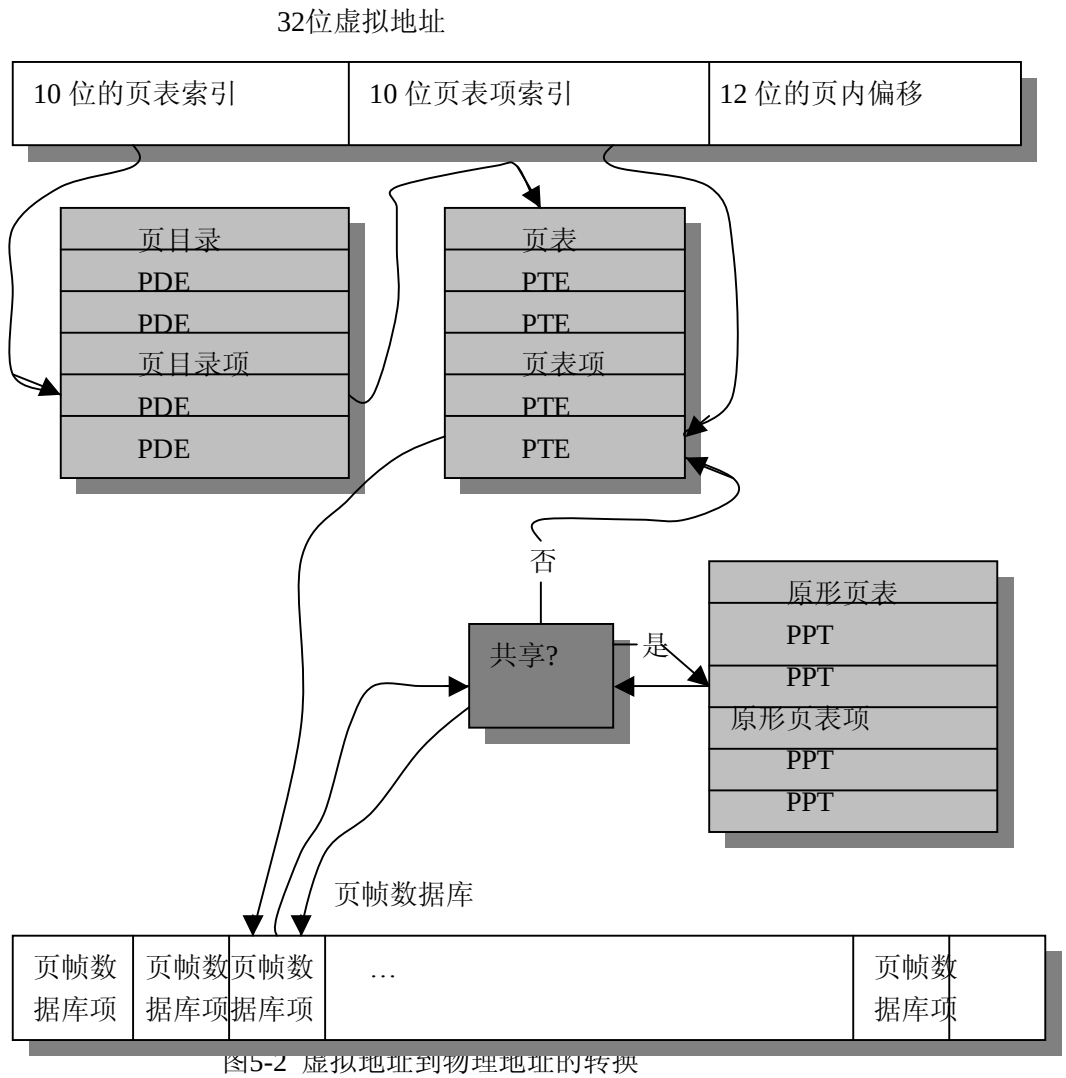

图5-2 虚拟地址到物理地址的转换

现在所有事情看起来相当清楚.MMU检查TLB,如果命中,他简单地返回已经转化的物理地址.另一方面,如果TLB没有命中,就必须检查进程的页表来定位相应的可能包含要访问的地址的物理页帧的PTE.现在,如果PTE指出页面在内存中,而且保护属性和访问模式匹配,MMU就允许访问继续.否则,一个适当的异常(页错误或者保护违例)发生,控制传给VMM.但是,细心的读者一定已经发现在图5-2中出现了一个另外的表叫做”原形页表(PrototypePage Table)”.那么PPT实际上在这个我们理解的清楚的模型中应该在什么位置呢?

PPT用来存放那些被一个以上的进程共享的页面的页帧的页表项.当超过一个进程在同一个映射对象上映射同一字节范围的时候就发生共享页面和页帧.因此,要理解PPT,你必须先理解共享内存和内存映射文件的感念.

共享内存和内存映射文件支持

今天应用开发人员看起来可以非常便捷地访问内存.应用进程可以简单地调用一个malloc (或者相等的操作),从VMM得到一个虚拟地址,然后开始使用这个虚拟地址来访问分配的内存块.操作系统和硬件一起负责管理物理内存和在虚拟地址和物理地址之间做适当的转换.另外,操作系统能够观察所有运行在系统中的进程的行为,为特定的进程的物理内存分配作出理性的决定.

同时,大多数应用必须进行其他的计算动作而需要内存.尤其是,所有的进程需要执行一些和辅助存储器之间的I/O.另外,高级的应用有时候还希望互相共享内存中的数据.

传统上,I/O是通过适当的文件系统来处理读/写系统调用完成的.这些系统调用要执行一个系统陷入来把处理器从用户模式切换到内核模式,缺点很多.对于一个读请求,文件系统必须首先把数据读进系统内存然后复制到应用程序分配的缓冲区中.对于写请求,操作系统首先从用户应用程序的缓冲中复制数据到系统内存.从系统缓冲中复制进出数据,和为I/O请求发起系统调用的负载一起,能够为应用进程造成固有的执行负载.

现在考虑系统在的两个进程访问同一个文件.这些进程可能会访问相同的字节范围,但是因为他们各自用自己的私有缓冲来存放数据,所以每个缓冲对应的物理页和其他的缓冲的物理页不一样,每个进程在相同的数据上有不同的视图.进程1可能把数据读进缓冲然后修改但是还没有写到磁盘上;如果进程2读相同的字节范围,他不会看见修改的数据而是从磁盘上得到原始数据.这种情况可以使用在这两个进程之间高效地共享数据来避免,因为每个进程必须保证他的修改在其他的进程读进这部分的字节范围之前写到磁盘上.

现在假设每个进程能够简单地把磁盘上的文件映射到他们的虚拟地址空间中.VMM通过在需要时候和在磁盘上的页文件交换数据来提供虚拟内存支持.进程分配一些内存,试图访问的时候可能产生一个页错误.页错误处理以后(这一章的后面将看到),神奇的,应用程序就能够访问为他保留的物理内存了.

现在假设数据最先从磁盘文件读进内存而且预定要写到磁盘文件中.这种情况下,为什么不使用文件本身作为数据的备份存储来代替页文件呢?当应用程序发出读/写系统调用来访问数据,系统简单地使应用程序保留一段和磁盘文件字节范围相关的虚拟地址范围,试图访问这些虚拟地址范围代表内存(实际上是访问虚拟地址相联系的字节范围),得到一个页错误,然后操作系统通过分配一些物理内存和从磁盘文件中得到适当的数据来处理这个页错误.类似的,应用进程可以简单地修改内存中的数据,每当需要的时候,操作系统会把修改的数据写回磁盘文件中,而且,可能会释放物理内存来为其他进程腾出空间来.

上面的文件映射的方法还有一个好处;所有试图映射到相同文件的进程可以有自己不同的虚拟地址但是保存在相同的物理页中,因此所有的进程将总是看见一个一致的数据的视图,不管实际上是任何进程在任何时候修改了数据.

NT VMM 支持文件映射。映射的对象是磁盘上的文件。当你执行一个文件（假设 WORD，这个执行文件（映射的对象）就映射到你的进程虚拟地址空间中然后指令就被执行。现在，

如果这个机器上的某个其他用户也想运行 WORD，这个同样的执行文件会被映射进他（她）的虚拟地址空间，因为对应 VAD 的物理页可能已经在内存中，其他的用户应该就能得到一个相当快的回应。图 5-3 是文件映射的图示

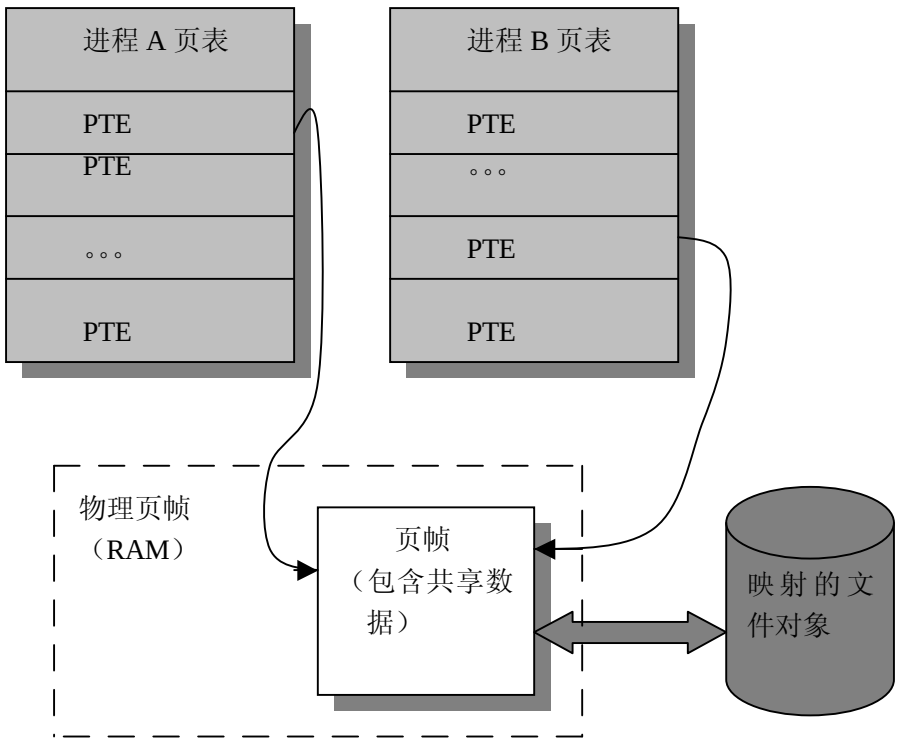

图 5-3 两个进程映射相同的页到自己的虚拟地址空

注意文件映射不是两个进程之间共享内存的唯一途径。因为 VAD 是用来把存放虚拟地址的物理页帧分开，对 VMM 来说完全可能简单地保证这两个进程的适当的 VAD 存放相同的物理页帧的方式来共享内存。文件映射只是这个概念的一个扩展，这里的共享的内存对象实际上是一个磁盘上的永久性文件，而不是页文件。既然你可以创建基于永久文件的共享内存对象，也可以创建基于系统页文件的共享内存对象。通常在你希望在系统中的两个进程或者模块之间共享内存的时候这样做。经常，内核驱动设计者需要在一些用户空间助手进程和内核驱动程序之间中共享内存。VMM 的共享内存支持允许这个功能。当一个共享内存对象（不是一个磁盘文件）创建的时候，和这个共享内存对象关联的起始虚拟地址表示在共享对象中偏移量 0 的地方。因此，所有共享这个对象的进程可以索引到适当的字节操作数据。但是，你必须注意修改共享内存对象后并没有写入到磁盘文件中，所以不是永久的；例如，这些修改在共享对象被使用这个对象的进程关闭的时候将丢失。
那么映射实际上是怎么做的呢？VMM 支持映射/共享对象会创建什么数据结构呢。在我回答这些问题之前，我们再次回到在图 5-2 中描述的 PPT。

原形页表（PrototypePage Table）
包含共享的（映射的）页面的页帧用一个特别的结构（PrototypePage Table）来描述。这个结构可以从分页的或者非分页的内存中分配。
当 VMM 为进程创建一个映射或者共享对象的时候，他创建原形页表项（PPTE）来描述要

存放文件映射的物理页帧。共享对象的 PPT 被所有的映射相同对象的进程共享。每个 PPTE 引用的页面可能出现在内存中也可能不在内存中；例如，页可能包含在一个物理页帧中，或者在访问的时候需要从辅助存储器中读入。因为所有进程使用同一个 PPT（和相应的 PPTE），因此所有进程使用相同的物理页帧，看见映射数据的相同的视图。

每当一个页帧被赋值给一个 PPTE，这个 PPTE 被标记为可用。PEN 数据库中的页帧项然后被初始化为指向这个 PPTE。注意不管是 INTEL X86 还是 MIPS R3000 或者类似的架构都不支持 PPT。VMM 是怎样处理这些事情使 MMU 能够处理共享内存呢？

考虑 INTEL X86 架构。INTEL X86 MMU 严格定义页表和 PTE 的结构。当一个进程创建一个文件映射的时候 VMM 在分配的内存中创建 PPT（还有 PPTE）。现在想象一下这个进程试图访问一个属于一个文件映射对象的范围的虚拟地址。MMU 将把这个虚拟地址转换成页目录表偏移然后根据这个偏移找到适当的页表。在第一次访问这个虚拟地址的时候，PTE 将指出需要的页不在物理内存中。

这就导致一个页错误产生然后控制传给 VMM 页错误处理器。页错误处理器注意到包含要访问虚拟地址的 VAD 被标记为内容存放于一个映射对象。VMM 然后发现适当的 PPTE 和错误的页。在这时候，这个 PPTE 标记为有效，指向一个 PFN 数据库项，相应的，一个 PFN 数据库项又指向这个 PPTE。同时，VMM 初始化 PTE 为有效并使他指向适当的物理地址。最后的结果是 PPTE 和 PTE 都包含物理地址的信息，但是相应的 PFN 数据库项只是指向 PPTE。现在，内存访问重新进行，而且，MMU 找到正确初始化的 PTE（他不知道也不关心 PPTE），虚拟地址到物理地址的转换也可以进行了。

PPT 设计的一个小问题

必须知道的是，因为 PFN 数据库项从不指向 PTE，VMM 从 PFN 数据库项就没有办法找到那些映射进所有进程的虚拟地址空间的 PTE。VMM 能做的最好的办法是找到 PPTE 引用的 PFN 数据库项，然后操作 PPTE 的内容。

这个设计有一系列的缺点：假设一个内核组件想要请求 VMM 从物理内存中清除某些页面（注：你可能会问为什么有人会想要这样做？假设你正实现一些复杂的多个节点的分布式数据访问，你的模块要保证保持所有的数据一致。现在，如果在远程机器上的一些进程修改了映射进本地节点的共享数据，你可能希望确保所有访问这些数据的节点用最后修改的数据来刷新他们的内存。这正好是分布式文件系统如 OSF DFS 所努力做到的。还可能有一些其他类似的情况如分布式架构上的某些复杂的功能可能需要这样的支持。），你绝对可以让 VMM 这样做，然后 VMM 将会把 PFN 数据库项标记为失效。另外，VMM 会使用存储在 PFN 数据库项中的信息来查找那些正引用这个 PFN 数据库项的进程的地址空间的适当的 PTE。然后标记这些 PTE 项失效来保证下一次访问包含在这个页面中的地址的时候，MMU 产生页错误然后重新载入页面。

但是，如果这个页面是属于一个共享对象，VMM 没有办法访问所有的包含共享页的页帧所使用的 PTE。因此，如果你请求 VMM 清除一个内存中的共享页面，VMM 会返回一个错误指出这个功能不能用于映射对象。对于任何第三方开发者在需要从内存中清除页面的时候会碰到的一个严重的问题。

段和视图

WINDOWS NT 系统是强烈倾向与对象为中心的；也就是说，大多数的功能是通过对象和操作对象的方法来提供的。文件映射的创建和访问有两步处理：

1， 请求文件映射或者共享内存对象的时候 VMM 会创建一个段对象。

2， 当进程实际需要访问一个映射文件或者共享内存对象的一个字节范围的时候，调用

者必须请求 VMM 映射一个视图到文件中。概念上，这个视图就象文件中的一个窗口，允许访问一个有限的字节范围。当然，一个进程可能也会同时请求一个文件的多个视图，就象多个进程可能同时拥有同一个映射文件的不同的视图也是可能的。

注意段对象有一些保护属性和他们相联系，就象 WINDOWS NT 环境中的所有其他对象一样。通过给段对象指定一些保护属性，进程可以定义这个对象（还有这个段对象代表的任何可能映射进来的文件对象的所有数据）被操作的方式。

磁盘文件保存的段对象分为两类：

- 可执行印象文件映射
- 文件（非印象的）映射

当你告诉 VMM 创建一个段对象来代表一个映射文件的时候，你可以指定这个映射的文件应该怎么解释。系统加载器使用文件映射来运行可执行文件和指定这个文件映射应该被解释为一个可执行印象文件映射。但是，你完全有权利请求把一个可执行文件当作一个非印象文件来映射。

注意 VMM 会执行测试来验证为可执行印象文件创建的段对象是否确实映射了一个有效的执行文件。如果你试图把一个简单的文本文件作为一个可执行印象文件来映射，你将从 VMM 得到一个错误。同一个可执行文件同时被映射为一个可执行印象文件和一般的文件也是完全可能的；虽然这两种映射文件的地址对齐可能很不相同。

在可执行印象文件映射和非印象文件映射（或者简单的共享内存）之间的主要的差别在于 VMM 怎样处理修改映射的范围内的内容。当一个非印象文件映射被一个进程修改的时候，这个修改立即被其他映射进这个文件的进程看见，因为共享物理页的内容被 VMM 改变了。当修改被刷新到辅助存储器的时候，最后会反映在磁盘映射对象上。但是，当一个印象文件的映射被修改的时候，会为进程产生一个私有复制页面来做这个修改。这个私有页面现在用一个页文件来保存，因为对一个印象文件的映射的修改从来不写到映射对象上（磁盘文件）。这些修改最后在进程关闭映射这个文件的时候被丢弃。

要创建一个共享内存对象（段对象），NT VMM提供了一个例程叫NtCreateSection。虽然这个例程没有导出给内核开发者，但是可以使用ZwCreateSection例程来代替，这个例程定义如下：

```
NTSTATUS
NTAPI
ZwCreateSection (
    OUT PHANDLE             SectionHandle,
    IN ACCESS_MASK          DesiredAccess,
    IN POBJECT_ATTRIBUTES   ObjectAttributes OPTIONAL,
    IN PLARGE_INTEGER       MaximumSize OPTIONAL,
    IN ULONG                SectionPageProtection,
    IN ULONG                AllocationAttributes,
    IN HANDLE               FileHandle OPTIONAL
);
```

参数：

SectionHandle

如果这个例程返回一个成功状态，创建的段对象的句柄就在这个参数中返回。注意这个句柄只在创建这个段对象的进程中有效。如果你想在其他进程的上下文中访问这个段对象的话，必须用ObReferenceObjectByHandle（在DDK中描述）这个对象管理器例程来取得一个实际的段对象的指针。接着，你可以使用ObOpenObjectByPointer例程来得到其他进程的上下文中的句柄。

DesiredAccess

这个参数指定这个段对象期望的访问模式：SECTION-MAP-EXECTUE，SECTION-MAP-READ，或者 SECTION-MAP-WRITE。

ObjectAttributes

这个参数可以是NULL，或者你可以指定一个初始化了的结构（可以使用InitializeObjectAttributes宏来做）。注意如果你需要在两个进程之间（或者在执行在用户模式的模块和内核驱动程序之间共享内存）共享一些内存，你可以用这个结构来为这个段对象指定一个名字。这个有名段对象随后就能够被其他进程打开，从而不使用一个辅助存储器上的命名文件就能够共享内存中的数据。

这个结构也可以用来为段对象指定一个安全描述符，来适当地保护你的段对象。

MaximumSize

对于一个基于页文件的段（就是说你只是想创建一个共享内存对象）这个值不能为NULL，因为他代表段的大小。

对于一个文件的映射，他表示这个段可能被扩展的最大的尺寸。如果这个段是一个已经映射的文件而且文件比这个值小，文件的大小在这时就扩展（注：文件系统开发者要注意这是非常重要的一点，因为你应该准备接受扩大文件尺寸的请求，当内存管理器创建文件映射的时候）。

注意你指定的任何值都会被舍入为本机页面尺寸的倍数。最后，如果为已经映射的文件指定这个值为 NULL，VMM 将把这个值设置为文件的末尾。

SectionPageProtection

这个参数定义段中包含的每个页面的保护属性，下面的适当的值：

- PAGE-READONLY
- PAGE-READWRITE
- PAGE-EXECUTE
- PAGE-WRITECOPY

AllocationAttributes

这个属性让调用者通知 VMM 如果段对象代表一个共享内存（保存在页文件中），可执行文件的文件映射，或者其他类型的文件的文件映射。下面是可以使用的选项：

- SEC-IMAGE，一个可执行文件将被映射进进程虚拟地址空间
- SEC-FILE，提供的指向打开一个文件的文件句柄必须被解释为非印象文件映射
- SEC-RESERVE，所有分配给段对象的页面必须被置为保留状态（只有在不是由磁盘文件保留的共享内存对象上才有效）
- SEC-COMMIT，所有分配给段对象的页面必须被置为提交状态（如果 SEC-FILE 已经设置或者共享内存对象不是一个可执行印象文件映射的时候必须设置）

FileHandle

这个可选参数指出这个段对象代表一个已经映射的文件(这个句柄必须指向一个打开的文件)。否则，VMM 简单地创建一个用页文件保存的段对象（简单的共享一些内存）。

功能提供：

这个例程可以被内核模式驱动用来创建共享内存对象（命名的或者匿名的）或者为磁盘文件创建文件映射。即使你是文件系统开发者正在实现一个磁盘的或者网络的文件系统，你可以使用这个调用来创建共享内存对象或者映射文件对象(不要使用这个调用试图在你自己的文件系统中创建映射的文件对象，除非你真的知道自己在做什么)。

有时，内核驱动开发者希望和用户空间的模块共享内存中的数据。或者你设计一个从网络得到数据的驱动或者使用一个用户进程来在网络上传输数据，你可以使用这个调用来建立一个

共享内存对象或者基于文件的共享内存对象来在用户进程和内核驱动之间进行高效的数据传输（考虑使用一个命名对象来让用户模式的服务容易地打开这个对象）。

提示：在描述 `ZwCreateSection` 例程的时候我提到存在一个对象管理器函数可以用来得到任意进程的上下文中对象的句柄，只要有这个对象的指针。这个例程叫做ObOpenObjectByPointer，定义如下（这个例程通常不在 DDK 中描述）：

```
NTSTATUS
ObOpenObj ectByPointer(
    IN PVOID                Object,
    IN ULONG                HandleAttributes,
    IN PACCESS_STATE        PassedAccessState OPTIONAL,
    IN ACCESS_MASK          DesiredAccess OPTIONAL,
    IN POBJECT_TYPE         ObjectType OPTIONAL,
    IN KPROCESSOR_MODE      AccessMode,
    OUT PHANDLE             Handle
);
```

通常，你可以为PassedAccessState和ObjectType传NULL。小心，只能请求在原来的打开操作（得到对象的指针的操作）中DesiredAccess参数允许的类型的访问。HandleAttributes可以从先前调用ObReferenceObjectByHandle的时候得到，这个函数返回Handlelnformation，其中包含了HandleAttributes。

还有一个叫做 `ObReferenceObjectByPointer` 的例程，这个例程仅仅递增指定对象的引用记数。这个例程定义在 WINDOWS NT IFS 包中：

```
NTSTATUS
ObReferenceObjectByPointer(
    IN PVOID Object,
    IN ACCESS_MASK DesiredAccess OPTIONAL,
    IN POBJECT_TYPE ObjectType OPTIONAL,
    IN KPROCESSOR_MODE AccessMode,
};
```

还有一些在 DDK 中描述的例程，用来打开和关闭先前创建的段对象，使用一个段对象来映射或者取消一个映射的视图。查阅下列系统支持例程的可用的文档：

- `ZwOpenSection()`
- `ZwMapViewOfSection()`
- `ZwUnmapViewOfSection()`

文件映射结构

当一个进程创建一个文件映射的时候，进程必须指定是一个可执行文件或者其他类型文件要被映射。虽然两种类型的文件映射最后都是文件内容被映射进进程的虚拟地址空间，但是 NT VMM 不同地解释映射请求。

在前面提到的，任何属于印象文件映射的页面的修改将不会影响映射的磁盘上的执行文件。页面将被用一个页文件代替，而且对这个页面所有的修改在映射关闭的时候被丢弃。

在内部，NT VMM 为每个映射的文件对象维护两种类型的段对象（和相关的数据结构）。对每个类型的映射，VMM 维护一个 SEGMENT 数据结构来描述映射。因此，每个映射文件可能有两个

段（segment）数据结构相关；印象段（segment）和数据段（segment），然后，每个段（segment）数据结构指向这个映射对象的 PPT。
虽然段数（segment）据结构对内核驱动开发者是不透明的，但是这里要注意的是两种类型的映射可以同时存在。一个可执行印象文件被映射为印象文件和非印象文件。对每种类型的映射，VMM将创建和维护一个段（segment）数据结构和内存中的文件相关联。因为有两种不同的数据结构创建了，依赖于执行什么类型的映射，那么一个文件包含在一个页中的相同的字节范围可能同时存在于内存中的不同的两个页帧中。这是可能的，因为每种类型的映射有自己的（segment）数据结构和自己的PPT和PPTE。

修改和映射页写者
前面讨论到，NT VMM有责任为每个进程提供大量的可用虚拟内存空间。即使系统上的物理内存是有限的。要完成这个任务，NT VMM必须使用辅助存储器来作为后备存储器用于内存中数据和页数据的换进和换出。分页调度对执行在系统中的进程是通明进行的。
NT VMM自动刷新脏（dirty）或者修改过的页面到辅助存储器来回收页帧给系统中的其他线程使用。一个页面中修改过的数据将被写到16个可能的页文件中的一个或者写入到磁盘上的有名字的文件（如果页帧是分配给一个已经映射的段对象）。除非修改页已经刷新到磁盘上，NT VMM不能重新使用页帧，因为这样会造成数据丢失。
为了保证在请求的时候有足够的RAM可用，NT VMM总是保持相当数量的页帧可用。这些页帧必须不能包含任何修改了的数据，因此他们可以重新使用。如果VMM不维护这个可用页帧的池，就可能使进程阻塞，等待把那些被修改了数据的页帧刷新的辅助存储器后再把页帧交给进程。使进程阻塞不利于系统的性能。
因此，NT VMM创建至少两个特别的奉献线程叫“已修改和映射页写者“线程。注意，很可能创建的线程的数量大于两个。至少一个修改页写者线程被创建开异步地把已修改的页帧写到页文件中。至少还有一个叫映射页写者的线程异步地写修改页帧到映射的文件中。这些线程本质上执行同样的功能，因此在本书中，术语已映射页写者线程和修改页写者线程是可以互换的。
这些线程的根本的目的是刷新修改页帧到辅助存储器中，以便保持相当数量的页帧可重复使用。这些线程都是适时线程，优先级至少为LOW-REALTIME-PRIORITY+1。
修改页写者使用的算法显示在下面。注意，下面的伪代码是基于映射的和修改页写者线程执行刷新页帧到内存映射的或者页文件中的操作；这两个线程的区别在需要的时候清晰地指出了：

```
// 下面的例程概述一个奉献工作者线程执行的MPW代码
// 但是，注意虽然各种版本的操作系统使用的方法可能不同
// 但是这里描述的基本逻辑是一致的
MiModifiedPageWriterWorker() {
    for (;;) {
        // 等待事件被设置指出自由的（未修改的）的页面存在不足
        // 这个事件在系统运行在底可用页面和VMM希望写出一些修改的页面以便能够被重
        // 新使用的时候被触发
        // 当修改过的页面总数超过一个系统预先定义的阀值
        // （阀值决定于系统被配置成工作站还是服务器和系统中有多少RAM）
        KeWaitForSingleObject(ModifiedPageWriterEvent, ...);
        // 现在，锁定PFN数据库
        ...
        for (;;) {
            // 事件指出现在有一些页面需要被刷新，取出一个要刷新的页帧（在
```

```
            // PFN数据库中修改页列表的第一个？)调用适当的例程.
            MiGatherMappedPages (PageFramelndex, ...) ;
            // 上面的例程负责实际的刷新.
            // 要执行刷新I/O,PFN数据库锁将被释放，而且重新被
            // MiGatherMappedPages() 例程获得. 因此，检查是否有足够的干净的页
            // 面可用，如果是就停止刷新.
            if (enough free pages are available) {
                // 解锁 PFN 数据库.
                break;
            }
        } // MPW线程刷新修改页到磁盘结束.
    } // 无限循环等待事件被设置以便MPW线程开始刷新数据结束.
} // MiModifiedPageWriterWorker() 例程结束。

// 下面的例程负责收集一串相邻的修改的页面然后写出到页文件中
// 类似的负责写映射文件页的例程叫MiGatherMappedPages().
MiGatherMappedPages (...) / MiGatherPageFilePages (...) {
    // 找一个存放页的分页文件
    if (分页文件不可用) {
        // 由于所有分页文件都正在进行I/O所以现在不能在任何事.
        return;
        // 使用每个分页文件的位图来寻找分页文件空间的一个连续的块.
        // 或者如果是一个映射文件，确保这个映射文件不是一个执行印象文件.
        // 初始化一个MDL(内存描述符列表)在I/O操作中使用
        // 从给出的页帧索引开始向前和向后扫描来寻找一个连续的修改的页
        // 来写出到页文件或者映射的文件.
        for (每个候选PTE) {
            if (由页文件支持或者映射文件支持的PTE 已经修改){
                // 增加 PTE 的引用记数
                PTE->ref erence_count++ ;
                // 标记这个 PTE 为 "未修改" 来预期我们的写出回成功.
                PTE->modified = FALSE;
                // 标记这个PTE正要被刷新，所有对这个PTE的刷新(就是说从文件系统或
                // 者从缓存管理器)将被阻塞直到这个I/O 完成.
                PTE->being_flushed = TRUE;
                // 把页文件的页地址放到PTE中.
                // 把这个页帧加入到MDL描述的物理页列表中.
            }
        }
        // 现在我们有了我们要刷新的页帧的列表
        if (页文件中保留的页面数量 > 发现的连续的修改过页帧的数量) {
            // 释放预先从页文件中分配的额外的空间(如果有的话).
        }
        // 解除 PFN 数据库 锁定.
```

```
        ...
        /*********************************************************/
        // 注意: 如果这个例程处理的是映射文件,就要执行下面这些附加的处理:
        {
            // 只是处理映射文件.
            if (这个文件标记为"fail all I/O"就返回) {
                return;
            }
            // 向文件系统发出一个回调，通知文件系统一个分页I/O在进行.
            // 这是非常重要的:
            // 文件系统必须回应这个回调 – 使用所有可能得到的资源来满足这个
            // 分页I/O操作.我们在这章的后面和第三部分会更详细地介绍这个回调函数.
            if (FsRtlAcquireFileForModWrite (...)) {
                // 回调成功, 发出I/O
                ...
                ZoAsynchronousPageWrite (...)
            } else {
                // 回调失败.返回错误代码 : STATUS_FILE_LOCK_CONFLICT;
                // 注意页面将保持“脏“状态，操作将在一段时间后重试
            }
        } // 映射文件的单独处理结束.
        /*****************************************************/
        // 注意: 下面的代码只为页文件支持页面执行
        {
            // 执行一个异步的，分页I/O操作.这个操作是一个特殊的请求，
            // I/O 管理器快速把请求发送给页文件所在的文件系统...
            loAsynchronousPageWrite (...) ;
            // 返回;
        } // 页文件的单独处理结束.
        /*****************************************************/
        // 锁定页帧数据库.
        ...
    } // 结束 MiGatherPageFilePages ( ) / MiGatherMappedPages ( )

// 下面的例程作为异步过程调用(APC)在文件系统完成异步分页I/O的时候调用.
// 注意文件系统可能选择同步地处理I/O，虽然不推荐这样 ...
MiWriteComplete (Context, StatusOfOperation, Reserved) {
    BOOLEAN FailAllIoWasSet = FALSE;
    // 内容实际是发送给文件系统的MDL
    MdlPointer = Context;
    // 锁定 PFN 数据库
    ...
    for (组成MDL的将被写出的每个页面) {
        // 设置写出处理失败
```

```
        PTE->being_flushed = FALSE;
        // 如果发现错误 ...
        if (error AND 写出到映射文件 AND 映射文件属于网络文件系统) {
            // 这对写重定向器文件系统的开发者很重要：VMM 假设
            //如果一个通过网络的分页I/O失败，那么网络一定出错了，在这种情况下
            // VMM 标记这页为"fail all I/O"，这页所有的修改数据被丢弃。
            FailAllIOWasSet = TRUE;
        }
        // Dereference the page.
        PTE->reference_count−;
        if (error AND 没有文件属于网络文件系统) {
            // 再次标记页面被修改，以后写出会重新试.
            PTE->modified = TRUE;
        }
    } // Loop for each page.
    // FOR MAPPED FILES ONLY ...
    ReleaseFileResources(); // 使用文件系统callback得到的资源
    // Unlock PFN database.
    ...
    if (FailAllIOWasSet) {
        // 用户看到著名的错误信息"system lost write-behind data".
        loRaiselnformationalHardError(STATUS_LOST_WRITE_BEHIND_DATA,
                                              FileName, Status);
    }
} // end of MiWriteComplete()
```

注意在这个伪代码中，VMM 使用一个 I/O 管理器函数 `IoAsynchronousPageWrite` 来刷新修改的数据到辅助存储器中。这个调用将会被 I/O 管理器迅速传递给管理目标页文件或者映射文件所在的挂载文件系统的文件系统驱动。

文件系统驱动可以轻易的认出这个写请求是一个分页I/O请求，因为发送给文件系统的I/O请求包有IRP_PAGING_IO 和 IRP_NOCACHE标志。注意，在文件系统处理分页I/O写出请求的时候不允许发生其他的页错误。I/O管理器区别地处理异步页面写出，当执行这个分页I/O请求描述的IRP结构的后处理的时候。本质上，I/O管理器调用内核APC例程MiWriteComplete来完成异步分页I/O IRP。这个例程在MPW线程上下文中被调用。

页错误处理

当一个虚拟地址指向的内容不在物理内存中的时候，NT VMM负责处理这种情况。虽然硬件MMU通常转换虚拟地址到物理地址，但是当MMU发现这个PTE指出的页面不在内存中的时候，MMU将把这个问题交给VMM来解决。不管在内核模式还是用户模式下发生的页错误，都会调用VMM的MmAccessFault例程，这个例程接受三个参数：

- 引起这个页错误的虚拟地址
- 一个BOOLEAN型参数指出是否是一个存储/写操作引起这个页错误（FALSE指出是一个读操作）
- 页错误发生时的模式（内核模式还是用户模式）

首先，MmAccessFault例程检查当前的IRQL，如果比APC-LEVEL高，如果页目录或者PTE指这个页面无效，VMM将会BUGCHECK系统，下面的信息将打印到你的调试妻屏幕上：

```
MM:***PAGE FAULT AT IRQL > 1 Va %x, IRQL %x
```

VMM中解决页错误的例程相应的叫做MiDispatchFault 。MmAccessFault例程调用MiDispatchFault来处理页错误，使页帧的内容有效。这个例程处理访问系统地址空间和用户地址空间的的页错误。页错误再基于发生页错误的地址被分派给适当的子例程进行更多的处理：

- 如果发生页错误的地址在一个页文件中，MiResolvePageFileFault例程被调用。
  这个例程执行下面的任务：
  — 在内存中分配足够的页帧，以便数据能够从页文件中读进来
  — 注意这个例程使用这一章前面提到的MiEnsureAvailablePageOrWait例程
  — 从PTE计算出要进行读操作的页文件
  — 创建一个MDL来包含可用物理页的列表
  — 标记将要被读进内存的页为“过渡中“（in transition）
  — 给调用者MiDispatchFault返回特定的状态0XC0033333
  MiResolvePageFileFault例程返回状态0XC0033333，MiDispatchFault然后会调用一个I/O管理器导出的IoPageRead例程来执行I/O读操作。就象在修改页写者讨论中描述的分页I/O写请求一样，I/O管理器调用文件系统来满足页读请求，文件系统根据I/O请求包中的IRP_PAGING_IO和IRP_NOCACHE标志认出这是一个分页读请求。注意文件系统在处理分页I/O读请求的时候不能发生任何页错误。
  VMM然后就等待页错误读请求完成，如果成功了，就把这个页面加入到活动进程的工作集中。
- 如果页错误地址的PTE指出这个页面正在“过渡中“（in transition），然后例程MiResolveTransitionFault就被调用。过渡中的页面被标记为正在过渡中是因为下面的原因：
  — 页帧包含有效数据，但是被放在自由列表中，因为是自动清理的
  — 页帧包含有效数据，但是在已修改列表中，这是自动从进程工作集中清除出来的
  — 这个页帧正在从辅助存储器读入；这是一个页错误冲突
  这个例程执行下面的任务：
  — 对于正在从辅助存储器读入的也帧，MiResolveTransitionFault将阻塞，等待I/O完成。如果有错误发生，他将把PTE标记为无效然后返回成功，强迫调用者发生另一次页错误，这样这个PTE就不用再标记为过渡中了
  — 否则，例程将标记PTE为有效然后把它加入当前进程的工作集中
  注意这个例程将不再返回状态0XC0033333，因为已经没有页面读操作要被MiDispatchFault初始化来执行了。
- 如果页错误的虚拟地址属于共享内存范围或者内存映射文件，MmAccessFault就用一个PPTE来调用MiDispatchFault。在这种情况下，MiDispatchFault调用MiResolveProtoPteFault子例程，来执行下面的任务：
  — 如果PPTE属于一个映射文件，MiResolveMappedFileFault例程被调用来决定要被读进内存的页面，分配一个MDL然后返回0XC0033333。注意VMM试图把页面串在一起来提高性能。
  — 如果PPTE是用于共享内存，包含在页文件中，MiResolvePageFileFault例程就被调用。这个例程决定要执行分页I/O读操作的页文件的数量，建立一个随后用来执行读操作的MDL结构，返回0XC0033333。
  — 如果PPTE指出正在过渡中，这个例程自己调用上面讨论的MiResolveTransitionFault。
  — 如果请求空白的页面，MiResolveDemandZeroFault子例程就被调用。

当一个适当的子例程被成功的调用后，MiResolveProtoPteFault例程将使PTE反映这个PPTE的内容。现在进程的PTE将指向PFN数据库项中那个将被读进内容的页帧（如果返回0XC0033333）或者已经有效的页帧，如果页错误转换解决了的话。

- 有时候，VMM只需要在页错误的回应中给出一个空白的页帧。这可能在一个线程试图扩大一个磁盘上的文件的时候，或者一个线程试图访问一些新分配的，提交的内存的时候发生。在这种情况下，MiDispatchFault简单地调用MiResolveDemandZeroFault例程，MiResolveDemandZeroFault接着从可用页帧列表中分配一个空白的页帧。如果没有页帧可用，MiResolveDemandZeroFault例程返回0XC7303001，这会导致页错误的重新发生，在那时候应该有一个物理页面可用（记住MPW线程总是努力保证有足够的自由的和未修改的页帧可用于分配）。

就象你看见的，VMM帮助MMU来解决在虚拟地址到物理地址转换后所要访问的页面不在系统内存中的情况。如果你开发驱动程序在IRQL高于或者等于DISPATCH-LEVEL的时候发生页错误，就会造成系统BUGCHECK，因为VMM在这样高的IRQL上不会提供页错误处理。确保在高IRQL的时候的所有代码和数据被预先锁定在非分页的系统内存中。

和文件系统驱动交互

NT虚拟内存管理器和文件系统驱动之间相互依赖。NT VMM依靠文件系统驱动提供页文件I/O段对象表示内存映射文件支持。然后，文件系统依靠NT VMM来解决发生在文件系统驱动中的页错误；操作用户缓冲和系统缓冲，分配，操作和释放内存，帮助缓存文件流数据（注：在第六章，NT缓存管理器，会更正式的定义文件流，现在你可以用文件来代替文件流）。

这里是NT平台上VMM提供给文件系统的功能：

- 文件系统驱动是一个可执行的，动态加载到系统虚拟地址空间的驱动程序，和VMM和文件系统驱动程序共同协助其系统地址空间中包含的执行体。默认的，文件系统和其他内核驱动的代码是不分页的；也就是说，这些驱动一旦加载就常驻与RAM中。类似的，所有和内核驱动相关联的全局内存默认也是从不分页的。有一个编译指令可以用来标记你的驱动的部分代码是可分页的。这个编译指示在所有NT兼容编译器上定义如下：

  #pragma alloc_text (PAGExxxx, NameOfRoutine)

  注意这里的xxxx应该是一个四字符的唯一的序列标志一个可分页的部分（也指可分页段）代码。另外，在运行时，你的驱动可以调用MmLockPageableDataSection或者MmLockPageableCodeSection例程来动态锁定代码或者数据。这些例程还有相应的解锁例程，都在DDK文档中。使驱动可分页的一些信息也在第二章中描述过。
- 文件系统驱动，过滤驱动，和设备驱动都需要在运行时分配内存。通常，你的驱动将调用一个版本的ExAllocatePoolWithTag例程来请求可分页的，非分页的或者缓存对齐（cache-aligned）的内存。你甚至可以给出请求内存失败就自动引起系统崩溃的条件。虽然执行体支持例程管理这些你的驱动取得内存的池，但是物理内存和物理内存的操作只由VMM来执行。任何使用ExAllocatePool中的一个例程返回的虚拟地址指针都保证是在内核虚拟地址空间中。

  注意的你的驱动也可以调用ZwAllocateVirtualMemory例程来直接向VMM请求内存，但是返回的虚拟地址将位于进程虚拟地址空间的底2GB中。
- 由于在执行在系统中的任意进程的上下文中你的驱动都必须是可访问的，VMM操作系统中每个进程的虚拟地址空间，底于2GB的进程私有的空间和高于2GB的系统地址空间，系统地址空间在系统中的所有进程上下文中的映射的物理地址是一样的。
- 作为一个文件系统或者内核模式驱动，你的代码常常要使用从用户模式代码传来的缓冲区（例如，一个执行在用户模式的线程分配了内存然后传给你的驱动）。你的驱动必须使用

这个缓冲区来传入或者传出数据。但是，有两个问题你的内核驱动必须要解决：

— 除非你的驱动总是保证执行在这个用户模式线程的上下文中，否则不能使用用户空间线程传近来的虚拟地址，因为这个虚拟地址只有在那个特定的线程上下文中在有效。

— 有时候，你的驱动可能需要在IRQL大于APC-LEVEL等级访问传入的缓冲区。这种情况下，你必须确保这个缓冲区在物理内存中锁定，因为这时候发生页错误将会导致系统崩溃。

VMM帮助你解决上面的问题。把缓冲区相关的物理内存页锁定在内存中可以调用VMM例程如：MmProbeAndLockPages，MmBuildMdl或者其他类似的例程。这会请求VMM创建一个MDL，一个用来描述存放你分配的虚拟地址范围的物理页帧的列表的不透明的结构。调用这个VMM例程，页面将被锁定在内存中；分配给缓冲区的页帧直到他们解锁的时候才会被回收。如果你需要把传进来的地址映射到系统虚拟地址空间，可以使用VMM例程MmGetSystemAddressForMdl。

提示：内存描述符列表（MDL）是一个系统定义的用来描述代表一个虚拟地址范围（缓冲区）的物理页面的结构。他包含一个数组，其中的每一项指向一个存放虚拟地址范围的物理页帧索引。数组在MDL结构后面接着分配；就是说，MDL结构和数组（他们都从非分页池中分配）在物理内存中连续的。

通常，你的驱动会经常请求VMM为用户缓冲创建MDL结构，然后常常把缓冲区映射进系统虚拟地址空间中。这样就确保页面保持锁定直到处理完他们，这样你就可以在任意进程的上下文中访问这个虚拟地址。

DDK中提供的ntddk.h头文件中包含MDL结构。注意你的驱动最好不要直接访问这个结构的域，因为他们可能会被系统改变。

- VMM管理为系统中执行的线程分配的栈帧。分配给一个执行在内核模式的线程的栈是一个固定的大小。NT 3.51以前版本，限制在两个页帧。WINDOWS NT4.0中扩大到3个4KB的RAM（12288字节）。
- VMM协助文件系统（和NT缓存管理器）缓存文件数据。所有物理内存的操作是集中在NT VMM。因此，使用物理内存来缓存字节流是非常需要VMM的支持的，最终能够提高系统的吞吐量。
- VMM在处理页错误的时候支持集群处理，这能够提高系统性能。
  通常，VMM试图集群处理16页的I/O操作。在Intel x86平台，这是64KB的I/O大小，但是在Alpha平台上，这个量变成了128KB的I/O操作。
- 有时候，过滤驱动需要做一些不寻常的事情，象缓存数据到一个本地文件系统的文件中。或者，用户模式代码和内核模式驱动之间可能需要传送数据缓冲区。要解决这些问题，内核驱动和用户模式应用程序可以使用VMM提供的共享内存对象或者内存映射文件服务。

提示：虽然本书的焦点不是设计和开发NT设备驱动，你应该意识到NT VMM实用例程和数据结构（MDL数据结构和操作它的例程）也应用于设备驱动开发者。有一些其他的VMM提供给设备开发者的支持例程，尤其是MmMapIoSpace例程，他映射一个物理地址范围到非分页系统空间中。到DDK中查阅这个例程的更多的信息和其他的HAL提供的例程。

但是请记住，不管你你开发什么样的内核驱动，你都需要理解这一章中的概念。

- NT VMM提供的支持例程MmQuerySystemSize，有时候对文件系统驱动很有用。
  MmQuerySystemSize例程没有参数，他只返回一个枚举类型的值，可以是下面三个之一：

—MmSmallSystem (enumerated type value = 0)

— MmMediumSystem (enumerated type value = 1)

— MmLargeSystem (enumerated type value = 2)

返回值决定于系统配置的物理内存数量。VMM初始化一个全局变量MmSystemSize，在系统初始化的时候根据机器上的可用物理内存设定为这三个值之一，MmQuerySystemSize返回这个全局变量的内容。

在不同的WINDOWS NT版本上，实际的RAM数量可能导致返回一个值而不是其他的易改变的值。例如，如果你的系统的物理内存低于12MB，你可能期望MmQuerySystemSize返回MmSmallSystem，类似的，如果你有低于20MB的可用内存，你可能期望返回MmMediumSystem。

这个函数调用通常被内核模式组件用来指导他们在资源分配中做决定。例如，假设调用这个函数返回MmSmallSystem的情况，现在你的文件驱动可能不知道SmallSystem的真实意思，但是你可以推测，至少可以说，可用物理内存的数量比MediumSystem或LargeSystem少。因此，和MediumSystem或LargeSystem的系统相比，你的驱动可以预先分配较小尺寸的池，或者创建比较少的工作者线程。用这个例程来得到系统的附加的信息帮助你的驱动决定资源利用策略。

毫无疑问还有其他的因素要考虑来最后作出你的驱动应该消耗的资源数量（物理内存）。

NT VMM也依赖于文件系统的下面的功能：

- 页文件是在已挂载文件系统上创建和操作的。因此要实现虚拟内存支持，VMM需要文件系统执行分页I/O的读写操作

  第三部分有一个说明的示例代码，当收到直接对页文件的I/O请求的时候文件系统必须完全信赖VMM。因此，文件系统应该避免获得任何资源（用来提供任何同步），在处理读/写请求的时候绝不应该发生页错误，绝不应该延迟这个请求进行异步处理，绝不应该因为任何原因阻塞这个请求。他只应该立即把请求（在决定了这个请求的磁盘参数之后）交给适当的底层设备驱动。
- 为了支持共享内存或内存映射文件，VMM需要文件系统的积极支持。首先，VMM要求文件系统提供适当的回调函数来帮助维护NT系统中的锁定分级。另外，VMM要求文件系统准备接受用户进程在指令序列访问映射的内存时候发生的页错误。

  文件系统必须实现的回调函数是AcquireFileForNTCreateSection和ReleaseFileForNtCreateSection。文件系统期望在NT VMM执行create section请求的代码的时候得到可能需要的所有资源。我会在第三部分讨论这些回调函数的实现。

为FSD实现提供的支持例程

VMM提供两个特别的例程，MmFlushImageSection和MmCanFileBeTruncated，这在文件系统设计中非常重要，但是没有好的文档。第三部有使用的例子。

MmFlushImageSection

文件系统驱动使用这个例程来要求VMM丢弃内存中的那些和一个特定印象段（image section）对象相关的页。例如，假设一个用户映射了一个WORD可执行文件的拷贝到内存中而现在希望删除它，可能是升级这个软件到新的版本。文件系统必须保证在实际删除这个文件之前，所有包含这个文件数据的页面被刷新（丢弃）。在通常执行中，这些页面可能在所有用户到这个文件的句柄都被关闭以后还保留着文件数据。但是，文件系统在计划删除这个文件流的时候不允许这些信息留在内存中。

注意：VMM强制限制一个文件在任何用户映射到这个文件流中的时候不能被删除；如果这

个文件当前正在执行，它不能被删除。但是，从这章的讨论中，你已经知道VMM会在内存中保留文件数据，即使所有的映射文件流的句柄都关了之后。只要物理内存不真的需要重新使用。这样在用户关闭了文件句柄以后但是很快又重新使用的时候能更快地回应。在执行删除之前文件系统必须刷新系统页面的这些情况正好。

在一个线程打开一个文件进行写访问之前，文件系统驱动也会调用这个函数。如果有一个线程把一个文件作为可执行印象映射到内存中，VMM通常不会允许一个用户打开进行写访问。

MmFlushlmageSection函数定义如下：

```
BOOLEAN
MmFlushImageSection (
    IN PSECTION_OBJECT_POINTERS    SectionObjectPointer,
    IN MMFLUSH TYPE                FlushType
);
```

这里

```
typedef enum _MMFLUSH_TYPE {
    MmFlushForDelete,
    MmFlushForWrite
} MMFLUSH_TYPE;
```

资源获得的约束：

文件系统必须保证在调用这个函数之前文件流被互斥地得到。通常，在调用MmFlushImageSection之前这个文件的主要资源都要互斥地得到。

参数：

SectionObjectPointer

在下一章中，SECTION_OBJECT_POINTERS结构将会消息地说明。现在，记住内存中的文件流都有都有一个关联的这个类型唯一的实例。VMM要求传递一个这种结构的指针给MmFlushImageSection这个函数。

FlushType

这只能是两个值中的一个：MmFlushForDelete或者MmFlushForWrite。在检查用户对一个盘上文件流的创建/打开请求是否允许继续的时候，文件系统应该传递MmFlushForWrite。在试图实际删除一个磁盘文件（在“set file information”分派例程和“cleanup” 分派例程中，都在本书第三部分描述）之前，文件系统应该传递MmFlushForDelete。

功能提供：

- 如果这个例程收到MmFlushForDelete参数，而有任何其他用户线程已经把这个文件流作为固定的数据流（内存映射文件）映射到他的虚拟地址空间中，VMM立即返回FALSE。
- 如果有任何线程把这个文件流作为执行印象映射到自己的虚拟地址空间，不管传递的是MmFlushForDelete或者MmFlushForWrite，VMM会拒绝请求返回FALSE。
- 否则VMM会抓取页帧数据库锁和标记印象段对象为删除。

当VMM决定了刷新印象段的操作是安全的，就会实际的遍历段里包含的“脏”页面然后刷新到辅助存储器中，如果他们是由盘上页文件保存的话。注意属于映射文件的任何“脏”页面（修改）会被立即丢弃。在实际开始刷新操作之前，VMM保证在这个文件流上的任何异步的修改页写操作会停止（实际上将会阻塞直到正在进行的写操作完成）。如果你的文件系统支持页文件，如果“脏”页面由页文件保存在你的文件系统上，你的文件系统这时候应该会收到递归的分页I/O请求。

一旦已修改页被刷新了，VMM把印象段对象交给文件系统，使得文件系统可以安全的处理删除或者打开操作。

调用这个函数之前有两个很重要的地方你必须知道：

- 当试图刷新印象段对象的已修改页到页文件中的时候，VMM将忽略发生的任何I/O错误。
- VMM会解除引用印象段对象被创建的时候引用的文件对象。

  对于文件系统设计者来说，这意味着你的驱动可能正在执行这个调用的时候收到一个关闭请求。如果你的驱动正在处理一个创建操作，不要惊讶会突然收到这个文件流的最后的关闭操作（作为调用这个函数的结果）（注：虽然你在实际设计你的文件系统的时候会更加赏识这一点，但是这可能导致你的文件系统一系列的问题，如果你不得不在放弃文件系统结构（因为收到了最后的关闭请求）和使用他们之前权衡的时候，因为你正在处理一个创建/打开请求）。

MmCanFileBeTruncated

VMM提供这个例程帮助文件系统决定是否允许对一个文件流进行截断操作。VMM在文件大小修改或者删除操作允许执行的时候强加某些限制。在文件流被作为执行文件映射的时候用户不允许截断这个文件流；当印象段对象由VMM创建并正被一个用户线程使用的时候截断请求会被拒绝。这样做的基本原理是这会使用户执行的文件极度混乱，如果一个页错误发生，因为这个截断请求后这页面相应的内容就不载存在磁盘上了，虽然刚才还存在。如果任何线程把文件流作为数据文件（非印象段）映射，而且如果新的文件大小比当前映射的文件流长度小，VMM也不会接受这个截断请求。

MmCanFileBeTruncated通常由文件系统在允许一个截断请求执行之前调用。使用这个函数的一个例子在第十章提供，这个函数定义如下：

```
BOOLEAN
MmCanFileBeTruncated (
    IN PSECTION_OBJECT_POINTERS     SectionPointer,
    IN PLARGE_INTEGER               NewFileSize
) ;
```

资源获得的约束：

文件系统必须保证在调用这个函数之前文件流被互斥地得到。通常，在调用MmCanFileBeTruncated之前这个文件的主要资源都要互斥地得到。

参数：

SectionPointer

在下一章中，SECTION_OBJECT_POINTERS结构将会消息地说明。现在，记住内存中的文件流都有都有一个关联的这个类型唯一的实例。

NewFileSize

一个大整数指针，包含建议的新的文件大小

功能提供：

- 在内部，MmCanFileBeTruncated调用MmFlushImageSection并指定MmFlushForWrite参数执行刷新请求。如果MmFlushImageSection函数返回FALSE，MmCanFileBeTruncated也会返回FALSE来拒绝截断请求。
- 另外，这个函数检查是否有任何的用户线程映射视图存在，如果存在而且新文件大小比映射的视图小，MmCanFileBeTruncated函数返回FALSE。

这里遵循的基本哲学观点是：

- 如果这个文件流的印象段在使用中VMM会返回FALSE
- 如果这个文件流的用户数据段存在，而且如果新的文件大小比当前映射视图小，VMM会返回FALSE。
- 如果上面的情况都不成立VMM会返回TRUE。

这一章讲述虚拟内存管理器，下面三章将讨论NT缓存管理器，这是一个协助文件系统驱动缓存数据的内核组件，这个组件高度依赖于NT VMM而且是VMM很明确的支持的。