Windows NT File System Internals
第二部分 管理器

第四章 NT I/O 管理器

- NT I/O 子系统
- 通用数据结构
- I/O 请求：一个讨论
- 系统启动顺序

顺利地和外部设备交互是任何计算机系统的要素。一个象 WINDOWS NT 这样的通用商业操作系统也要和各种外部设备交互，对我们来说最普通的一个是每天使用，还有一些不怎么常用的设备可能在某些特定的设定中会有用。例如，我们期望 NT 操作系统给我们提供内建的支持硬盘，键盘，鼠标，和视频监视器的功能，但是，如果我连接一个可编程的新设备（新发明的）到我的运行 WINDOWS NT 的系统上，我怀疑我不得不开发一个设备驱动来控制这个设备。另外，如果我希望开发这个驱动能成功，很显然我不得不期待操作系统提供一个适当的环境和支持结构使开发，安装和使用这个驱动的这个可能很困难但是不是不可能完成的任务。
虽然有人可能争辩说这样的期望对一个操作系统是不切实际的，但是 WINDOWS NT 操作系统确实提供了这样一个框架，以便象我们这样的人能够开发必要的驱动程序来控制这些深奥的设备如可编程外设。实际上，NT 操作系统提供一个兼容的，定义良好的子系统，所有请求和外部设备交互的代码都可有放在这个子系统里面。I/O 子系统是广泛的，包括文件系统驱动，中间层驱动，设备驱动和服务，支持和和这些驱动交互。它在处理外部设备上也是兼容的。
在这一章里，我给出一个 NT I/O 管理器的介绍，NT I/O 管理器是负责创建，维护和管理 I/O 子系统的组件。要为 NT 操作系统开发任何类型的驱动程序，理解 I/O 管理器提供的框架是极其重要的。首先我会描述一些 I/O 管理器提供的服务。然后会给出组成 I/O 子系统的组件的概览，包括讨论各种能够存在与 I/O 子系统的中的各种类型的驱动。接着要描述一些内核驱动开发这应该熟悉的通用数据结构。然后讨论一些通用的问题包括向一个内核驱动发 I/O 请求。最后描述系统启动的顺序，重点是 I/O 子系统和内核中的驱动的行为。

NT I/O 子系统

NT I/O 子系统是一个框架，在它里面所有的内核驱动程序控制和接口和外部设备。它由下面的组件组成：

NT I/O 管理器，它定义和管理整个框架

文件系统驱动负责本地的，基于磁盘的文件系统

网络重定向器，访问 I/O 请求，通过网络发出请求。实现和其他的文件系统驱动类似。

网络文件服务器，接收其他节点上的重定向器发给他的访问请求，再把这些请求发送给本地文件系统。虽然文件服务器不需要实现为内核模式驱动程序，通常是考虑性能的原因。

中间层驱动，例如 SCSI 类驱动程序。这些驱动程序给对一个集合的设备提供通用的功能。中间层驱动还包括提供附加功能的驱动，比如软件镜像或者错误冗余等使用设备驱动服务的驱动。

直接和硬件接口的设备驱动，例如控制卡，网络接口卡，磁盘驱动。这些通常属于最低

层的内核驱动。

过滤驱动，把自己插入驱动层中来完成使用现有的驱动不能直接完成的功能。例如一个过滤驱动可以把自己放在文件系统上，截获所有发给文件系统驱动的请求，过滤驱动还可以把自己置于文件系统下面，设备驱动的上面，截获所有发送给设备驱动的请求。注意在概念上，在过滤驱动和中间层驱动之间唯一实际的区别是过滤驱动通常拦截到某个存在的设备的请求然后提供自己的功能，提供请求原来接收者的额外的功能或者场所。

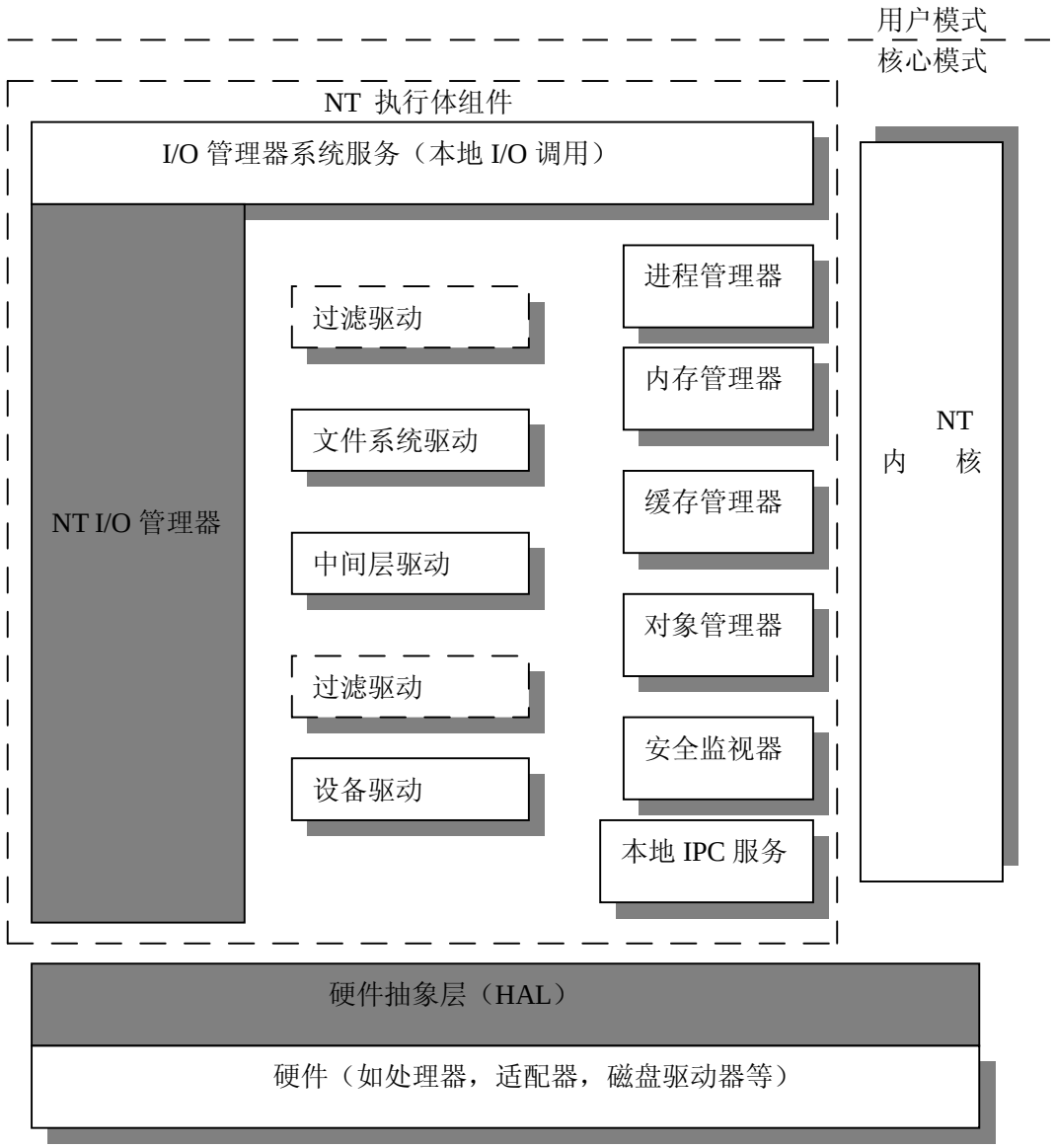


图 4-1 内核模式组件，包括 I/O 子系统

NT I/O 管理器提供的功能

NT I/O 管理器监视着 NT I/O 子系统。下面是 NT I/O 管理器提供的一些功能的列表：

NT I/O 管理器定义和支持一个框架来使操作系统能够使用连接到系统的外部设备。WINDOWS NT 可以使用的外部设备的类型和数量是没有限制的，因为外部设备是不断在被设计可开发。因此 I/O 子系统对于一个商业操作系统如 NT 这样的必须是定义良好的而且可扩展的，这样就能够轻易地适应千变万化的设备，每个都有自己唯一的特征的设备。

NT I/O 管理器提供全面的系统服务让其他的不同子系统使用来执行实际的 I/O 或者向内核模式驱动请求其他的服务。

考虑一个用户进程初始化一个读请求。这个请求指向控制子系统如WIN32子系统。注意WIN32子系统并不直接把读请求发送给文件系统驱动或设备驱动，而是调用一个由I/O管理器提供的系统服务调用叫NtReadFile。NtReadFile系统服务负责把请求发到适当的驱动然后把结果输送给WIN32子系统。还要注意由用户进程提供的用于读操作的缓冲通常不能直接被最后处理读请求的内核驱动使用。I/O管理器提供自动执行必要的操作从而使内核驱动能够使用在内核中可访问的缓冲地址的支持。在这一章的后面，会更详细的描述操纵用户模式缓冲的操作。

虽然本地NT系统服务文档非常少（几乎没有），你可以在本书的附录A中找到那些服务的详细的描述。

I/O 管理器定义了一个系统中执行的驱动都必须服从的单一 I/O 模式。上面提到过的，这个模式由对象和用来操纵对象的一组方法组成。内核驱动不需要关心 I/O 请求发起者，因为他们以同样的方式回应所有的 I/O 请求。

这样的结果就是给 I/O 子系统提供一个兼容的接口，例如 WIN32 或者 POSIX 子系统，这也保护了子系统不必关心发出的 I/O 请求的变化无常的各种子系统。

另外，因为每一个内核驱动必须服从单一 I/O 模式，内核驱动可以使用彼此提供的服务，因为内核驱动并不关心 I/O 请求来自内核模式还是用户模式。但是如果你要从你的驱动调用其他的驱动提供的服务，有一些事项你必须知道，这些在这一章的后面描述。

最后，单一 I/O 模式能够实现 I/O 管理器支持的层次化的内核驱动。每个驱动层中的内核驱动可以利用下面的驱动提供的服务来完成特定的操作。接下来，下面的驱动可以完成接到的请求而不管请求来自用户进程或者驱动层中的位于上面的驱动。

I/O 管理器支持用连接到系统的外部设备实现可安装文件系统

NT 操作系统支持包括 CD-ROM 文件系统，基于记录的 NTFS 文件系统，遗留的 FAT 文件系统，LAN 管理器文件系统重定向器，还有 HPFS 文件系统。除此之外还支持比如本地和基于网络的文件系统，I/O 管理器提供开发外部可安装文件系统的基础，即是说，第三方作者实现的文件系统。你可以购买商业实现的 NFS（网络文件系统），DFS（分布式文件系统），和其他文件系统和网络重定向器的实现。

NT I/O 支持动态可加载内核驱动程序

I/O 管理器能被 NT 操作系统的其他组件使用的设备无关的服务，也可以被第三方作者实现的内核驱动利用。

如果一个内核驱动需要调用另一个内核驱动程序的分配例程，它可以使用I/O管理器提供的IoCallDriver服务。类似的，如果一个内核驱动要分配一个内存描述符列表（MDL）结构，可以使用IoAllocateMdl例程。还有其他的这样的由I/O管理器提供的内核组件（包括内核驱动）可以使用的通用的服务。可用的服务列表在DDK中。

NT I/O管理器和NT缓存管理器相互作用提供文件数据虚拟块缓存。

在本书的后面，你会学到更多NT缓存管理器提供的功能。

NT I/O管理器和NT虚拟内存管理器相互作用和文件系统实现来提供内存映射文件。

在下一章，你会读到内存映射文件的细节。提供内存映射文件是NT I/O管理器，NT虚拟内存管理器和适当文件系统驱动共同提供的。

如果你希望为NT开发内核驱动，你必须服从NT I/O管理器提供的规范。这包括创建和维护一些I/O管理器定义的数据结构，还要提供操作这些对象的方法。另外，你的驱动必须适当地回应NT I/O管理器发出的请求，你的驱动还必须返回每个操作的结果给NT I/O管理器。不使用任何NT I/O

管理器提供的服务几乎是不可能成功的开发内核驱动程序的。这一章的后面更详细的解释这些问题。

I/O管理器设计中的概念

NT I/O子系统的设计显示出一系列的特征在下面的节中描述。

基于包（Packet-based）的I/O

I/O子系统是基于包的；例如，所有的I/O请求是用I/O请求包（IRP）来提交的。IRP通常由I/O管理器在用户请求的时候构建然后传递给目标内核驱动程序。但是，任何内核组件可以使用 IoAllocatelrp () 创建一个IRP然后使用 IoCallDriver ()发送给一个内核驱动。

I/O请求包是你能够用来向I/O子系统驱动请求服务的唯一方法。通过严格的服从这个基于包的I/O模式，I/O管理器确保I/O子系统的兼容和使层次化的驱动模式建立起来。

每个发送给内核驱动的IRP表示那个驱动程序的一个未决的I/O请求。一个IRP将持续存在直到IRP接收者为这个特定的IRP调用IoCompleteRequest服务例程为止。调用IoCompleteRequest的结果是I/O操作被标记为完成，I/O管理器然后触发任何那些等待这个I/O请求完成的完成后的处理。一个特定的IRP只能被完成一次，也就是说，只有一个内核驱动能够为任何系统中存在的IRP调用IoCompleteRequest。

你应该知道的，虽然基于包的I/O是WINDOWS NT的规则，NT I/O管理器，NT缓存管理器和各种各样的文件系统实现合作实现叫做“快速I/O路径（fast I/O path）”的功能是这个规则的一个例外。I/O操作的快速I/O方法只对文件系统驱动有效。NT缓存管理器不在使用普通的IRP方法，而是使用直接的文件系统中的函数调用来实现这些操作。快速I/O路径在本书的后面会详细描述。

NT 对象模式

NT I/O管理器是遵循NT执行体的对象管理器组件的NT对象模型来定义和实现的。

内核驱动程序，外部设备，控制器，适配器，，中断和打开文件是实例在内存中都是用能够被操作的对象来表示的。和这些对象关联的还有一套方法，一套可以在对象上执行的操作。例如，每个控制器在系统中用控制器对象表示，而每个打开文件的实例用文件对象数据结构表示。控制器对象只能使用和这个对象相关的方法来访问。相同的限制也应用于文件对象结构，还有所有其他由I/O管理器定义的对象类型。

记住为NT开发的内核驱动程序必须和I/O子系统的其他部分一样遵循这个基于对象的模式。所有的驱动程序必须初始化一个驱动程序对象结构来表示加载的驱动程序自己。另外，如果驱动程序管理设备或者连接到系统的外部设备还必须创建和初始化一个或者多个设备对象结构。

因为I/O管理器使用NT对象模型，它可以使用安全子系统的服务来控制对象的访问。I/O管理器支持命名对象结构。例如，文件对象有一个相关的名字来代表在磁盘上的文件。你也可以创建其他命名对象比如设备对象，然后就能够被其他进程或者内核驱动打开。

分层的驱动程序

I/O管理器支持分层的内核驱动程序，每一个层中的驱动程序接受一个I/O请求包，处理，然后调用层中的下一个驱动程序。

层中越下层的驱动越接近真实的硬件。但是，通常只有最底层的驱动直接和硬件设备或者接口卡交互。分层的驱动模式对希望提供那些基本操作系统不提供的有附加值的功能的的设计者是一个福音。这个特性使中间层驱动和过滤驱动能够在需要的时候插入到驱动层中，因此新的功能就能轻易地加入系统中。另外，层次中的每个驱动和上面和下面的驱动使用一种兼容的方式交互，开发，调试和维护内核驱动和其他操作系统实行相比就容易很多。

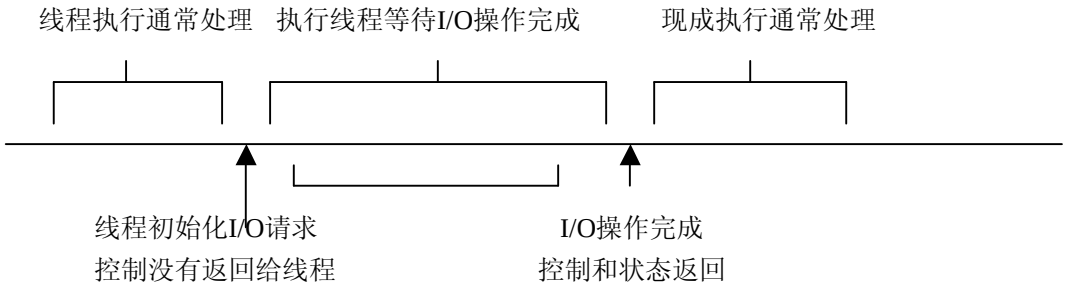
异步I/O

NT I/O管理器支持异步I/O允许线程请求一个I/O操作然后继续执行其他的计算任务直到先前请

求的I/O操作被完成。这样就使计算任务有了更大的并发性，对于纯粹的线性模式来说，在那种模式中的一个线程必须在进行其他活动之前等待I/O操作完成。

图4-2图形显示当执行同步和异步I/O操作的时候的动作序列。正如你从图中看到的，异步I/O能够继续并行执行计算动作和已经初始化的I/O请求服务一起。这导致系统更高的执行性能和更高的网络吞吐量。注意默认的I/O机制是同步模式。

同步I/O操作



异步I/O操作

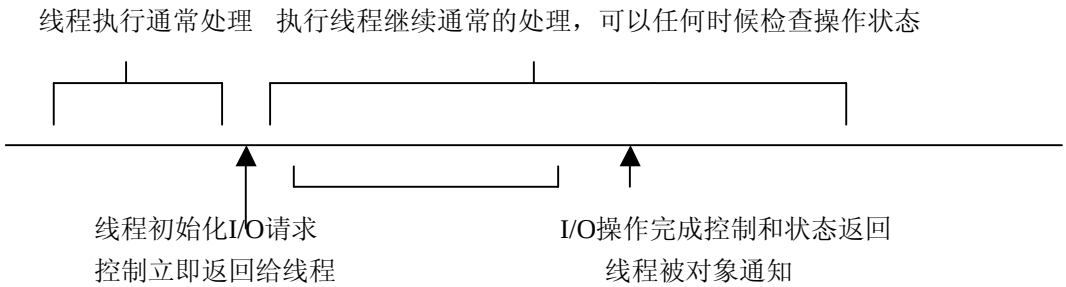


图4-2 同步/异步处理

抢占和中断

I/O子系统是可抢占和可中断的。理解这两个概念对内核启动开发者来说是极端重要的。

每个执行在内核模式的线程都执行在某个系统定义的中断请求等级（IRQL）。每一个IRQL有一个中断向量和他相联系，WINDOWS NT一共定义有32个不同的IRQL。任何线程的执行可以中断执行取决于一个比当前正在执行的线程更高的IRQL。当这样一个中断发生的时候，和这个特定的中断相关的中断服务例程（ISR）会在当前执行线程的上下文中执行。这导致暂停当前的执行流以便线程来执行ISR代码（ISR的执行也能够被更高IRQL的中断所中断）。

IRQL范围从PASSIVE-LEVEL（定义为数字0）（默认的所有用户线程和系统工作者线程执行的等级）到IRQL HIGH-LEVEL（定义为数字31）（系统中最高的硬件IRQL等级）。大多数文件系统分配例程执行在PASSIVE-LEVEL。但是大多数低层设备驱动例程（例如，SCSI类驱动的读/写分派入口点）执行在更高IRQL等级—通常在IRQL DISPATCH-LEVEL（定义为数字2）。

因为I/O子系统中的所有代码是可中断的，NT操作系统的驱动程序开发者必须使用适当的同步和保护机制来防止在不同IRQL等级访问数据的时候造成数据丢失。例如，如果你的内核驱动在一个系统工作者线程上下文中以PASSIVE-LEVEL等级访问一个数据结构，如果这个驱动程序还可能在处理中断请求的时候在DISPATCH-LEVEL等级访问这个相同的数据结构，这个驱动程序不得不使用一个自旋锁来提供互斥的访问这个数据结构（自旋锁总是在DISPATCH-LEVEL等级得到，这是可能得到自旋锁的最高IRQL等级）。

线程执行内核中的I/O子系统代码的时候也是可以抢占的。WINDOWS NT操作系统把执行优先级和线程关联。这些优先级通常是可变的，大多数用户层线程和系统工作者执行在相对比较底的优先级，这就允许他们在一个更高优先级的线程被调度运行的被NT调度代码（在NT内核中）抢占。线程可能被抢占的事实使得在执行内核模式代码的时候提供同步机制来保证数据一致性也很必要。这个需求在其他操作系统中不需要，例如WINDOWS 3.1操作环境，或者一些版本的UNIX（例如HPUX，SUNOS），这些操作系统中的线程或者进程在内核模式中执行的时候不允许抢占。内核驱动设计者在从不同线程上下文得到公共资源的时候必须非常小心，因为NT内核不提供任何内建的对程序错误的安全保护，例如在第一章中描述的优先级导致的情况。

如果你开发一个驱动需要在IRQL小于或者等于DISPATCH-LEVEL等级得到多于一个同步资源，你必须小心地定义严格的锁定层次。例如，假设你的驱动不得不锁定两个FAST-MUTEX对象，fast_mutex_l和fast_mutex_2。你必须定义驱动中所有能得到这个两个互斥对象的线程的得到顺序。这个顺序可能是“得到fast_mutex_2之后得到fast_mutex_1”或者相反的顺序。严格定义和维护锁定层次的原因是避免一种情形，假设线程A得到fast_mutex_1，想要得到fast_mutex_2，但是被抢占。线程B在其时得到调度执行，得到fast_mutex_2，现在又需要得到fast_mutex_1。这个情况将导致死锁发生。

移植性和硬件无关

I/O子系统是可移植和硬件无关的。NT环境内核模式驱动也要求是可移植和硬件无关的。

NT硬件抽象层（HAL）负责提供下层处理器和总线特征的抽象给操作系统的其余部分。NT驱动程序必须小心的使用适当的HAL，NT执行体，和I/O管理器支持例程来保证在Alpha, MIPS, PowerPC,和Intel平台上的可移植性。

NT I/O子系统的绝大部分使用C这种高级的可移植的语言写成。NT现在也要求内核驱动开发者使用C语言来写代码，虽然通过一些额外的工作来用汇编写和链接驱动程序是可能的。但是，用低级语言如汇编语言是很令人沮丧的，因为汇编语言天生是处理器/架构特定的，因此这种驱动不能在一种以上的处理器架构上执行（注：有第三方提供的库声称能帮助你用C++开发NT驱动程序）。

多处理器安全

I/O子系统是多处理器安全的。WINDOWS NT的设计是能够在对称多处理器环境上执行的。执行在多处理器机器上执行NT内核代码和驱动程序要求内核设计者细心地同步来避免数据一致性问题。例如，在单处理器机器上，用来在处理一个中断的时候避免数据一致性问题的一个常用的习惯是禁止同一机器上的所有其他中断（例如，在X86架构上通过一个汇编指令完成）。但是，同样的机制在对称多处理器系统上将失败，因为可能在其他的处理器上会发生中断，即使所有在中断在当前处理器上都被禁止了。类似的，在但处理器系统中，可以保证在某一时刻只用一个线程能够访问一个特定的数据结构（例如使用一个关键代码段）。但是在对称多处理器架构上，即使一个处理器上的一个线程的抢占权临时被挂起，但是在其他处理器上执行的线程也能够毫无疑问地同时访问这个相同的数据结构。

通常，必须一贯地和正确地在NT驱动中使用自旋锁和其他高层（执行）同步机制来保证在对称多处理器系统上的正确性。

模块化

NT I/O子系统是模块化的，任何NT I/O子系统中的驱动可以被另一个提供和原来的驱动提供一样的分派例程的驱动代替。使用I/O请求包来提交I/O请求和基于对象的模型的通过标准方法（或者定义良好的分派例程入口点）调用所有I/O操作，这些特性允许容易地用一个有相同分派例程的驱动来代替一个驱动。

所有驱动使用一个定义良好的一致的服务来调用I/O管理器提供的服务和有效功能，因此，理论

上I/O管理器也是能够轻易地代替的。但是，实际上I/O管理器是核心NT操作系统的一个极其复杂的和完整的组件，将是极其难以被轻易地代替的，即使是MICROSOFT自己的开发者。
但是，模块化的I/O子系统的一个明显的好处是I/O管理器支持的函数和驱动功能能够比较容易地重新实现而不会影响任何使用I/O管理器和驱动提供的服务的客户。只要接口是保持一致的，任何内部实现就可以在需要的时候作出改变。
可配置的
I/O子系统的所有组件是可配置的。I/O管理器和所有组成I/O子系统的组件努力最大化运行时的配置性。NT I/O管理器和HAL在系统启动的时候确定连接到系统的外部设备。它然后初始化适当的数据结构来支持这些连接的设备。这些处理任何要求硬编码设备配置到操作系统中。WINDOWS NT还不真正的支持即插即用，虽然在不久的将来应该会。
内核驱动可以被设计来操作设备；每个驱动程序可以动态地加载和卸载，最小化不不要的内核负载。I/O管理器根据注册表中的项来决定驱动被加载，被加载的顺序。I/O管理器的配置参数，也是那些内核驱动要求的，也从NT注册表中得到。
任何你开发的驱动应该是尽量可以配置的。这包括避免在驱动代码中硬编码值而是从系统注册表中得到这些值，最大化用户配置性。

进程和线程上下文
在讨论I/O管理器和I/O子系统的其他细节之前，了解线程/进程上下文的概念和认识到为什么深入掌握这些概念是理解WINDOWS NT内核中的各种组件的操作的本质是很有用的。要成功的在NT平台上设计和开发内核驱动，你需要牢固地掌握这些概念。
每个WINDOWS NT操作环境中的进程用一个进程对象结构代替，有一个进程上下文在进程中是唯一的。进程的执行上下文包括进程的虚拟地址空间（在下一章描述），进程的可见的资源，一些属于这个进程的线程。属于进程的资源包括例如这个进程打开文件的文件句柄，这个进程创建的任何同步对象，任何其他的这个进程创建的或者代表这个进程的进程创建的对象。每个进程至少创建有一个线程属于这个进程，虽然进程的确可以有许多的线程属于他。记住在WINDOWS NT中基本的调度单位是线程而不是进程。
在内部，WINDOWS NT内核使用一个进程环境块（PEB）结构来描述一个进程，这个结构对操作系统的其他部分是不透明的。PEB包含进程全局上下文，例如启动参数，印象基地址，进程范围的同步对象，加载器数据结构。在创建的时候，进程分配一个访问令牌叫进程的主要令牌。默认的，这个令牌被和这个进程关联的线程访问任何WINDOWS NT对象的时候拿来验证他们自己。每个新进程对象结构要创建一个对象表。这个对象可以是空的，也可以是父进程的对象表的一个复制品。这依赖于提供给系统的创建进程例程的参数和父进程的对象表中的每个对象的继承属性（OBJ_INHERIT）。新进程默认的访问令牌和基本优先级和父进程一样。
线程对象是实际执行程序代码的实体，也是被WINDOWS NT内核调度执行的实体。每个线程对象都和一个进程对象关联；几个线程对象可以和一个进程对象关联，这允许在一个进程中同时执行多个线程。在单处理器系统上，线程不能并发执行。但是，在多处理器上，并发执行是可能的也是这样发生的。
每个线程对象有一个线程上下文是唯一的，这个上下文是依赖于架构的，通常有下面的组成：

- 截然不同的用户栈和内核栈，用一个用户栈指针和内核栈指针来标志
- 程序计数器
- 处理器状态
- 整形和浮点寄存器
- 架构相关的寄存器

你将注意到，存储在进程的对象表中的对象句柄和其他相关打开的对象结构对所有和进程关

联的线程是全局的。因此，进程中的所有线程可以访问进程的打开的所有的句柄，即使是进程中的其他线程打开的。属于其他进程的线程只能访问那些他附属的进程的对象；任何试图访问属于其他进程的资源都将导致WINDOWS NT对象管理器组件返回一个错误（注：通常，如果你写一个内核驱动，试图使用一个在当前执行进程上下文中无效的句柄的话，你会看见返回一个错误状态给你：STATUS_INVALID_HANDLE）。

线程通常涉及到是用户模式还是内核模式的线程。注意在内部表示上这些线程没有差别，直到WINDOWS NT操作系统是相关的。唯一的概念概念上差别是当这些线程执行代码时候处理器的模式和线程能够访问的虚拟地址的范围。例如，一个WIN32应用进程包含的线程在处理器在用户模式的时候执行代码因此属于用户模式线程。另一方面，有一个属于特定系统进程上下文全局的线程池由WINDOWS NT执行体创建，用来在处理器在内核模式的时候执行操作系统或者驱动程序代码；这些线程通常属于内核模式线程。

虽然用户模式线程通常在处理器在用户模式的时候执行代码，但是他们常常请求系统服务，例如文件输入/输出，这会使处理器执行一个“陷阱”然后进入内核模式去执行文件系统代码来处理这个I/O请求。注意用户模式的线程现在处理器在内核模式执行操作系统（文件系统驱动程序）代码，当处理器在这种状态的时候有所有的权利和特权。当执行在内核模式的时候，线程内核虚拟地址和执行那些处理器在用户模式的时候总是被拒绝的操作。

执行上下文

考虑一个你开发的内核驱动。内核模式驱动的事实告诉我们：当代码被执行的时候，处理器将在内核模式，因此能够访问内核虚拟地址范围。你可能好奇那些线程会执行你开发的代码。会是你将要创建的一些特定的线程，或者会是一个请求你驱动的服务的用户模式线程，或者会是我前面提到的系统工作者线程池中取出的一个线程呢？

答案是，都有可能。你的驱动可能总是在一个特定你可能在驱动初始化的时候创建的线程中执行。或者可能在一个请求I/O服务的用户线程的上下文中执行，或者可能在系统工作者线程环境中被调用。这有太多的可能，如果你开发文件系统驱动，你的驱动将在所有这三种类型的线程的上下文中执行代码。另外，如果你开发设备驱动程序或者其他低层驱动，有被调用来处理中断的分配例程，你的代码将执行在中断发生的时候在处理器上执行的哪个特定的实例的线程的上下文中。这就是说在任意线程上下文中执行代码，也就是说，你的驱动不知道是哪个线程的上下文。操作系统临时的借用这个线程的执行上下文来执行你的驱动例程仅仅是因为在中断发生的时候这个线程恰好在处理器上执行代码。

因此，作为一个内核驱动开发者，你必须总意识到你的代码将要执行的上下文。执行上下文总是下面中的一个：

请求系统服务的一个用户模式线程的上下文

如果你开发文件系统或者在文件系统上面的过滤驱动，那么你的代码常常执行在请求的用户线程的上下文中，比如读操作。不的 代码能够访问内核虚拟地址范围，还有属于请求的用户模式线程的低于2GB的虚拟地址范围。

通常，只有文件系统或者截获文件系统请求的过滤驱动可以指望他们的分配例程将直接执行在用户模式线程的上下文中。其他的驱动不能指望这样，仅仅因为高层驱动可能使用户请求在一个工作者线程上下文中来异步地执行，或者是你的代码可能在中断的时候执行。

你的驱动或者其他内核组件（通常属于I/O子系统）创建的工作者线程的上下文

文件系统驱动有时在系统进程中创建特别的线程（使用PsCreateSystemThread）随后用来执行那些不能执行在用户模式线程上下文的I/O请求服务操作。过滤驱动也可能选择创建这样的工作者线程；或者因为这个原因，任何内核模式组件可以选择创建一个或者多个工作者线程。

如果你写文件系统驱动，你可能偶尔请求你创建的那些线程处理一些操作。你的代码将执行在你的特定的线程的上下文中，但是，如果你写低层的驱动，如果文件系统使用一个特殊线程处理I/O请求，你的驱动现在就可能在文件系统驱动创建的特殊的线程的上下文中被调用。不管那种方式，你都会看见代码执行在属于系统进程的一个特别创建的线程的上下文中。

I/O管理器特别创建来为I/O子系统服务的系统工作者线程的上下文

某些I/O操作执行在I/O管理器特别创建的系统工作者线程的上下文中是可能的。这些工作者线程常常被文件系统驱动实现使用，或者被设备驱动或者其他需要线程上下文执行他们的操作的内核组件使用。例如，假设从用户应用程序来的异步I/O请求。通常，文件系统把种请求“邮寄”给一个系统工作者线程来完成。一旦请求被邮寄，控制立即返回给调用应用程序，当请求被完成的时候，I/O管理器会在系统工作者线程上下文中通知应用程序。在这样一种情况下，所有低层驱动程序将使他们的分派例程在系统工作者线程上下文中被调用。注意系统工作者线程属于系统进程，就象前面讲的内核组件创建的奉献线程一样。

这里要注意的重要的一点是一旦请求被邮寄给系统工作者线程，现在在系统工作者线程上下文中可访问的虚拟地址空间就和原来请求I/O操作的用户模式线程上下文中可访问的虚拟地址空间不一样了。类似的，原来在用户模式线程上下文中有效的资源在系统工作者线程上下文中不再有效了。原因很明显：系统工作者线程执行在系统进程上下文中，请求I/O操作的用户模式线程属于一个截然不同的应用进程，有自己的对象表，虚拟地址空间和进程环境块。

某些任意线程上下文

现在考虑一个能够在任意时间点处理IRP的设备驱动程序。通常，大多数设备驱动把I/O请求放入队列延迟处理，然后立即把控制返回给他的上层驱动。IRP将在以后驱动把队列中位于他前面的IRP处理完的时候处理。

那么IRP是怎样从队列中取出来的呢？一旦当前I/O操作被目标设备完成的时候，设备通过一个中断通知操作系统。操作系统通过调用一个中断服务例程来回应这个中断，各种不同驱动和特定的中断相关联。这些中断服务例程中有一个是你的驱动指定的。在ISR执行的时候，当前IRP将完成，下一个IRP将被从设备队列中取出来调度执行实际的I/O（注：如果你开发设备驱动，你会注意到大多数上面描述的处理实际上是作为一个由ISR初始化的延迟过程调用（DPC）来执行的。但是，DPC也是执行在任意线程上下文中的。虽然本书不关注DPC和设备驱动开发，但是你可以查阅DDK得到更多的信息）。

这里要注意的一点是ISR是异步的执行在当前执行线程（任意线程）上下文中。因此，当处理这样一个中断的时候驱动不能假设他能访问的虚拟地址空间和那个发出这个现在正在处理的I/O请求的用户线程的一样。和那个线程相关的资源对驱动代码来说也是不能访问的，因为驱动不知道被借来执行ISR代码的是哪个线程的上下文。

线程和进程上下文的重要性

你的驱动代码将在前面描述的一个执行上下文中被调用。你开发的代码应该意识到将在什么执行上下文中被调用。因为这决定你的驱动必须在这个约束下运转。

考虑你要开发一个需要打开一些对象的内核驱动的情形；例如，你的驱动自己可能执行一些文件I/O，因此打开一个文件接收到一个返回来的文件句柄（注：虽然一个内核驱动希望执行文件I/O这看起来可能很奇怪，但是有一些过滤驱动需要这种能力。WINDOWS NT组件遵循的强力的基于对象的，层次化的模式使内核驱动有大量的弹性的可用的服务，这些导致能够设计非常有活力，有用的内核驱动程序）。如果你在你的驱动初始化代码（每个内核驱动都必须有的DriverEntry例程）中打开文件，你应该意识到这个句柄只有在内核进程和与这个内核进程相关的线程的上下文中才有效。那么，如果在系统工作者线程上下文中使用这个句柄是有效的。但是，如果你试图

在一个用户线程的上下文中使用这个句柄，或者一个任意线程上下文，你的句柄就不再有效了。类似的，如果你的驱动在一个用户线程的上下文中处理一个读请求的时候打开一个对象，这个句柄就只能在这个线程的上下文中使用。例如，任何试图在一个系统工作者线程中使用这个句柄都会导致一个错误。

你还必须知道什么时候你可以安全的使用传进驱动中用于读或者写I/O操作的用户缓冲区地址。用户指定的虚拟地址指针在那个用户线程上下文中是完全有效的。但是，如果这个I/O操作不是执行在那个用户线程上下文中（例如，I/O操作是异步执行的），用户应用程序传进来的虚拟地址就不再有效，因此不能在内核驱动中使用。I/O管理器提供支持在请求线程以外的上下文中访问用户缓冲区。我会在这章后面讨论支持的细节。

如上面讨论的，在你的驱动能够使用的资源上有一些约束，决定于你的代码执行的线程上下文。这个线程上下文决定于你的代码被调用的环境，这个线程上下文将决定你的驱动动能利用的资源。

对象和句柄

在WINDOWS NT执行体的内核组件创建的所有对象可以用两种方式引用，一种是使用当创建或者打开的时候NT对象管理器返回的对象句柄，一种是使用对象的指针。注意通常由内核组件分配的对象的指针在所有执行环境中都有效，因为对象引用的虚拟地址将在核心虚拟地址空间。但是，前面提到的，对象句柄是特定于得到这个句柄的线程的执行上下文的因此只有在那个特定在执行上下文中有效。

记住每个NT对象管理器创建的对象都有一个相关的引用计数器。当对象最初创建的时候这个引用计数器设置为1。每当一个内核组件请求对象管理器这样做时候引用计数器增加，通常是调用ObReferenceObjectByHandle（在DDK中描述）。引用计数器当在对象句柄上执行关闭操作的时候递减。内核驱动使用ZwClose系统服务调用来关闭任何系统创建的对象的句柄。引用计数器在内核组件调用ObDereferenceObject的时候也递减，这个函数要求传递对象的指针。当对象计数器变成0的时候，NT对象管理器将会删除这个对象。

在这本书是进程中，你会经常发现我们打开一个对象，收到一个句柄，然后得到这个对象的指针，然后藏到某个地方（可能在全局内存中），引用这个对象，关闭句柄。这给我们两个好处：

- 通过保存对象的指针，我们能够在不是原来打开对象的线程上下文中重新得到相同对象的句柄。你会在这本书后面发现这样的例子。
- 通过引用这个对象和关闭原来的句柄，我们确保对象不会被删除（直到我们最后一次解除引用），我们还确保一旦我们最后解除引用，对象将会自动被删除。

在头脑中记住上面的讨论，在你遍历这些讨论和遍及本书的代码的时候。这些关于对象和对象句柄的方法逻辑将可能在你开发你自己的内核驱动的时候大量使用。

公用数据结构

数据结构是任何计算机应用或者操作系统的核心。NT I/O定义的某些数据结构对内核驱动设计者和开发者是非常重要的。经常，你的驱动要创建和维护一个或者多个这些数据结构的实例来提供驱动的功能。在这一节，我会简要地对文件系统和过滤驱动开发者很重要的数据结构的结构和使用，注意所有这些结构都在DDK中有详细文档。但是，我们这里的目的是理解创建和使用这些数据结构的原因，还有去理解组成这些结构的重要的域。

驱动程序对象

DRIVER_OBJECT结构代表一个内存中加载的驱动程序的实例。注意一个内核驱动只能被加载一次；即是说，相同的驱动的多个实例不会被WINDOWS NT I/O管理器加载。驱动程序对象结构

定义如下：

```
typedef struct _DRIVER_OBJECT {
    CSHORT Type ;
    CSHORT Size;
    /* a linked list of all device objects created by the driver */
    PDEVICE_OBJECT DeviceObject;
    ULONG Flags;
    PVOID DriverStart;
    ULONG DriverSize;
    PVOID DriverSection;
    /*********************************/
    the following field is provided only in NT Version 4.0 and later
    /*********************************/
    PDRIVER_EXTENSION    DriverExtension;
    /*********************************/
    the following field is only provided in NT Version 3.51 and before
    /*********************************/
    ULONG      Count;
    /*********************************/
    UNICODE_STRING       DriverName;
    PUNICODE_STRING      HardwareDatabase;
    PFAST_IO_DISPATCH    FastloDispatch;
    PDRIVER_INITIALIZE   Driverlnit;
    PDRIVER_STARTIO      DriverStartlo;
    PDRIVER_UNLOAD       DriverUnload;
    PDRIVER_DI SPATCH    MajorFunction[IRP_MJ_MAXIMUM_FUNCTION +1];
} DRIVER_OBJECT;
```

在本章前面，我讨论了NT基于包的I/O模式。每个I/O请求包描述一个I/O请求。一个I/O请求包的主要功能是向驱动程序请求功能。

我们知道IRP将会被分派给一些I/O驱动例程。如果你观查驱动程序对象结构，你会注意到他包含一个叫做`MajorFunction`的函数指针的数组。内核驱动负责初始化这个数组的内容，其中每一个数组项是驱动支持的主函数。对于你的驱动必须支持的函数没有限制，也没有限制每个指针要指向一个唯一的函数；你可以初始化所有的主要函数指向一个例程，这也会工作的非常好（只要你的驱动例程处理了所有发给它的IRP），如果你开发内核驱动，你可能至少要支持一个主要函数，因此应该适当地初始化函数指针。

`DriverStartlo`和`DriverUnload`域也是留给驱动程序来初始化。低层的`WINDOWS NT`驱动提供一个`StartIO`例程，这个例程在`IRP`被分派给这个驱动的时候调用或者是当一个`IRP`从队列中弹出的时候调用。`DriverStartlo`域被低层驱动用来初始化为指向驱动提供的`StartIO`函数。通常，就象你会在本书中的代码看见的，文件系统驱动和过滤驱动不需要`DriverStartlo`例程，因为这些驱动程序通过其他的内部队列操作实现来管理他们的未决的`I/O`请求包。

`DriverUnload`应该指向一个在驱动程序卸载前执行的例程。这就允许你的内核驱动有机会确保任何磁盘上的信息是处于一致的状态，还允许低层驱动把他们控制的设备置为一个已知的状态。注意，并不要求你的驱动是可卸载的；在细节上，文件系统驱动要能够被设计为动态的卸载极其困难。如果你的驱动不能被卸载，你就不能初始化结构中的`DriverUnload`域（这个域被

I/O管理器初始化为NULL，因此你的驱动入口例程不需要做任何事）。

许多内核驱动创建一个或者多个设备对象结构。这些结构链接在驱动程序对象结构的DeviceObject域中。在驱动加载的时候，这个链表是空的。但是，NT I/O管理器用你的驱动程序使用IoCreateDevice服务例程创建的那些设备对象的指针填充这个链表。

要加载一个驱动程序，I/O管理器执行一个内部例程叫IoLoadDriver，这个例程执行下面的功能：

- 确定要加载的驱动程序的名字，检查是否已经加载的系统中。
  I/O管理器通过检查一个已加载内核模块的全局链表来检查驱动是否已经加载。如果驱动已经加载了，I/O管理器立即返回成功；否则，他继续执行加载驱动的处理。要加载你的驱动，你的安装工具必须已经创建了适当的注册表项。第三章有更多内核驱动和过滤驱动必须怎样配置注册表的信息。
- 如果驱动没有加载，I/O管理器请求VMM映射驱动程序的执行文件。在映射驱动程序代码的时候，VMM检查文件是否包含有效的WINDOWS NT 执行文件格式。如果驱动程序构建错误，VMM将使映射请求失败，放弃映射驱动程序，随后I/O管理器也会放弃加载驱动程序。
- 现在I/O管理器调用对象管理器，请求创建一个新的驱动程序对象。注意DRIVER_OBJECT类型是I/O管理器定义的对象类型，而且在系统初始化的时候由I/O管理器创建；因此NT对象管理器经过验证他是一个有效的对象类型。还要注意返回的驱动程序对象是非分页系统内存中分配的，因此在所有IRQL都可访问。
- I/O管理器把对象管理器返回的驱动程序对象结构清0。每个MajorFunction数组中的项被初始化为IopInvalidDeviceRequest，这是各种入口点的默认分派函数。这个例程仅仅设置一个返回状态STATUS_INVALID_DEVICE_REQUEST，然后把控制返回给调用进程。
- I/O管理器初始化驱动程序对象结构的Driverlnit域指向你驱动程序的初始化例程（DriverEntry）。DriverSection初始化为映射的执行印象的段对象指针，DriverStart初始化为这个驱动程序印象被映射的基地址，DriverSize初始化为驱动程序印象的大小。
- I/O管理器请求把这个对象插入到NT对象管理器维护的驱动程序对象链表中。接着I/O管理器得到这个对象的句柄。这个句柄由I/O管理器引用和关闭，从而保证在卸载驱动的时候（解除引用）对象会被删除。
- HardwareDatabase域被初始化为指向配置管理器的硬件配置信息的指针。低层驱动可以用这个域来检查当前启动周期的硬件配置信息。I/O管理器还初始化DriverName域以便需要的时候被错误记录组件使用。
- 最后，I/O管理器调用驱动初始化例程，这时候你的驱动得到初始化自己的机会，包括初始化驱动程序对象中的函数指针。你应该记住你的驱动初始化例程总是在IRQL为PASSIVE-LEVEL调用，使你可以使用更多的可用的系统服务。另外，你的初始化例程将在系统进程上下文中被调用；这一点尤其重要要记住，如果你打开或者创建任何对象得到一个句柄，任何这样的句柄只是在这个系统进程的上下文中才有效。为了能够在其他进程的上下文中使用这些句柄，你不得不使用这章中前面将的方法，你可以得到这个对象的指针然后需要的时候再在其他线程上下文中得到对象的句柄。
  如果你的驱动初始化例程失败，驱动将会自动被I/O管理器卸载。记住在控制返回给I/O管理器之前释放任何分配过的内存和关闭和解除引用任何引用过的对象。否则之后会降低和削弱系统性能。

内核驱动的入口例程就是被I/O管理器调用的初始化例程。每个驱动程序还可以注册一个重新初始化例程，这个例程在所有其他驱动被加载后，在I/O子系统的其他部分，还有其他内核组件都被加载后调用。在NT3.51及以前，驱动程序对象中的Count域保存着这个重新初始化例程被调用过的次数。

在NT4.0及以后版本，I/O管理器扩展原来的驱动程序对象，分配一个附加的结构。这个驱动扩展结构定义如下，包含为低层驱动管理硬件设备和外部设备的支持即插即用的域。在新版本中Count域移到了驱动扩展结构中；但是，他还是提供在原来版本中一样的功能。低层驱动提供的即插即用支持不会在本书中出现。

```
typedef struct _DRIVER_EXTENSION {
    // back pointer to driver object
    struct _DRIVER_OBJECT *DriverObject;
    // driver routine invoked when new device added
    PDRIVER_ADD_DEVICE AddDevice;
    ULONG Count;
    UNICODE_STRING ServiceKeyName;
) DRIVER_EXTENSION, *PDRIVER_EXTENSION;
```

最后，注意在驱动程序入口结构中有一个fast I/O dispatch table的指针。当前，只有文件系统实现提供支持fast I/O path。本质上，fast I/O path只是一个避免抽象的，干净的，模块化的，但也是相对比较慢的基于包的I/O的一个途径。使用文件系统驱动在这个结构中提供的函数指针，NT I/O管理器可以直接调用文件系统分派例程或者直接向NT缓存管理器请求I/O而不用建立一个IRP结构。FastIoDispatch域应该被文件系统初始化函数初始化为指向包含文件系统分派例程的适当的结构。在下一章的NT缓存管理器中，你会看见详细的讨论组成fast I/O方法的例程。

设备对象

设备对象被内核驱动创建来来代表一个逻辑的，虚拟的，或者物理的设备。例如，一个物理设备，比如一个磁盘驱动器，在内存中用一个设备对象来表示。类似的，假设你开发一个中间层驱动来把一个大的物理磁盘表示为三个小的磁盘或者分区。现在，将会有一个设备对象代表一个大的物理磁盘，是由低层磁盘驱动创建的，你的中间层驱动将会创建三个设备对象，每个代表一个虚拟磁盘。最后，驱动程序可能会创建设备对象来表示逻辑设备；例如，文件系统驱动创建一个设备对象来表示这个文件系统实现。这个设备对象可以被其他进程打开，还可以用于给文件系统驱动自己发送特殊的命令。

没有设备对象，这个内核驱动就收不到任何I/O请求，因为被I/O管理器分派的每个I/O请求必须要有一个目标设备。例如，如果你开发一个磁盘驱动程序但是不创建一个设备对象来表示这个特定的磁盘设备，就没有用户进程能够访问这个磁盘。但是，一旦你为这个磁盘创建一个设备对象，文件系统驱动就可以潜在的挂载这个物理介质上的任何卷，用户模式的进程就可以从这个磁盘上读取和写入数据。

没有名字的设备对象很少被内核驱动程序创建，因为这样的设备对象不容易被其他的内核驱动或者用户模式组件访问。如果你常见一个没有名字的设备对象，系统中就没有其他的组件能够打开他，因此，就没有组件会直接对他发起I/O请求。但是，未命名设备对象的一个常见的例子是那些文件系统驱动创建来代表加载的文件系统卷的设备对象。在这种情况下，有一个磁盘驱动程序创建的设备对象来代表物理的或者虚拟的磁盘，文件系统卷在这个物理的或者虚拟的磁盘上，还有一个卷参数块（VPB）结构在命名物理磁盘设备对象和文件系统驱动创建的未命名逻辑卷设备对象之间完成关联。I/O请求是发送给代表物理磁盘的设备对象。但是，I/O管理器检查这个磁盘上是否有挂载的卷（挂载卷在代表这个物理磁盘的设备对象的VPB结构中一个适当的标志来标识），如果有，他会把这个I/O请求重定向到表示这个加载卷的实例的未命名设备对象上。

当你的驱动程序调用IoCreateDevice来请求创建一个设备对象的时候，可以指定分配一些非分页内存和新创建的设备对象关联。原因是为这个特定的设备对象保留和关联一个全局内存区域。这个内存叫做设备对象扩展，I/O管理器将会为你的驱动分配。I/O管理器初始化DeviceExtension域

指向这个分配的内存。I/O管理器并不约束你的驱动怎样使用这个内存对象。你可能想知道请求设备扩展和声明全局静态变量之间的区别。答案可以总结为更清晰的代码设计。另一个重要的好处是设备特定的全局变量存储在设备对象中使得和设备对象立即变得逻辑相连了，因此你可以在避免访问这个设备对象之前不必要的使用同步资源。

你的内核驱动声明的任何静态变量对于整个WINDOWS NT操作系统都是全局的。他们不和任何特定的设备对象逻辑相连，因此如果你的驱动创建和管理多个设备对象结构，你将不得不设计一些方法来使得全局数据结构能够和特定的设备对象关联起来。但是，注意静态变量和设备扩展都是从非分页池中分配的，虽然你可以要求你的静态变量被分页（通常从不这个干）。许多内核驱动使用静态变量用于整个驱动程序，而使用设备扩展来包含特定于设备对象结构环境的的全局变量。

设备对象定义如下：

```
typedef struct _DEVICE_OBJECT {
    CSHORT                  Type;
    USHORT                  Size;
    LONG                    ReferenceCount ;
    struct _DRIVER_OBJECT   *DriverObject ;
    struct _DEVICE_OBJECT   *NextDevice ;
    struct _DEVICE_OBJECT   *AttachedDevice ;
    struct _IRP             *CurrentIrp;
    PIO_TIMER               Timer;
    ULONG                   Flags;
    ULONG                   Characteristics ;
    PVPB                    Vpb;
    PVOID                   DeviceExtension;
    DEVICEJTYPE             DeviceType;
    CCHAR                   StackSize;
    union {
        LIST_ENTRY          ListEntry;
        WAIT_CONTEXT_BLOCK  Wcb;
    } Queue;
    ULONG                   AlignmentRequirement
    KDEVICE_QUEUE           DeviceQueue;
    KDPC                    Dpc;
    ULONG                   ActiveThreadCount ;
    PSECURITY_DESCRIPTOR    SecurityDescriptor ;
    KEVENT                  DeviceLock;
    USHORT                  SectorSize;
    USHORT                  Sparel;
    /*******************************************************
    the following fields only exist in NT 4.0 and later
    *******************************************************/
    struct _DEVOBJ_EXTENSION    *DeviceObjectExtension;
    PVOID                   Reserved;
```

```
    /**************************************************
    the following field only exists in NT 3.51 and earlier versions
    **************************************************/
    LARGE_INTEGER                   Spare2;
} DEVICE_OBJECT;
```

任何内核驱动都可以用IoCreateDevice例程来要求I/O管理器创建设备对象。如果成功，这个例程返回指向从非分页内存中分配的设备对象的指针。设备对象中的许多域是保留给I/O管理器使用的。下面给出一些重要域的简要描述：

- 只要ReferenceCount域为非空，有两个事情就成立。首先，设备对象绝不会被删除，第二，创建这个设备对象的驱动程序的驱动程序对象绝不会被删除（也就是说：只要驱动程序创建的任何设备对象的引用记数为正值，这个驱动就绝不会被卸载）。ReferenceCount在不同是分别可以由I/O管理器和他的驱动程序来操作（注：如果你的驱动程序操作设备对象中的ReferenceCount域一定要小心，因为你没有办法和I/O管理器同步你的操作。这可能导致不一致的行为）。这个域被I/O管理器增加的一个例子是当一个新的文件流在一个挂载卷上打开的时候；表示这个挂载卷的设备对象的引用值被加1，从而保证只要有任何一个文件是打开的，文件系统驱动就不会卸载，因为卸载这个驱动可能导致系统崩溃。类似的，每当一个新的卷挂载的时候，代表逻辑卷的设备对象使他的引用值增加来确保设备对象和相应的驱动程序对象不会被删除。
- I/O管理器初始化域指向调用IoCreateDevice例程的内核驱动程序实例的驱动程序对象。
- 一个内核驱动程序创建的所有的设备对象用NextDevice域链接起来。注意这些设备对象在内核驱动中是没有特定的顺序的，遍历这个链表应该能找到创建的设备对象。新设备创建的时候，I/O管理器把他加在链表的头上，因此，你可能在链表的开始处找的最后创建的设备。
- 在这一章，还有第十二章（过滤驱动程序），你将看到怎样在WINDOWS NT环境开发过滤驱动程序的更多细节。过滤驱动程序是中间层的驱动程序，通过把自己插入到驱动层中和把自己挂接到目标设备驱动程序上来拦截到某些设备对象的I/O请求。挂接到一个设备对象的概念很简单，如图4-3所示：

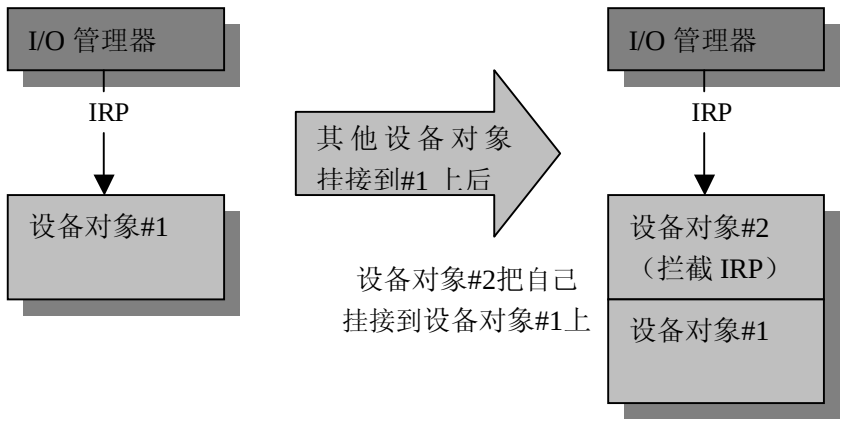


图 4-3 图示一个设备对象被挂接到其他设备对象上

当一个设备对象挂接到其他设备对象上的时候（通过I/O管理器提供的IoAttachDevice或者IoAttachDeviceByPointer例程），在被挂接的设备对象（图4-3中的设备对象#1）中的AttachedDevice域将被设置为挂接的设备对象的地址（设备对象#2）。

- CurrentIrp域对于那些设备驱动或者其他低层驱动设计者很重要。这些驱动程序通常使用I/O

管理器提供的IoStartNextPacket或者IoStartPacket例程从驱动程序的未决IRP队列中排队和出列一个IRP。一旦I/O管理器出列一个新的IRP，I/O管理器把出列的当前IRP交给驱动程序处理。为了这样做，他把IRP指针插入设备对象的CurrentIrp域。I/O管理器然后在调用设备驱动的StartIo分派例程的时候传一个DeviceObject->CurrentIrp指针。

这个域对于高层驱动程序不是很有用。

- Timer域是当驱动程序调用IoInitializeTimer例程的时候初始化。这就I/O管理器每秒调用驱动程序提供的定时器例程。
- 设备对象的Characteristics域描述这个设备对象代表的物理的，逻辑的，或者虚拟的设备的一些附加的属性。可能是值有FILE_REMOVABLE_MEDIA, FILE_READ_ONLY_DISK, FILE_FLOPPY_DISK, FILE_WRITE_ONCE_MEDIA, FILE_REMOTE_DEVICE, FILE_DEVICE_IS_MOUNTED, 或者FILE_VIRTUAL_VOLUME。这个域由I/O管理器，还有文件系统或者管理这个设备对象的内核驱动操作。
- DeviceLock是一个I/O管理器分配的同步类型事件对象。通常的，这个对象由I/O管理器用来优先分配给文件系统驱动的挂载请求。这可以同步对一个卷的多个挂载请求。只有你设计的文件系统驱动要使用I/O管理器提供的IoVerifyVolume例程（在第三部分描述）的时候要关注这个事件对象。在这种情况下，要小心不要在得到I/O管理器的挂载请求的时候调用这个例程。因为DeviceLock已经被I/O管理器预先得到来优先发送挂载IRP给你；调用这个例程会使得I/O管理器试图得到这个资源而导致死锁。
- I/O管理器为设备扩展分配内存，然后初始化DeviceExtension域指向这个分派的内存。

I/O请求包（IRP）

就象前面描述的，WINDOWS NT I/O子系统是基于包的。组成I/O子系统的内核驱动程序接I/O请求包（IRP），IRP包含了请求的细节。IRP接收者负责处理IRP，或者把IRP传给其他内核驱动进行附加的处理，或者完成这个IRP，指出IRP描述的请求已经处理完。

IRP分配

所有的I/O请求通过NT I/O管理器传递，觉大多情况下，一个用户进程执行一个WIN32或者其他子系统特定的I/O请求（例如，CreateFile（）），这个请求被转换为一个对I/O管理器的NT系统服务调用。根据接到的这个I/O请求，I/O管理器找出服务这个I/O请求的驱动程序。最有可能的，这会是一个I/O请求的目标文件所在的物理设备的挂载文件系统的驱动程序。

要分派请求给内核驱动，I/O管理器用IoAllocatelrp例程分配一个I/O请求包。这个结构可以在分页池中分配。在不同WINDOWS NT版本中分配的方法稍微有些不一样。

注意：“池”是NT执行体支持的一个系统定义的结构，用来更有效的管理分配和释放固定大小内存块。在第二章中有更详细的讨论。

在NT3.51及以前的版本中，I/O管理器尝试从一个由IRP结构固定大小组成的池中分配IRP。就象你将在IRP讨论中读到的，IRP的大小依赖于IRP要求的“stack locations”的数量。因此，I/O管理器维护两个可用的池，一个用于相对很少的stack locations的IRP，另一个用于有很多stack locations的IRP。如果尝试从中分配的池空了（这会在一种有很多I/O并发处理的高负载的时候发生），I/O管理器直接从VMM请求分配这个IRP（实际上，I/O管理器使用NT执行体提供的ExAllocatePool例程）。对于来自用户模式的I/O请求，如果当前没有内存可用就返回一个错误给用户应用程序指出系统可用资源耗光了。但是如果是来自内核模式的I/O请求，I/O管理器试图从内存池NonPagedPoolMustSucceed中为IRP分配内存。如果分配不成功将导致系统检查（BUGCHECK）。在NT4.0中使用的方法有一点变化：I/O管理器使用旁视列表（lookaside lists），在新版本中引入

的新的用来管理固定大小的内存池的数据结构，来代替“池”。使用这个新结构的原因是得到一些性能上的效率，因为旁视列表不总是使用自旋锁来执行同步；如果架构支持的话他使用原子8位比较交换指令来代替。

除了I/O管理器之外的其他的内核组件可以使用I/O管理器提供的IoAllocateIrp例程来请求一个新的IRP结构。这个IRP随后可以用来向一个内核驱动发送I/O请求。其他的I/O管理器提供的IRP相关例程也是使用IoAllocateIrp来得到一个IRP，然后在初始化某些域以后返回这个新的IRP，包括：IoMakeAssociatedIrp，IoBuildSynchronousFsdRequest，IoBuildDeviceloControlRecjuest和IoBuildAsynchronousFsdRecruest。参考DDK这些例程的更多信息。第三部分会在实现过滤驱动中使用其中一些例程。

IRP结构

逻辑上，每个IRP由下面的组成：

- IRP头部
- I/O栈位置（Stack Locations）

IRP头部包含关于I/O请求的通用信息，对I/O管理器和内核驱动有用的是这个请求的目标。IRP头部中的许多域可以被内核驱动访问；其他域的存在只是为了I/O管理器的方便，处理这个IRP的驱动程序应该把他们看作是禁止进入的。

这里是IRP头部中重要域的一个简单说明：

MdlAddress

内存描述符列表（MDL）是一个系统定义的用来描述组成一个缓冲区的虚拟地址范围的支持物理页面的数据结构。在I/O请求的处理中有不同的途径把缓冲区传给内核驱动程序。这很快会描述三种方法。但是现在记住，如果使用直接I/O（DirectIo），域会包含一个MDL结构的指针，可以用来进行数据传输。

AssociatedIrp

这是一个三个元素的联合，定义如下：

```
union {
    struct      _IRP *MasterIrp;
    LONG        IrpCount;
    PVOID       SystemBuffer;
} Associatedlrp;
```

任何已经分配的IRP可以被分为主IRP（master IRP）或者相关IRP（associated IRP），相关IRP从定义上是和某个主IRP关联的，能够被高层内核驱动程序创建的IRP。通过创建一个或者多个相关IRP，最高层驱动可以分割原来的IRP然后把某个相关IRP发送给驱动层中低层的驱动进行更多的处理。

例如，高层驱动有时候执行下面的循环：

```
while (还要求更多的处理) {
    使用IoMakeAssociatedlrp创建一个associated IRP ;
    用IoCallDriver  把associated IRP发送到低层驱动;
    if (  返回STATUS_PENDING) {
        wait on an event for the completion of the associated IRP;
    } else {
        associated IRP  完成了;
        检查返回值决定是否继续;
    }
}
```

对于一个相关 IRP，这里描述的联合包含一个主 IRP 的指针。但是，如果是主 IRP，这个联合包含这个主 IRP 的相关 IRP 的记数。或者如果没有创建相关 IRP，SystemBuffer 指针可能被初始化指向在内核地址空间中分配的一个用于数据传输的缓冲区。系统缓冲区是在内核驱动程序请求有缓冲 I/O（buffered I/O）的时候由 I/O 管理器创建的。

注意 IrpCount 域是在一个内部 I/O 管理器同步资源的保护下操作的。因此外部内核驱动程序一定不要试图直接操作和访问这个域的内容。

ThreadListEntry

这个域通常由 I/O 管理器操作。在通过 IoCallDriver 调用驱动程序的分配例程之前，所有的 I/O 管理器例行公事地把 IRP 插入发生 I/O 操作的线程的 IRP 链表中。例如，如果一个用户线程调用一个读请求，I/O 管理器将分配一个新的 IRP 结构，然后插入这个用户线程的将要被调用文件系统读分派例程的 IRP 列表中。

注意：在每个线程结构中有一个叫 IrpList 的域，用做未决 I/O 请求包的链表的头指针。前面讲的 ThreadListEntry 域就是用于把 IRP 插入这个链表。这个链表用于记录这个线程的所有提出的未决 IRP；这在 I/O 子系统试图取消特定线程的 IRP 的时候特别有用。注意 IoAllocateIrp 例程不会把返回的 IRP 出入当前线程的 IRP 链表中。因此，当取消一个已经投递的 IRP 的时候，不会在这个线程的 IRP 链表中找的要取消的 IRP。

IoStatus

你的内核驱动程序在完成这个 I/O 请求包之前应该适当的更新这个域。这个结构的描述在本章的后面。记住这个域是 IRP 结构的一部分，不是请求 I/O 操作的线程传给 I/O 管理器的 I/O 状态块结构的一部分。I/O 管理器负责从这个域转换 I/O 请求的结果到提出请求线程提交的 I/O 状态块结构。这个操作在 IRP 被内核驱动完成的时候由 I/O 管理器作为后处理进行执行。

RequestorMode

当代码在你的驱动中执行的时候，如果你知道调用这是一个用户模式线程（例如，一个应用程序请求 I/O 操作）还是内核组件（某些其他驱动在系统工作者线程环境中请求你的驱动的服务）是有用的。

你可能会奇怪这种信息为什么有用。考虑调用者是用户模式线程的情况；然后你知道不能盲目的假设传进驱动的参数都是有效的。如果你的驱动使用直接 I/O（direct-IO）的方法传递缓冲区指针（后面解释），你需要转换传进的地址为你的内核代码可访问的地址。这在请求将被你的驱动异步处理的是尤其重要。

另一方面，如果你的驱动是从系统工作者线程被调用，你就可以绕过这些参数检查，因为你可以假设传进驱动的地址是有效的，可以被你的驱动直接使用。

类似的，NT I/O 管理器，还有其他内核组件比如虚拟内存管理器，都需要识别和区分他们服务的客户执行在内核模式（操作系统），还是请求是来自一个用户模式的组件。这个信息用于检查传进这些内核组件的参数的合法性（注：如果 I/O 管理器读系统服务（NtReadFile）盲目地假设传进来的地址是一个合法的内核可用地址，那么恶意的用户就有机会覆盖操作系统数据）。

整个 NT 执行体的解决方法是识别调用线程调用这个内核组件的服务的时候执行的处理器的模式。记住这里的关键概念是调用线程先前的模式是重要的；非常真实的是在当前实例线程执行的是内核模式代码，当检查告诉我们的时候，当前模式总会是在内核模式。为了得到先前的模式信息，I/O 管理器直接访问线程结构中的一个域。DDK 中描述的 ExGetPreviousMode 函数为第三方驱动开发者提供相同的功能。这个例程返回被检查的线程的先前模式是用户模式还内核模式。

I/O 管理器把请求线程的先前模式放入 RequestorMode 域，然后调用 IoCallDriver 例程，这个例程接着调用你的驱动的一个分派例程。不管在你的驱动中，还是调用系统服务例程比如 MmProbeAndLockPages 你都应该使用这个信息。

PendingReturned

每个 IRP 会被驱动层中的一个或者多个处理，为了异步地处理一个 IRP，内核驱动必须执行下面的步骤：

a. 调用 IoMarklrpPending 函数标记 IRP 未决
b. 在内部排队这个 IRP，低层驱动可以使用 StartIo 函数代替
c. 返回状态 STATUS_PENDING，IoMarkIrpPending 调用简单地在当前 I/O 栈位置的 Control 域设置 SL_PENDING_RETURNED 标志。

在 IRP 处理完成的时候，执行 IoCompleteRequest 函数期间，I/O 管理器遍历在驱动层中的驱动程序使用过的每个栈位置，寻找需要被调用的任何完成例程。这个遍历和处理 IRP 是使用的栈顺序相反。最近被使用的栈位置最先处理（处理这个 IRP 的最低层驱动）。接着是下一个，以此类推。

在每个栈位置弹出的时候，I/O 管理器注意那些 SL_PENDING_RETURNED 标志是否在 I/O 栈位置中设置，如果是那么设置 PendingReturned 为 TRUE。但是，如果这个标志没有在 I/O 栈位置中设置 I/O 管理器就设置 PendingReturned 为 FALSE。

警告：PendingReturned 域的值可能在遍历 I/O 栈位置的时候改变，当 I/O 管理器寻找需要被调用的任何完成例程的时候。

那么为什么这个域的值那么重要呢？稍后在 IoCompleteRequest 中，I/O 管理器检查 PendingReturned 域的值来决定排队一个特殊内核异步过程调用（APC）到请求这个 I/O 操作的线程上。你的文件系统驱动或者过滤驱动将不得不和 I/O 管理器合作来确保正确的动作流程。你将在这一章后面看到你的驱动的动作怎样影响 I/O 管理器的行为。

Cancel, CancelIrql, 和 CancelRoutine

内核驱动处理 IRP 可能需要不明确的时间间隔才能完成的应该提供适当的 IRP 取消支持。我们的观点是文件系统驱动或者过滤驱动，如果我们不传递 IRP 给低层磁盘驱动或者网络驱动而是自己处理的话就应该提供这个功能。注意上面列出的三个域在要求提供取消未决 IRP 的能力的时候被驱动程序或者 I/O 管理器操作。

ApcEnvironment

当 IRP 完成的时候，I/O 管理器在 IRP 上执行后处理，处理的细节下面给出。ApcEnvironment 域由 I/O 管理器在原来请求 I/O 操作的线程上下文中执行 IRP 后处理的时候内部使用。这个域由 I/O 管理器在分派 IRP 的时候初始化，驱动程序设计者不应该访问它。

Zoned/AllocationFlags

Zoned 在 NT4.0 中代替 AllocationFlags 域。这个域记录内部簿记信息，用于 I/O 管理器在 IRP 完成的时候确定 IRP 从什么地方分配的，是从 Zoned/旁视列表，或者系统非分页池，或者从系统 nonpaged-mustsucceed pool。在驱动程序设计这的观点看来这些信息是无用的，除非在调试驱动程序试图定位所有 IRP 结构分配自全局旁视列表或者池中。

调用者提供的参数

下面是 IRP 结构的一部分：

```
PIO_STATUS_BLOCK    UserIosb;
PREVENT             UserEvent;
```

```
union {
    struct {
        PIO_APC_ROUTINE     UserApcRoutine;
        PVOID               UserApcContext;
    } AsynchronousParameters;
    LARGE_INTEGER       AllocationSize;
} Overlay;
```

IRP 中的 UserIosb 域被 I/O 管理器设置为指向请求线程提供的 I/O 状态块。作为在完成的 IRP 上执行后处理的一部分，I/O 管理器复制 IoStatus 域的内容到 UserIosb 指向的 I/O 状态块中。大多数 NT I/O 系统服务例程（在附录 A 中）接受一个可选的事件参数。这个参数（由调用者提供）被 I/O 管理器初始化为无信号状态而在 I/O 完成的时候设置为有信号状态。I/O 管理器用调用者提供的事件对象填充 UserEvent 域。

Overlay 结构中的 AllocationSize 域只有在文件创建请求中有效。用户可以为要创建的文件指定一个可选的初始大小。I/O 管理器初始化 AllocationSize 域为调用者提供的大小然后调用文件系统驱动的创建/打开分配例程。

NT I/O 管理器为 I/O 操作提供的许多 NT 系统服务允许异步操作。调用者线程可以请求异步执行请求的 I/O，还可以指定在 IRP 完成时候调用的 APC 例程。对于这些系统服务，I/O 管理器忠实地调用这些用户提供的 APC，传递提供的 APC 上下文，作为内核驱动程序完成 IRP 后 I/O 管理器执行的 IRP 后处理的一部分。I/O 管理器在 UserApcRoutine 域中存储调用线程指定的 APC 函数指针。调用上下文则存储在 **UserApcContext 域中。异步系统服务的一些例子有目录控制**，read, write 以及锁操作（lock）。要注意的是 create/open 请求总是同步处理，因此 AllocationSize 域以及 AsynchronousParameters 就组成 Overlay union 结构的要素。

对于牵涉数据传输的 I/O 操作，调用者提供数据缓冲区。这个缓冲区可以作为输入缓冲区，输出缓冲区或者充当两者。不管那种情况，I/O 管理器在调用 **IoCallDriver()之前用调用者提供的缓冲区指针初始化 UserBuffer 域。在 IRP 完成的时候，如果有数据需要复制回调用者的缓冲区，I/O 管理器就在执行 IRP 后加工的时候执行这个操作。如果你的驱动程序没有指定直接 I/O 或缓冲 I/O 作为用户缓冲区的操作方式，I/O 管理器将假设你将自己处理用户缓冲区，因此不会分配 MDL，或者给你的驱动提供系统缓冲区地址。你的驱动程序接着可以直接使用在 UserBuffer 域中的缓冲区指针（注：注意用户模式虚拟地址只有在请求 I/O 的用户模式线程中有效。如果你的驱动程序是一个文件系统实现，你可以在用户模式线程上下文中立即完成 I/O，此时你可以直接使用用户提供的虚拟地址，否则你的驱动程序将必须为用户提供的缓冲区得到适当的系统模式的地址，在你把 IRP 排队到系统工作者线程等待稍后处理之前）。**

Tail

IRP 有一个定义如下的 Tail 结构：

```
union {
    struct {
        KDEVICE_QUEUE_ENTRY     DeviceQueueEntry;
        PETHREAD                Thread;
        PCHAR                   AuxiliaryBuffer;
        LIST_ENTRY              ListEntry;
        struct _IO_STACK_LOCATION   *CurrentStackLocation;
        PFILE_OBJECT            OriginalFileObject;
```

```
    } Overlay;
    KAPC        Apc;
    ULONG       CompletionKey;
} Tail;
```

这个结构中的内容只能被 NT I/O 管理器直接操作和访问。并不推荐你的驱动程序直接访问这些域。

DeviceQueueEntry 用来排队到一个特定低层驱动程序的 IRP。大多数低层驱动程序允许 NT I/O 管理器维护一个未决 I/O 请求包的列表。当设备驱动程序分派例程调用 IoStartPacket()的时候发现设备对象忙的时候，使用 DeviceQueueEntry 域来排队到这个目标设备对象的请求包。DDK 中描述了 IoStartPacket(), IoStartNextPacket(), 和 IoStartNextPacketByKey()支持例程，他们操作这个域。内核模式驱动程序不要尝试直接访问和操作 DeviceQueueEntry 域的内容。

在分派 IRP 之前，I/O 管理器初始化 Thread 指向分派将发生的上下文的线程。这个域随后被低层驱动程序和文件系统驱动程序使用。

考虑出现硬件错误的情况。文件系统使用 IoRaiseInformationalHardError()调用来在系统控制台弹出消息框提示用户错误的情况。这个调用是阻塞的，他通过投递特殊的内核 APC 到目标线程来显示错误信息。问题是显示消息框的那个线程将被阻塞，直到一个用户物理地把错误消息从系统控制台消除。但是，如果在 IoRaiseInformationalHardError()参数中没有指定线程，错误消息将被分派到一个特定的系统工作者线程。

通常，如果任何错误发生，内核模式驱动程序将检查 Overlay.Thread 域来确定线程是否是一个系统工作者线程。如果是的话，他会向 IoRaiseInformationalHardError()提供一个 NULL 线程参数，因为把系统工作者线程阻塞不确定的时间是非常不可接受的。

Thread 域的重要性的另一个实例是在处理可移动介质中。如果读/写可移动介质的时候发生用户引起的错误，低层驱动程序使用 IoSetHardErrorOrVerifyDevice()例程来指出发生了某些意外，高层驱动程序也应当向用户报告错误或者查证在驱动器中介质是正确的。响应这个调用，I/O 管理器仅仅在 IRP 中的 Overlay.Thread 域指向的线程对象的 DeviceToVerify 域中存储需要查证的设备对象。更高层的驱动程序（文件系统）接着调用 IoGetDeviceToVerify()的时候提供从 Overlay. Thread 域得到的线程对象指针，作为响应，I/O 管理器退还存储的设备对象指针。

注意 I/O 管理器服务例程 IoAllocateIrp()并不设置返回的 IRP 中的 Thread 对象。这是调用这个例程的调用者的责任。

AuxiliaryBuffer 的存在是为了支持向内核模式驱动程序传递一些没有包含在 IRP 中的额外的信息。但是，到现在为止，没有 I/O 管理器例程使用这个域来传递信息给内核模式驱动程序（注：如果你开发文件系统驱动程序的话，你可能注意到对于目录控制 IRP 来说这个域不为空，但是，同样的包含目录名的缓冲区指针在 IRP 的当前栈单元中的 Parameters.QueryDirectory.FileName 域中也是可访问的）。

CurrentStackLocation 域是一个简单的指向 IRP 当前栈位置的指针。栈位置在本章后面讨论。要记住重要的一点是内核驱动总应该使用 I/O 管理器提供的访问函数来得到 IRP 的当前和下一个栈位置的指针。为了维持可移植性，你的驱动千万不要试图直接访问这个域的内容。

OriginalFileObject 域被 I/O 管理器初始化为 I/O 操作目标的文件对象的地址。相同的信息对于当前栈位置发送的 I/O 操作的最高层驱动也可用。但是，I/O 管理器在 IRP 头中保存这个信息，因此能够在低层驱动操作的栈位置中独立的方式访问。文件对象在 IRP 处理完的时候进行后处理。例如，如果文件对象不为 NULL（就是说，OriginalFileObject

域在 IRP 分配的时候初始化了），I/O 管理器检查是否需要发送消息到一个完成端口，或者解除引用任何事件对象，或者在这个文件对象上执行类似的通知或者清除操作。这个域为 NULL 也是合法的，在这种情况下，I/O 管理器会跳过那些他应该执行的后处理。Apc域被I/O管理器在IRP完成时排队一个APC请求到发出I/O请求线程的上下文中执行IRP的最后的后处理的时候内部使用。

前面提到的，每个I/O请求包由IRP头和IRP的栈位置组成。一些IRP结构中的域比如StackCount, CurrentLocation, 和 CurrentStackLocation和栈位置操作有关。IRP栈位置接下来讨论。

栈位置和重用IRP结构

WINDOWS NT I/O请求包是可重用的。在分层驱动环境中，例如WINDOWS NT I/O子系统，驱动层中每个高层驱动调用下一个低层的驱动程序，直到某些驱动实际完成原来的IRP。非常可能，也是常常发生的，同一IRP会从驱动传到驱动直到他被完成。

完成IRP要求调用IoCompleteRequest，当这个调用发出之后，没有组件能够接触IRP，除了I/O管理器，因为这个IRP可能在任何时候被释放。

那么一个 IRP 结构是怎样能够重用的呢？NT I/O 管理器提供的解决方法是使用包含对目标设备 I/O 请求描述的栈位置。当最初分派 IRP 给一个内核驱动的时候，I/O 管理器用要进行的操作的参数填充一个栈位置。接下来，接收这个 IRP 的驱动确定自己完成这个 IRP，还是需要调用驱动层中的其他驱动。如果他需要调用低层的驱动，IRP 的当前所有者可以简单地初始化下一个 IRP 栈位置，然后通过 IoCallDriver 例程来调用低层驱动，传递这个 IRP。 这个过程不断重复直到这个链中的一个驱动执行了所有请求的处理而决定完成这个 IRP。

当 IRP 结构分配的时候，NT I/O 管理器分配多个关联的栈位置的空间。每个栈位置能够包含一个 I/O 请求的完整描述。例如，为读请求分配的 IRP 结构应该包含下面的信息：

- 一个功能码，内核驱动用来确定发出的请求类型。在这个例子里，功能码指出是读请求。
- 一个读数据的开始的偏移量。
- 输出缓冲区的指针。

除了上面的，还有其他的与读请求相关的信息可能也被传递给这个读操作的目标设备对象的的驱动程序。所有这些信息都封装在一个栈位置结构中。

分配给 IRP 的栈位置的数量依赖于 IRP 针对的目标设备对象的 StackSize 域。当设备对象创建的时候 StackSize 域设置为 1。然后可以被管理这个设备对象的驱动程序设置为任何值。当一个设备对象附加到其他设备对象上的时候 StackSize 域也会改变。作为附加操作的一部分，StackSize 被设置为被附加设备对象的值增加 1。这里的逻辑很简单：一个 IRP 发送给初始设备对象需要一个栈位置；然后每个过滤驱动或者驱动层中的将对这个 IRP 执行某些处理的驱动程序需要一个栈位置。

就象在图 4-4 中显示的，如果读请求发送给磁盘 A 上的一个加载卷的文件系统驱动程序，I/O 管理器在创建读 IRP 的时候将分配 4 个栈位置。这些栈位置以反向的顺序使用，类似于后进先出的栈结构的用法。当调用一个驱动的时候，I/O 管理器总压栈栈位置指针来指向下一个栈位置；当调用的驱动释放 IRP 的时候，栈位置指针弹出重新指向先前的栈位置。因此，当调用下图 4-4 中过滤驱动程序分派例程的时候，I/O 管理器使用栈位置#4，最后分配的栈位置。

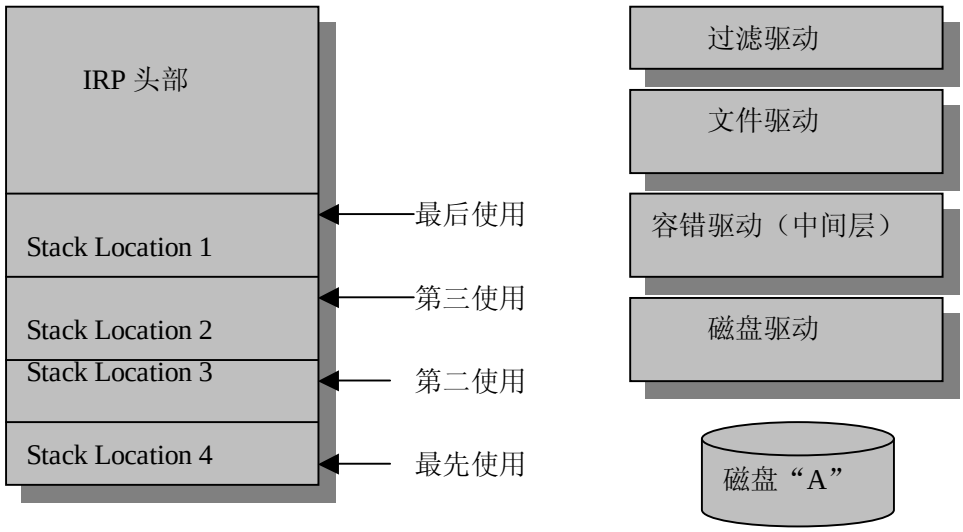


图 4-4 驱动层中的 IRP 栈位置

NT I/O 管理器 IRP 头部中的 StackCount 域为分配给 IRP 的栈的总数。CurrentLocation 被初始化为（StackCount+1）。这个值在每次通过 IoCallDriver 例程来调用一个驱动程序分派例程的时候递增。

因此，如果 StackCount 是 4，CurrentLocation 的初始值就为 5，这是一个无效的栈位置指针值。但是，这样做的原因是在分派 IRP 给层中下一个驱动的时候，内核组件总会取下一个栈位置的指针然后给这个请求填充适当的参数。

当一个 IRP 被分派给层中第一个驱动的时候，下一个栈位置将等于 4（CurrentLocation - 1），就是上面的用于过滤驱动的正确的栈位置。

I/O 管理器经常使用这个值来执行健壮性检查来保证 IRP 通过 I/O 子系统正确地分派给驱动了。例如，在 IoCallDriver 例程中，I/O 管理器首先递减 CurrentLocation 域（因为一个新驱动被调用，他需要下一个栈位置），然后检查 CurrentLocation 的值是否小于或者等于 0，如果只个值变得小于或者等于 0，很明显 IoCallDriver 对于初始分配栈位置的数量来说被调用的太多了（或者有一些漂移指针正破坏内存），因此 I/O 管理器用错误码 NO_MORE_IRP_STACK_LOCATIONS 执行一个系统检查（BUGCHECK）。

注意：系统检查（BUGCHECK）的原因是，在调用 IoCallDriver 的时候，危急的损害可能已经造成了，因此在被调用的驱动要使用的下一个栈位置里的内容可能被填入了任何可能的内容。但是，在这种情况下，下一个栈位置是 IRP 结构的末尾后面的一些没有分配的内存，其中可能包含着任意的内容。在这时候继续执行会导致一系列的问题，包括可能损害用户数据。

I/O 管理器维护一个指向当前栈位置的指针，除了前面提到的 CurrentLocation 值之外。这个指针维护在 IRP 头部包含的 Tail.Overlay 结构的 CurrentStackLocation 域中。内核驱动绝不要试图自己操作 CurrentLocation 或者 CurrentStackLocation 域的内容（注：NT 文件系统确实自己在这些域上执行一些秘密的操作。但是，对大多数内核驱动来说，最后还是使用 I/O 管理器提供的访问方法来查看和修改这些域的内容）。I/O 管理器为驱动提供了取得当前栈位置的指针的例程，调用 IoGetCurrentIrpStackLocation，要得到下一个栈位置指针使用 IoGetNextIrpStackLocation，因此驱动程序就能够为层中的下一个驱动设置适当的栈位置，很少使用情况下使用 IoSetNextIrpStackLocation 来设置栈位置的值。

栈位置结构定义在 NT DDK 中，有些组成域和栈位置描述的原始 I/O 请求是独立的，这是这些域：

MajorFunction

NT I/O 管理器定义了一系列主要函数，每个标志内核驱动能够实现的一个通用功能。函数用功能码或者数字识别，这一系列函数特意定义很全面的，因此功能码用于所有类型的 NT 内核驱动，包括文件系统驱动，中间层驱动，设备驱动和其他低层驱动。

当一个 IRP 分发到一个内核驱动的时候，驱动必须解释当前栈位置中的 MajorFunction 来从驱动中找出期望的函数。可能的主函数代码在下面：

```
#define IRP_MJ_CREATE     0x00
#define IRP_MJ_CREATE_NAMED_PIPE 0x01
#define IRP_MJ_CLOSE      0x02
#define IRP_MJ_READ 0x03
#define IRP_MJ_WRITE      0x04
#define IRP_MJ_QUERY_INFORMATION 0x05
#define IRP_MJ_SET_INFORMATION      0x06
#define IRP_MJ_QUERY_EA      0x07
#define IRP_MJ_SET_EA     0x08
#define IRP_MJ_FLUSH_BUFFERS    0x09
#define IRP_MJ_QUERY_VOLUME_INFORMATION OxOa
#define IRP_MJ_SET_VOLUME_INFORMATION OxOb
#define IRP_MJ_DIRECTORY_CONTROL OxOc
#define IRP_MJ_FILE_SYSTEM_CONTROL OxOd
#define IRP_MJ_DEVICE_CONTROL OxOe
#define IRP_MJ_INTERNAL_DEVICE_CONTROL OxOf
#define IRP_MJ_SHUTDOWN 0x10
#define IRP_MJ_LOCK_CONTROL Oxll
#define IRP_MJ_CLEANUP 0x12
#define IRP_MJ_CREATE_MAILSLOT 0x13
#define IRP_MJ_QUERY_SECURITY 0x14
#define IRP_MJ_SET_SECURITY 0x15
#define IRP_MJ_QUERY_POWER 0x16
#define IRP_MJ_SET_POWER 0x17
#define IRP_MJ_DEVICE_CHANGE 0x18
#define IRP_MJ_QUERY_QUOTA 0x19
#define IRP_MJ_SET_QUOTA Oxla
#define IRP_MJ_PNP_POWER Oxlb
#define IRP_MJ_MAXIMUM_FUNCTION Oxlc
```

从 IRP_MJ_DEVICE_CHANGE 到更高的功能码是在 NT4.0 中引入的。也不是所有的主功能码都被实现了；例如，配额相关的功能码在本地 NT 文件系统驱动中没有任何支持。上面列出的功能码没有一个是强制内核驱动实现的，除了驱动管理的对象的打开和关闭能力之外。打开和关闭操作很重要是因为如果打开操作失败，没有 I/O 请求能够提交，因此就没有存在的任何对象可以作为 I/O 请求的目标。类似的，如果打开成功，关闭操作最终会被调用，而且关闭操作不能失败（I/O 管理器不会检查关闭操作的返回值）。

因此如果你没有实现关闭操作来补足你的打开，系统可能最后会泄露资源，这依赖于在打开操作期间执行的操作，还依赖于打开操作中创建的数据结构。

在文件系统驱动和过滤驱动环境中的主功能码在第三部分讨论。

MinorFunction

辅助功能码（Minor function codes）提供 I/O 栈位置中的主功能码个更多的信息。例如，考虑上面的主功能码 IRP_MJ_DIRECTORY_CONTROL，I/O 管理器发送包含这个功能码的 IRP 给文件系统驱动。目的是执行一些文件目录操作。但是，I/O 管理器希望文件系统驱动执行什么样的目录控制操作？

可用的操作包括取得目录内容信息的操作(IRP_MN_QUERY_DIRECTORY)和当目标目录中的某些文件或者目录属性发生变化的时候通知 I/O 管理器(IRP_MN_NOTIFY_CHANGE_DIRECTORY)。

Flags

标记（Flags）域也提供期望目标驱动执行功能的附加的修饰信息。例如，考虑前面讨论的 IRP_MJ_DIRECTORY_CONTROL 主功能码。如果辅助功能码是 IRP_MN_QUERY_DIRECTORY，标记（Flags）域可以包含附加的信息，使得当被请求的目录内容返回的时候文件系统做不同的行为。

例如，如果 SL_RESTART_SCAN 标志被是设置，文件系统驱动将从被请求的目录开始从新扫描。或者如果标志被设置，文件系统驱动将仅仅返回匹配指定搜索标准的第一个项。

低层驱动也关心这个标志的设置。例如，如果文件系统驱动分派给可移动介质驱动一个 SL_OVERRIDE_VERIFY_VOLUME 标志被设置的 IRP，他将执行一个读请求。但是如果这个标志没有被设置，而且设备驱动程序发现介质发生变换（通知文件系统获悉），他就会放弃所有的 I/O 请求，包括所有读请求。

Control（控制）

当内核驱动必须异步的处理一个 IRP 的时候，驱动可以排队这个 IRP，通过调用 IoMarkIrpPending 来标记为“未决”，然后把控制返回给调用者。调用 IoMarkIrpPending 只是设置当前栈位置的 Control 域的 SL_PENDING_RETUKNED 标志。任何内核驱动都可以检查 Control 域的这个标志。

这个标志在内部也被 NT I/O 管理器用来存储关于是否应该调用和当前栈位置关联的完成例程，如果在 IRP 完成时返回码指出这是一个成功的，失败的，或者取消的操作的时候。这些标志被指定为 SL_INVOKE_ON_SUCCESS, SL_INVOKE_ON_FAILURE, 和 SL_INVOKE_ON_CANCEL。内核驱动通常不需要直接关心这些标志的状态。

`DeviceObject`

这个域在 I/O 管理器在 IoCallDriver 例程中执行的时候设置。内容被设置为目标设备对象的设备对象指针。

FileObject

I/O 管理器设置这个域为 I/O 操作的目标对象的文件对象指针。注意只是在你的驱动中调用 IoAllocateIrp 例程不会使这个域被设置。如果你想使用返回的 IRP 在特定的文件对象上操作，你必须自己设置这个域。

CompletionRoutine

这个域的内容在调用 IoSetCompletionRoutine 宏的时候由 I/O 管理器设置。I/O 管理器在 IRP 完成的时候执行后处理的时候要检查完成例程。如果提供了一个完成例程，这个例程将在执行后处理的线程的上下文中调用；通常这是调用 IoCompleteRequest 例程的那个线程上下文。因为 IRP 经常被低层驱动在高 IRQL 上完成，所以这个完成例程很可能

将在一个高 IRQL 被调用。

你应该知道完成例程以“最后指定最先调用”的顺序被调用。因此，最高层驱动的完成例程将在其他所有完成例程被调用之后才被调用。如果任何驱动从调用驱动指定的完成例程中返回 STATUS_MORE_PROCESSING_REQUIRED，I/O 管理器立即停止这个 IRP 的所有后处理。为这个 IRP 释放内存的责任落到返回 STATUS_MORE_PROCESSING_REQUIRED 状态的那个驱动身上。

如果你开发高层驱动象文件驱动或者过滤驱动，如果你指定一个完成例程，在你的完成例程中总要执行下列的代码序列：

```
if (PtrIrp->PendingReturned) {
    IoMarklrpPending(Ptrlrp);
}
```

如果你忘记这样做，如果在你的驱动上面还有其他驱动在调用层中，IRP 可能会被错误地处理，而你可能陷入驱动或者进程“悬挂”（hang）。悬挂的原因将在本章后面进一步解释。

Context

这个域包含内核驱动为 IRP 指定完成例程的时候指定的环境信息。

如果你开发中间层驱动程序，当你准备传递 IRP 给层中下一个驱动的时候，你将不得不小心的复制在当前 I/O 栈位置中包含的一些值到下一个 I/O 栈位置。例如，你必须复制 Flags 域，以便低层驱动知道他将执行一个文件系统驱动请求的 I/O 读操作，即使先前他已经通知文件系统介质发生了变化。

处理 IRP

处理一个发送给你的驱动的 IRP 可以很简单。下面的的四个图举例说明处理一个分派给内核驱动的 IRP 的一些通用的方法。

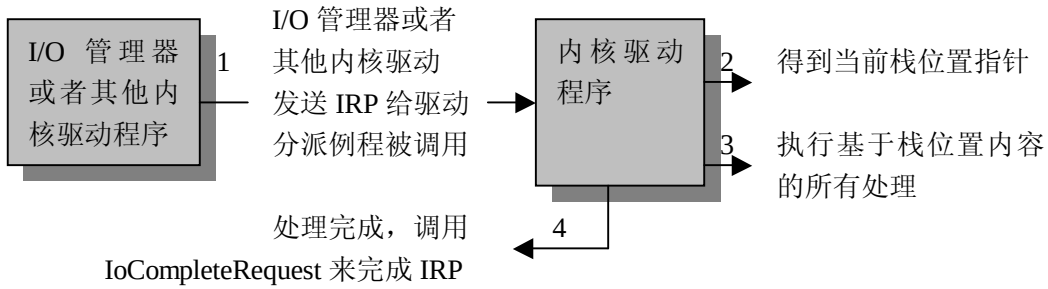


图 4-5 被调用驱动完成 IRP 时候的简单 IRP 处理

在图 4-5 中你能看见目标内核驱动收到一个 IRP，得到当前栈位置指针，执行基于栈位置内容的一些处理，最后完成 IRP。但是，注意在接收到请求和开始处理之前可能存在一个延迟，因为驱动可能会排队这个 IRP，如果他现在正忙于处理其他请求的话。排队的 IRP 随后将在工作者线程上下文中得到处理。

还要注意一旦驱动从调用 IoCompleteRequest 后得到控制，绝对不能再接触这个 IRP 或者 IRP 中的域。这样做可能会导致数据丢失和系统崩溃。

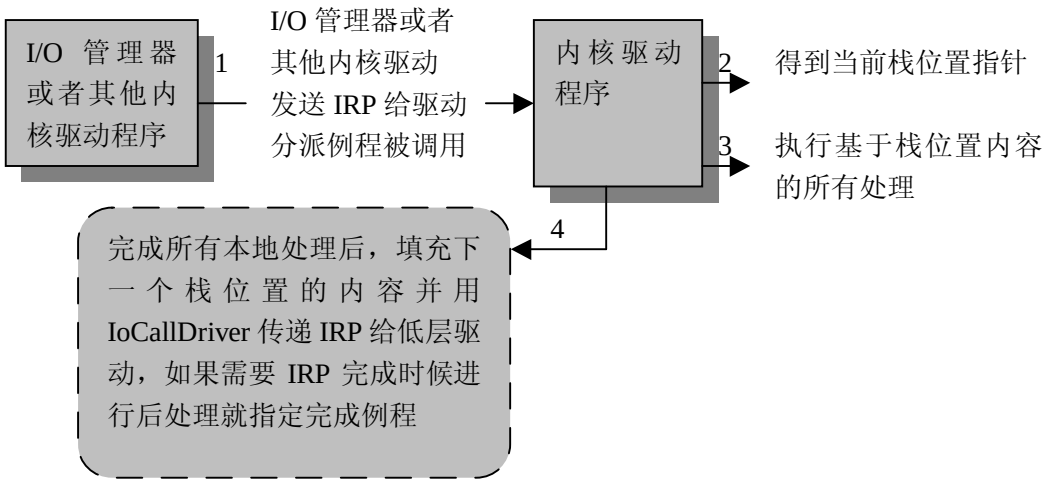


图 4-6 被调用驱动重用 IRP 并传递 IRP 给低层驱动时候的 IRP 处理

图 4-6 举例说明内核驱动收到一个 IRP，得到当前栈位置指针，执行基于栈位置内容的一些处理，但是，这个驱动在请求的功能完全完成之前可能需要调用低层驱动的服务。因此，IRP 接收者可以初始化下一个栈位置的内容并传递 IRP 给驱动层中的低层驱动。

如果你的驱动传递 IRP 给其他驱动，就不允许再访问这个 IRP 了，因为他不知道低层驱动什么时候会完成那个特定的 IRP。通常，传递 IRP 是通过调用 IoCallDriver 来完成的。I/O 管理器将在调用 IoCallDriver 的线程上下文中调用低层驱动；但是，低层驱动收到 IRP 可能会返回 STATUS_PENDING 然后异步地完成这个 IRP。

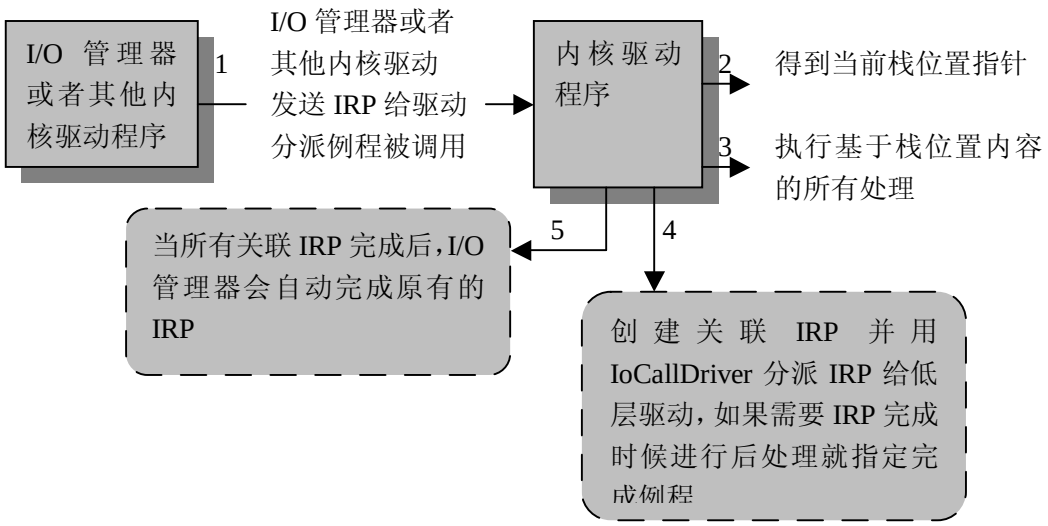


图 4-7 使用关联 IRP 结构来处理 IRP 一个 IRP

图 4-7 说明当高层内核驱动程序（例如文件系统驱动程序）用关联 IRP 来提交 I/O 请求给其他低层驱动程序的流程序列。例如，高层驱动希望分割一个 I/O 请求的时候可能这样做；还有就是如果高层驱动需要执行多个低层驱动集的处理的时候。

注意高层驱动不需要在原来的 IRP 上调用 IoCompleteRequest；当所有关联 IRP 被低层驱动完成后，I/O 管理器会自动完成原有的 IRP。但是，高层驱动可以在关联 IRP 完成的时候请求执行一个完成例程，因此给他机会来执行某些后处理，也允许他自己有机会在他自己的机制中完成原来的 IRP。

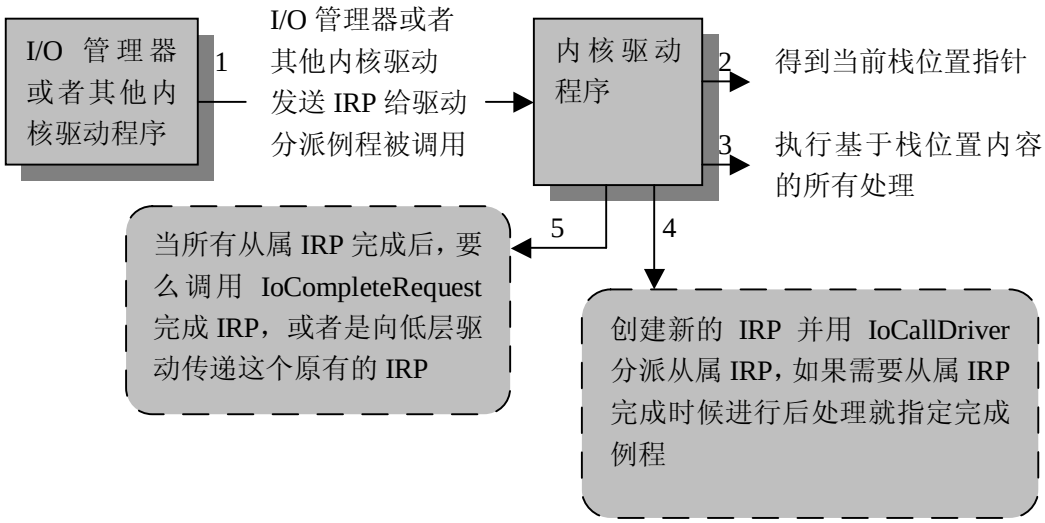


图 4-8 使用新分配的 IRP 结构来帮助处理一个 IRP

图 4-8 说明一种和使用关联 IRP 不同的方法；这里内核驱动使用 I/O 管理器提供的函数创建新的 I/O 请求包，然后把新的 IRP 分派给其他内核驱动。一旦新的 IRP 完成的时候，原来的 IRP 可以重新分派给低层驱动进行更多的处理或者立即完成。

IRP 完成和释放

每个必须完成，以便 I/O 管理器得到这个 IRP 中包含的请求已经处理完成的通知。要完成 IRP，内核驱动必须调用 I/O 管理器提供的 IoCompleteRequest 例程。

一旦这个例程调用的时候，NT I/O 管理器在这个完成的 IRP 上执行一些后处理，如下：

1， I/O 管理器执行一些基本的检查来确保这个 IRP 是处于有效的状态。当前栈位置指针的值要验证确保是小于或者等于（栈总数+1）。如果这个无效，系统将用错误码 MULTIPLE_IRP_COMPLETE_REQUESTS 执行系统校验（BUGCHECK）。如果安装的是调试版本的操作系统，I/O 管理器会执行一些附加的诊断。比如当完成 IRP 的时候检查返回码 STATUS_PENDING，检查从这个驱动返回的其他无效的返回码。

2， 现在 I/O 管理器开始扫描这个 IRP 中所有的栈位置，寻找需要被调用的完成例程。每个栈位置可以有一个关联的完成例程，他能否被调用依赖于最后的返回码是成功还是失败，或者这个 IRP 是否被取消。I/O 栈位置以反向的顺序被扫描，I/O 栈位置最高值的被最先扫描。这导致例如一个磁盘驱动（最底层驱动）提供的完成例程会最先被调用，而最高层驱动（通常是文件系统驱动）提供的完成例程最后被调用。

完成例程在那个调用 IoCompleteRequest 的线程上下文中调用。如果任何完成例程返回 STATUS_MORE_PROCESSING_REQUIRED，I/O 管理器立即停止这个 IRP 的所有后处理，返回控制给调用的 IoCompleteRequest 例程。现在，为这个 IRP 调用 IoFreeIrp 释放内存的责任落到返回 STATUS_MORE_PROCESSING_REQUIRED 状态的那个驱动身上（注：DDK 假定只会被内核驱动为他自己创建的关联 IRP 调用。这没有问题，但是，要防止你的驱动对 I/O 管理器分派给你的一般 IRP 返回这个值。问题是有许多的这个 IRP 的后处理会被因为你的完成例程返回这个状态码而突然中断。你的驱动不得不设计一个方法使得这些后处理在以后能从新进行；这不是一个简单的工作）。

3， 如果被完成的 IRP 是一个关联 IRP，I/O 管理器就递减主 IRP 的 AssociatedIrp.IrpCount 域的值。然后 I/O 管理器调用一个内部例程 IopFreeIrpAndMdls 来释放为这个关联 IRP 分配的内存和 MDL 结构。最后，如果恰好是主 IRP 发出的关联 IRP 的最后一个 IRP 完

成，I/O 管理器就在主 IRP 上调用 IoCompleteRequest。

4， I/O 管理器要在原来请求 I/O 操作的线程上下文中执行很多后处理。为了这样做，I/O 管理器排队一个内核 APC，这个 APC 将随后在请求线程的上下文中执行。但是，这个方法不能用于某些类型的 IRP 结构，用于下列类型的操作：

Close 操作

描述一个 Close（关闭）操作的 IRP 是由 I/O 管理器产生的，每当一个内核对象的最后一个引用被删除的时候发送给相关的内核驱动程序。这可能在一个特殊的内核 APC 执行完的时候发生。要在一个 I/O 管理器定义的内核对象上执行 Close 操作，总是调用一个叫做 IopCloseFile 的内部 I/O 管理器例程。IopCloseFile 是同步的因此会阻塞。他分派和发送一个 IRP 给目标内核驱动并等待一个完成这个关闭操作的事件。因此，当完成一个关闭操作的 IRP 的时候，I/O 管理器只是复制返回状态（顺便说明，关闭操作的请求线程从不检查这个状态），通知执行 IopCloseFile 的线程等待的事件，然后立即返回控制。这个 IRP 随后在 IopCloseFile 中被删除。

分页 I/O 请求（Paging I/O requests）

分页 I/O 请求是 NT VMM 发出的。在 5-8 章，你会读到 NT VMM 和 NT 缓存管理器提供的相关的功能。但是现在，只要理解 I/O 管理器在完成一个分页 I/O 请求的时候不能发生页错误，因为那会导致系统崩溃。因此，I/O 管理器在完成一个分页 I/O 请求的时候会做下面两件事中的一件：

— 对于一个同步分页 I/O 请求，I/O 管理器会复制返回的 I/O 状态到调用者提供的 I/O 状态块结构中，通知调用者可能正在等待的内核事件对象，然后释放 IRP 和返回控制，因此这里没有附加的后操作要执行。

— 对于一个异步分页 I/O 请求，I/O 管理器会排队一个特定的 APC 到请求分页 I/O 的线程上下文中等待执行。这是修改者写者（MPW）线程，他是 VMM 子系统的一个组件。在下一章你会读到关于 MPW 线程的更多的信息。现在，知道这个例程执行在 MPW 线程（APC 被提交的那个）上下文中就足够了，从分页读操作中复制状态到修改页写者提供的 I/O 状态块中，随后用另一个内核 APC 调用一个 MPW 完成例程。

在后面，你会看见 I/O 管理器通常释放和这个 IRP 关联的所有 MDL，在释放这个 IRP 之前。但是，对于分页 I/O 操作，使用的 MDL 结构属于 VMM（就是说由 VMM 分配和只由 VMM 在 I/O 完成的时候释放）。这就是 I/O 管理器并不释放在分页 I/O 请求中使用的 MDL 结构的原因。

挂载请求（Mount requests）

如果你检查在 NT DDK 中提供的标志，那些指示分页 I/O 请求和挂载请求（分别是 IRP_PAGING_IO 和 IRP_MOTJNT_COMPLETION），你会注意到他们定义为相同的值。这是因为 I/O 管理器严格地把挂载请求当作一个同步的分页 I/O 读请求一样。因此，I/O 管理器为挂载请求执行和同步分页 I/O 请求描述的一样的后处理。

5， 如果 IRP 描述的不是分页 I/O，关闭，或者挂载请求，I/O 管理器接下来解锁任何锁住的和 IRP 关联的 MDL 描述的页面。注意 MDL 结构这时候不会释放；他的释放在请求线程上下文中执行的后处理中执行。

6， 在这时候，I/O 管理器已经完成了他能完成的所有后处理，除了在请求线程上下文中执行的。因此 I/O 管理器排队一个特定内核 APC 到请求这个 I/O 操作的线程上下文中。调用线程上下文中调用的 I/O 管理器内部例程叫做 IoCompleteRequest。但是，很可能没有任何线程来发送 APC 请求。这在线程开始一个异步 I/O 操作后就退出的情况可能发生，请求已经被低层驱动中开始，但是驱动程序不能在一个超时时间段中完成这个请求。在

这种情况下，I/O 管理器放弃这个请求，接着，他简单地释放为这个 IRP 分派的内存，因为没有更多的后处理可以执行了。

对于同步 I/O 操作，I/O 管理器不排队特定内核 APC，而是简单地立即返回控制。这些 IRP 结构的 Flags 域设置为 IRP_DEFER_IO_COMPLETTON 标志。例如，能够被延迟完成的 IRP 的主功能码是目录控制操作，读操作，写操作，创建/打开请求，查询文件信息和设置文件信息请求。通过立即返回控制，，I/O 管理器避免了和排队内核模式 APC 相关的负载和响应 APC 中断的负载。相反，当控制返回给他的时候，原来请求 I/O 操作的线程直接调用 IoCallDriver 来调用 IoCompleteRequest。这只是 NT I/O 管理器最佳的执行。

注意，为上面的情况，I/O 管理器会执行两个检查来确定 APC 是否应该被排队或者不应该排队：

— IRP_DEFER_IO_COMPLETION 标志是否设置为 TRUE。

— Irp->PendingReturned 域是否设置为 FALSE。

只有上面的两种情况都为 TRUE，I/O 管理器才简单地从 IoCompleteRequest 函数返回。

如果你在你的驱动中不小心处理的话，下面的情形可能导致问题的出现：

— 在传递一个请求给一个低层驱动程序之前你的驱动指定一个完成例程。

— 在调用层中你的驱动上面有驱动程序（例如，过滤驱动或者中间层驱动）。

— 如果 Irp->PendingReturned 域设置为 TRUE，你的驱动没有执行前面列出的调用 IoMarkIrpPending。

现在 I/O 管理器可能错误地认为没有 APC 被排队（认为将会在请求线程的上下文中执行完成）而原来的线程会永远阻塞。

其他的 I/O 管理器不排队 APC 的情形是如果文件对象有一个完成端口和他关联，这种情况下 I/O 管理器发送一个消息到完成端口。

在这时候，在 IoCompleteRequest 中要被执行的所有处理都完成了。

剩下的发生在原来请求这个 I/O 操作的线程上下文中的步骤描述如下。NT I/O 管理器在前面提到的 IoCompleteRequest 中执行这些步骤。

1， 对于缓存 I/O 操作，I/O 管理器复制成功执行 I/O 操作的结果的任何数据到调用者缓冲区中。缓冲 I/O 操作的细节在本章的后面提供；但是，现在注意如果驱动返回一个错误或者驱动在 IRP 的 IoStatus 结构中返回值指出要求一个检验操作，就不执行复制操作（注：你应该理解的是 NT I/O 管理器把警告的状态码作为成功的操作状态；即是说，即使状态码不是 STATUS_SUCCESS，但是只要不是一个错误 I/O 管理器就会复制数据到调用者缓冲区中）。

还有，复制到调用者缓冲区的字节数等于 IoStatus 结构中的 Information 域的值；因此如果这个域没有正确的设置，调用者就不会得到所有的或者任何的返回数据。

一旦复制操作执行完成，I/O 管理器分配的缓冲区也会被释放。

2， 任何与这个 IRP 关联的 MDL 在这时候释放。

提示：对于文件系统驱动可能故意的返回一个缓存管理器分配的 MDL 结构指针，当一个调用者请求读或者写 I/O 请求的时候。这些请求用发送给文件系统的 IRP 中的 IRP 栈位置的 MinorFunction 域的 IRP_MN_MDL 标志来区别。因为在这一点上所有和 IRP 关联的 MDL 是被盲目释放的，但是看起来文件系统驱动没有许多需要返回调用者 MDL 的地方。但是，当前唯一使用 IRP_MN_MDL 标志的内核客户只有 LAN 管理器服（LAN Manager）务模块，这个模块通常用在完成例程中返回 STATUS_MORE_PROCESSING_REQUIRED 来处理这个问题。见第九章，文件系统怎样处理 MDL 读和 MDL 写请求的讨论。

3， I/O 管理器复制 Status 和 Information 域到调用者提供的状态块结构中。
4， 如果调用者提供了一个要被通知的事件对象，I/O 管理器就通知那个事件对象。I/O 管理器通知和 I/O 请求包关联的任何文件对象中的 Event 域中的事件对象，如果调用者没有提供事件对象，这个 I/O 操作就被同步执行，因为这个文件对象是只为同步访问打开的。
5， 通常，NT I/O 管理器递增和任何 IRP 关联的文件对象或者事件对象的引用记数，在把这个 IRP 传递给驱动程序处理之前。现在，I/O 管理器递减这些对象的引用记数。
6， I/O 管理器从当前线程的未决 IRP 列表中取出这个 IRP。
7， 这个 IRP 的内存最后被释放；如果这个 IRP 是从池/旁视列表中分配的，这个包的内存被返回给池/旁视列表重新使用；否则内存返回给系统。

处理 I/O 请求包

有一些关键的概念你必须很好的理解，在小心处理发送给你的内核驱动的 I/O 请求包的时候：

- 一旦你的驱动收到一个 IRP，系统中没有其他组件，包括 I/O 管理器，能够并发访问这个 IRP。直到你的驱动传递这个 IRP 给其他内核驱动程序，或者完成这个 IRP，处理这个 IRP 描述的请求是你的驱动的单独的责任。
- 一旦你的驱动完成这个 IRP 或者把他传递给另一个内核驱动程序，你的驱动必须放弃这个 IRP 的控制，不要试图访问他的任何一个域。唯一的能够再次接触这个 IRP 的时间是如果你你在传递这个 IRP 的时候指定的完成例程。在这种情况下，I/O 管理器会在 IRP 完成期间的后处理执行中调用你的完成例程。
- 如果你指定一个在 IRP 完成的时候调用的完成例程，他可以执行任何必要的后处理。记住，虽然，你的完成例程可能会在 IRQL 小于或者等于 DISPATCH-LEVEL 被调用。如果你的完成例程在高 IRQL 等级被调用，那么当你的代码执行的时候不能发生任何的页错误。你可以从你的完成例程中返回 STATUS_MORE_PROCESSING_REQUIRED 来停止这个 IRP 所有的后处理。但是，当你这样做的时候一定要小心，特别是从低层驱动中返回这个值的时候，因为某些处于驱动链中更高层的驱动指定的完成例程通常会被调用，但是现在不能被调用了，除非对这个 IRP 做一些欺骗的手段。
- 没有 IRP 可以被完成一次以上（注：有可能一个完成例程会返回 STATUS_MORE_PROCESSING_REQUIRED，对这个 IRP 执行一些特定的后处理，然后在这个 IRP 上调用一个 IoCompleteRequest 来使 I/O 管理器正确地处理这个 IRP。这是这个规则的唯一的例外而导致一个 IRP 被完成一次以上的情况）。如果你试图这样做不管是故意的还是错误的，你可能造成数据损坏和/或系统崩溃。虽然检查 IRP 被完成一次以上是 I/O 管理器的责任，但是这个检查不是完全安全的，因此在你设计你的驱动的时候要意识到这个要求。
- 你的驱动不能盲目的假设会在原来请求 I/O 操作的线程是上下文中处理这个 IRP。事实上，低层驱动，例如中间层驱动和设备驱动程序，可能从来不会在调用线程的上下文中调用分派例程。因此，当在处理 I/O 请求包的时候试图访问对象，句柄，资源和内存的时候必须小心。理解你的分派例程被调用的上下文和对于那个特定的上下文的可用和有效的资源。
- 内核驱动在能够做的范围上有极大的只有。同时，内核模式的代码的责任也比用户空间的应用更大。如果你的驱动使用用户空间代码发送的缓冲区的指针，要小心你怎样使用那样的缓冲区。可能内核驱动轻易地妥协系统完整性而误用，或者没有小心地确认，包含在那些无防卫的用户模式应用发送的缓冲区指针中的缓冲区和数据。确认调用者的模

式来确认他发送给你的指针是否有效。使用调用者的先前模式来帮助你决定用户提供的缓冲区是否有效。

- 只使用 I/O 管理器提供的访问方法来操作 IRP 中的栈位置。内核驱动很可能要修改 IRP 的栈位置，这会影响 IRP 开始怎样被处理和 IRP 完成是时候怎样执行后处理。要抵制那些任何未文档的操作栈位置内容的方式的诱惑。
- 如果你想利用驱动层中上面或下面的其他驱动的服务，使用你自己的 IRP。避免使用私有的通信通道是不可扩展的。要创建 IRP 结构，使用 I/O 管理器提供的支持例程之一（即是说，IoAllocateIrp(), IoBuildSynchronousFsdRequest(), IoBuildAsynchronousFsdRequest() , IoBuildDeviceIoControlRequest(), 和 IoMakeAssociatedIrp()）。使用 IoInitializeIrp 联合 IoAllocateIrp 来初始化 IRP 头部中的通用域。小心重读前面的节来确定那些附加的你可能想要初始化的域。还有，IoFreeIrp 会不会需要调用依赖于你可能指定的任何完成例程中返回的状态码。
- 某些内核组件例如 LAN Manager 服务器，从内部池中分配 IRP，而不是从 I/O 管理器请求。要明白这些组件可能使用 IRP 中的某些域的方式和 I/O 管理器使用这些域的方式有区别。因此，当依赖于 DDK 中没有文档化的，I/O 管理器想要保持私有的域的内容时要小心了因为系统不保证在这些域中总是包含一致的值。

  另外，象 LAN Manager 服务器这样的组件在他们分配的 IRP 经常有一个最大栈位置的数值。如果你附加一个或者多个过滤或者中间层驱动到驱动层中，要求的栈位置的数值就会超过 LAN Manager 服务器能够处理的最大值。有一个处理这个问题的方法，你可以指示用户通过一个注册表参数来指定 LAN Manager 服务器应该分配的附加的栈位置。

卷参数块（VPB）

VPB 是代表加载卷的文件系统设备对象和代表包含物理的文件系统数据结构的物理或者虚拟磁盘的设备对象之间的链接。每个当一个打开磁盘上文件流的请求发送给物理或者虚拟设备的设备对象的时候（注：因为 WINDOWS NT 平台最常用的子系统是 WIN32 子系统，考虑一个用户执行一个打开驱动器符号 C：上的一个文件流的情况。这个驱动器符号无关紧要，但是 WIN32 子系统知道这实际上是一个 WINDOWS NT 命名的符号连接，比如是 \Device\HardDiskO\Partition1。访问 C：上的文件流就是访问名字为 \Device\HardDiskO\Partition1 的物理磁盘设备对象的上的文件流。注意 WINDOWS NT 命名的对象不是代表挂载卷的设备对象，相反，他是表示物理/虚拟磁盘驱动器的设备对象。VPB 用来在命名物理/虚拟硬盘设备对象和未命名的逻辑卷设备对象之间的关联），I/O 管理器调用一个叫做的内部例程。如果这个 VPB 关联的请求指示的目标物理/虚拟设备还没有挂载，这个例程负责初始化一个逻辑卷挂载操作，但是，如果这个卷预先挂载了，I/O 管理器就把这个打开操作重定向到从 VPB 的域得到的设备对象的指针上。

VPB 的内存是 I/O 管理器在调用 IoCreateDevice 创建设备对象的时候或者文件系统调用 IoVerifyVolume 调用的时候自动从非分页池中分配的，为下面类型的设备对象：

- FILE_DEVICE_DISK
- FILE_DEVICE_CD_ROM
- FILE_DEVICE_TAPE
- FILE_DEVICE_VTRTUAL_DISK（用于 RAM 磁盘或者任何类似的能够挂载卷的虚拟磁盘结构）

注意每个这种类型的设备对象上可以有一个逻辑卷，通常这些设备对象的每个类型也代表一个单一的设备的可挂载分区。VPB 用于映射文件系统（逻辑的）卷设备对象到也由一个设备对象表示的物理设备上。这个结构在 I/O 管理器分配的时候初始化为零。下面的定义描述

了 VPB 结构：

```
typedef struct _VPB {
    CSHORT                  Type;
    CSHORT                  Size;
    USHORT                  Flags;
    USHORT                  VolumeLabelLength; // in bytes
    struct _DEVICE_OBJECT   *DeviceObject;
    struct _DEVICE_OBJECT   *RealDevice;
    ULONG                   SerialNumber;
    ULONG                   ReferenceCount;
    WCHAR       VolumeLabel[MAXIMUM_VOLUME_LABEL_LENGTH / sizeof(WCHAR)];
} VPB, *PVPB;
```

每个挂载的卷可以有一个最大长度在 32 个字符的标签和他关联。VolumeLabelLength 域被文件系统初始化为这个卷的标签的实际长度，存储在 VolumeLabel 域中。每个文件系统可以有一个序列号和他关联，这个序列号应该被文件系统驱动从这个卷读出然后放入 SerialNumber 域中。只要 VPB 的引用值不为 0，I/O 管理器就不会删除这个 VPB 结构。RealDevice 由 I/O 管理器初始化为包含可挂载逻辑卷的物理或者虚拟设备对象的指针。DeviceObject 域由文件系统在挂载操作发生的时候初始化。这个域包含由文件系统创建来代表挂载卷的类型为 FILE_DEVICE_DISK_FILE_SYSTEM 的设备对象的地址。

VPB 结构中的 Flags 域可以是下面三个值的一个：

VPB_MOUNTED

这个位在一个文件系统挂载了这个VPB代表的逻辑卷的时候由I/O管理器设置。这发生在文件系统从发送给他的主功能码为IRP_MOUNT_COMPLETION的IRP上返回STATUS_SUCCESS的时候。

VPB_LOCKED

这个位可以被挂载这个 VPB 代表的逻辑卷的文件系统驱动设置或者清除。当这个位被设置的时候，I/O 管理器会放弃所有以这个逻辑卷为目的的打开/创建请求。文件系统可以响应应用请求锁定这个逻辑卷的时候设置这个位，或者如果他们想要临时的防止任何的创建/打开请求。FASTFAT 文件系统通过设置这个位响应应用的 IOCTL 请求锁定卷的请求(FSCTL_LOCK_VOLUME)。

VPB_PERSISTENT

这个域也由文件系统操作。如果这个域被设置，I/O 管理器就不会删除这个 VPB 结构，即使 VPB 中的 ReferenceCount 为 0。

NT I/O 管理器提供了两个例程来过滤驱动和文件系统驱动使用来同步访问 VPB 结构。这些例程定义如下：

```
VOID
IoAcquireVpbSpinLock (
OUT PKIRQL Irql
) ;
VOID
IoReleaseVpbSpinLock (
IN KIRQL Irql
) ;
```

参数：

Irql

对于 IoAcquireVpbSpinLock，这里返回一个当相应的释放例程被调用的时候必须归还的 IRQL 的指针。

对于例程 IoReleaseVpbSpinLock，这个参数包含得到自旋锁的时候返回的 IRQL 值。

功能提供：

有一个全局的自旋锁结构在 I/O 管理器操作 VPB 内容的时候在内部获得。如果你的驱动希望检查或者操作 VPB 中的 Flags，DeviceObject 或者 ReferenceCount 域，你应该首先调用 IoAcquireVpbSpinLock 支持例程来确保数据一致性。注意这是一个全局自旋锁，当这个自旋锁被得到的时候，没有多少 I/O 操作能够继续（例如，新的创建和打开操作会阻塞）。因此在访问特定域的时候一定注意短时间的获得这个自旋锁。

挂载操作执行流的更多细节，以及可移动介质上的挂载卷的 VPB 操作的细节说明在第三部分中。

I/O 状态块（I/O Status Block）

I/O 状态块用来转达一个 I/O 操作的结果。这个结构定义如下：

```
typedef struct _IO_STATUS_BLOCK {
    NTSTATUS    Status;
    ULONG       Information;
} IO_STATUS_BLOCK, *PIO_STATUS_BLOCK;
```

每个 IRP 有一个 I/O 状态块和他关联。内核驱动总应该把描述请求处理的结果的返回码插入到 I/O 状态块结构的 Status 域中。因此，这个域将包含一个指示成功的返回码(STATUS_SUCCESS)，警告的返回码，一个通知消息或者一个错误。错误状态码也包括那些指出在处理一个 I/O 请求的时候发生了一个异常（已经被驱动处理了）的情况。参考前面讨论 NT 返回码的章节。

Information 域通常包含着与这个请求的 I/O 操作相关的任何附加信息。例如，对于一个读 1024 字节的请求，Information 域在返回的时候将会包含实际读取的字节数，如果 Status 域指示的是 STATUS_SUCCESS 信息。因此，Information 域在这种情况下将包含一个在 0 到 1024 之间的值。

文件对象（File Object）

如果你开发 WINDOWS NT 中的文件系统驱动，或者你开发位于驱动层中文件系统驱动上面的过滤驱动，你会变的非常熟悉文件对象结构。文件对象是 I/O 拥有的在内存中表示一个打开对象。例如，如果一个打开磁盘上文件的操作成功执行了，I/O 管理器创建一个文件对象结构来表示这个打开操作的特定实例。如果在同一个文件流上执行另一个打开操作，I/O 管理器将创建一个新的文件对象来代表第二个打开操作，即使尽管两个打开操作实际上是对同磁盘上的同一个文件。

你应该在概念上把文件对象作为一个成功执行打开/创建请求的结果而创建的句柄的内核等价物。文件对象不局限与表示打开的文件流；其实，他是一个用来表示任何 NT I/O 管理器打开的对象的抽象。因此，如果你打开一个逻辑卷或者磁盘驱动设备对象，这个打开操作会导致创建和初始化一个文件对象数据结果。

所有目标为磁盘上文件流或者逻辑卷的 I/O 请求要求一个文件对象结构作为请求的目标（你不能在真空中执行一个读请求；你必须有一个先前成功对你要执行读操作的目标的打开操作的目标文件对象）。创建和维护一个文件对象数据结构是 NT I/O 管理器和文件系统驱动的共同责任。

文件对象结构由 I/O 管理器在传递打开或者创建请求给内核文件系统驱动之前创建。创建/打开 IRP 包含一个新分配的文件对象结构的指针；处理这个打开/创建请求来初始化这个文件对象结构中的某些域是文件系统的责任。

文件对象结构由 NT I/O 管理器定义的：

```
typedef struct _FILE_OBJECT {
    CSHORT                  Type;
    CSHORT                  Size;
    PDEVICE_OBJECT          DeviceObject;
    PVPB                    Vpb;
    PVOID                   FsContext;
    PVOID                   FsContext2;
    PSECTION_OBJECT_POINTERS SectionObjectPointer;
    PVOID                   PrivateCacheMap;
    NTSTATUS                FinalStatus;
    Struct _FILE_OBJECT     *RelatedFileObject;
    BOOLEAN                 LockOperation;
    BOOLEAN                 DeletePending;
    BOOLEAN                 ReadAccess;
    BOOLEAN                 WriteAccess;
    BOOLEAN                 DeleteAccess;
    BOOLEAN                 SharedRead;
    BOOLEAN                 SharedWrite;
    BOOLEAN                 SharedDelete;
    ULONG                   Flags;
    UNICODE_STRING          FileName;
    LARGE_INTEGER           CurrentByteOffset;
    ULONG                   Waiters;
    ULONG                   Busy ;
    PVOID                   LastLock;
    KEVENT                  Lock;
    KEVENT                  Event;
    PIO_COMPLETION_CONTEXT  CompletionContext;
} FILE_OBJECT;
```

DeviceObject 和 Vpb 域在 I/O 管理器传送创建或者打开请求给文件系统驱动之前初始化。

DeviceObject 被初始化为请求指向的目的物理或者虚拟设备对象的地址。Vpb 域初始化为和目标设备对象关联的挂载卷的 VPB。

FsContext，FsContext2，SectionObjectPointer，PrivateCacheMap 域是由文件系统驱动和 NT 缓存管理器初始化和/或维护。他们将在本书的后面更详细地讨论。NT I/O 管理器不维护这些域的内容，尽管他要检查和使用 FsContext 域的内容，这将在第三部分讨论。

FileName 域由 I/O 管理器初始化为表示要打开的文件，卷或者物理设备的字符串。这个名字可以是相对路径名或者是一个绝对路径名。相对路径名在 RelatedFileObject 域的非空值来表示。这个域中包含一个先前打开的文件对象数据结构的指针。在 FileName 域中提供的相对路径名现在必须被认为是相对于 RelatedFileObject 域表示的文件的名字。注意 RelatedFileObject 域只有在创建请求的时候有效，其他的任何时候这个域的内容都是未定义的。

CurrentByteOffset 域由文件系统为那些只为同步访问而打开的文件对象来维护的。这个域包含文件流的当前指针位置，这个指针在成功完成读/或者写 I/O 操作的时候更新。

CompletionContext 域由 I/O 管理器用来在完成一个 IRP 的时候发送一个消息到本地过程调用（LPC）端口。DeletePending 标志当文件系统收到一个设置信息的 IRP 指定这个文件流应该被删除的时候在文件对象结构中设置。

LockOperation 域在拥有这个文件对象结构的线程在文件打开是时候至少调用了一次字节范围锁定操作的情况下设置为 TRUE。这个域在随后当线程关闭这个文件对象来确定是否要发送一个解锁 IRP 给文件系统驱动的时候被检查。

不同的访问域（ReadAccess, WriteAccess, 和 DeleteAccess）由 I/O 管理器设置个清除。类似的还有各种共享访问相关域（）。这些域的状态决定文件当前是怎样打开的，还决定了随后的打开的请求某些特定类型的访问会被允许进行还是被以一个 STATUS_SHARING_VIOLATION 错误码拒绝。有一个叫 IoCheckShareAccess 的 I/O 管理器支持例程，维护着这些域的状态。这个例程通常只由文件系统驱动调用，在本书的后面会描述。

在本章后面，你会读到从文件系统驱动的观点来看同步和异步处理 I/O 操作要实现这些要求必须提供的代码。用户可以打开一个文件对象并指定所有在这个打开的特定对象上执行的操作要同步执行。这是在这个文件对象结构的 Flags 域设置 FO_SYNCHRONOUS_FLAG 来指出，设置是由 I/O 管理器作为创建/打开请求的一部分来执行的。要求同步 I/O 操作的一个影响是 I/O 管理器总是排队这个特定文件对象的所有的 I/O 操作。要实现这个序列化操作，NT I/O 管理器使用文件对象结构中的 Busy 和 Waiters 域。Busy 域当那个特定文件对象在处理中的时候使用。Waiters 域指示等待使用这同一个文件对象执行 I/O 操作的线程的数量。这些域不应该被其他内核驱动太多关注。

文件对象是一个可等待的内核对象，也就是说，线程可以要求异步 I/O，然后等待这个 I/O 操作完成。文件对象结构中的 Event 域由 I/O 管理器用来维护等待对象的状态。这个事件对象在一个 I/O 操作开始使用那个文件对象的时候被 I/O 管理器设置为无信号状态。随后当这个 I/O 操作完成的时候设置为有信号，尽管 如果只有在调用者没有明确地指定另一个等待的事件对象的时候。

Flags 域可以反映许多值，其他一个已经在这里描述了，每个都描述一个和文件对象结构关联的状态。我会推迟讨论这个域的每个可能的值，直到在我们的代码中实际使用到的时候。

确定要使用哪个对象

这里有一些简单的“把所有东西放在一起”的规则，当你开发你的驱动的时候：

- 当你的驱动加载的时候，I/O 管理器将创建和发送一个驱动程序对象给你的初始化例程。你必须填充驱动程序对象中的某些域，比如你想支持的功能的各种分派函数的指针。如果你没有填充函数指针，你的驱动就不会收到任何请求，因为所有请求将被默认例程处理（IopInvalidDeviceRequest()）。
- 为了提供任何功能，你可能要创建至少一个设备对象。更有可能，你会创建一个设备对象表示你的驱动程序，还有其他的设备对象表示你支持的其他虚拟和/或物理设备。大多数你创建的这些设备对象是命名的，除非您开发文件系统驱动，在这种情况下，大多数设备对象将代表逻辑卷，因此是未命名的。当对请求创建一个设备对象的时候，你可以指定一个设备扩展来存放和这个新设备对象关联的全局数据。
- 如果你写过滤驱动，你要为你需要拦截 I/O 操作的每个目标设备对象创建一个设备对象。随后你会把你的设备对象附加到目标设备对象上。这个附加的处理过程会导致所有以被附加的设备对象为目标的 I/O 请求被指向你的驱动程序对象。

- 如果你开发文件系统驱动，你必须操作你执行挂载操作的物理设备对象的卷参数块（VPB）。执行一个挂载会导致 I/O 管理器使得物理设备对象可读/写访问，这些请求将被送到代表挂载逻辑卷的你的设备对象上。
- 一旦你的设备对象可以接收 I/O 请求，请求就会被封装在 IRP 中发送给你。如果你开发文件系统驱动，你还能收到通过快速通道（fast path）的请求（本书后面会有更多快速通道的相关内容）。
- 当你收到一个 IRP，你要确定你的驱动被请求执行的本来的 I/O 操作。为了这样做，你应该得到 IRP 中当前栈位置的指针，然后使用他取出这个 I/O 请求的相关信息。你的驱动然后要对这个 IRP 执行适当的处理，不管是同步还是异步的。
- 你的驱动可能能完成这个 IRP，或者他可能决定这个 IRP 需要发送到驱动层中更第的驱动以进行一些附加的处理。在这种情况下，你可以得到 IRP 中的下一个栈位置的指针然后填充信息，随后驱动层次中的下一个驱动可以取处理确定自己要执行本来的处理。
- 如果你的驱动要完成这个 IRP，他必须在 I/O 状态块结构中返回 I/O 操作的结果。Status 域应该包含结果，而 Information 域应该包含你希望返回给调用者的任何附加信息。
- 最后，如果你开发文件系统驱动，可能在处理一个打开请求（和随后处理大多数 IRP 的时候）的时候会访问和可能修改文件对象结构。每个这样的结构表示一个成功打开操作的实例。

除了本章中提到的对象之外，如果你开发设备驱动，你还要关注其他的对象，包括控制器对象，适赔器对象和中断对象。

另外，你的驱动程序无庸质疑地要创建一个或者多个自己的对象类型。例如，文件系统驱动将在内存中创建一些文件流的内部表示。对于那些熟悉 UNIX 操作系统环境，回想 V 节点结构是被所有文件系统创建和维护的。这个结构的 NT 等价物是文件控制块（FCB），我们将在第三部分讨论的对象。还有，文件系统驱动会创建一个上下文在内部表示一个文件打开操作的实例（类似于系统定义的文件对象结构）。在 WINDOWS NT 说法上这个结构叫做环境控制块（Context Control Block）。

一旦你在你的代码开发中使用这些对象，他们应当变成你的第二本能，你将毫不费力的断定设备对象代表的什么。

I/O 请求：一次讨论

下面的讨论提供一些附加的信息，这些信息在我们开发高层驱动程序的时候你应该记在脑海中。他们不仅使用在本书的驱动例子程序中，而且应用在任何你设计和开发的商业内核驱动中。

同步/异步 操作

有一些 I/O 操作总是 同步执行的；因此你的文件系统驱动只需要为这种类型的请求设计一个同步处理 IRP 的方法。其他的操作可以同步或者异步的处理，因此你的驱动就必须提供同步和异步处理这样的 IRP 的路径。

内核驱动怎样确定一个 IRP 应该被同步处理还是异步处理呢？

在我们解决这个问题之前，来看看为什么正确地处理异步请求的重要性可能是有用的。考虑你设计的一个文件系统，不理会异步请求，所有的请求都被同步执行。你的实现在大多数时候都会正确的工作。一个可能发生的问题是当你的驱动收到异步分页 I/O 写请求的时候。这些请求通常是 NT 修改页写者发出的。对于修改页写者来说可用的工作者线程的数量是固定的。有可能 MPW 只使用两个线程来执行这些分页 I/O，一个负责页文件，另一个负责内存映射文件。

在底内存和高压力的情形下，VMM 会试图快速的刷新修改页面到辅助存储器上而为系统中

的其他数据腾出空间来。MPW 通过快速的向管理一个或者多个修改页面（不管是映射文件还是页文件）的文件系统发出异步写请求来完成这个工作。如果你的驱动阻塞 MPW 线程直到 I/O 完成，就会降低刷新数据到磁盘的整个进程，这可能导致用户不能接受的长时间的延迟。

因此如果你设计一个高层内核驱动，就应该慎重地提供异步 I/O 操作的支持。

只有一些 I/O 系统服务能够被异步的处理：

- 读请求
- 写请求
- 目录控制请求
- 字节范围的锁定/解锁请求
- 设备 I/O 控制请求
- 文件系统 I/O 控制请求

你可能已经注意到了，上面列出的所有请求类型潜在的都要耗费较长的时间才能完成。因此调用者可以要求异步的处理这些请求是合乎逻辑的。所有其他的 IRP 主功能应该尽快的完成。

因此如果你的文件系统或者高层过滤驱动（在文件系统上面）收到一个 IRP 的主功能不在这里列出的类型之中，你可以假设你被允许阻塞在调用线程的上下文中。

对这些列出的主功能，调用者可以选择是否指定请求应该被同步或者异步处理。要找出调用者希望的是怎样，你的驱动可以调用下面的 I/O 管理器支持例程：

```
BOOLEAN
IoIsOperationSynchronous (
    IN PIRP         Irp
);
```

参数：

Irp

发送给你驱动的 I/O 请求包。这个 I/O 请求包已经被 I/O 管理器设置标志来确定这个 IRP 是需要同步或者异步处理。注意异步操作总是可以被同步地执行（注意上面讨论中的警告）；但是，即使你的驱动异步地执行一个同步操作，返回一个 STATUS_PENDING 给 I/O 管理器，NT I/O 管理器也会在内核中为调用线程执行一个等待操作。

功能提供：

这个简单的函数可以执行下面的检查：

- 如果设置了 IRP_SYNCHRONOUS_IRP 标志，这个 IRP 就应该同步地执行。所有 IRP 结构描述的主要功能不在上面的列表中的都会设置这个标志。这个标准的存在将会导致 IoIsOperationSynchronous 返回 TRUE。
- 就象在这一章的前面描述的，调用者可能仅仅为了同步访问而打开了目标文件。这是通过在文件对象数据结构中设置 FO_SYNCHRONOUS_IO 标志来指示。这个标准的存在将会导致 IoIsOperationSynchronous 返回 TRUE。
- IRP 可能是一个分页 I/O 的读或者写请求，通过在 IRP 中设置 IRP_PAGING_IO 标志指示。此外，即使分页 I/O 请求可以是同步或者异步的。同步分页 I/O 请求通过在 IRP 中设置 IRP_SYNCHRONOUS_PAGING_IO 标志指出。如果后面一个标志没有设置，I/O 管理器就知道这是一个异步分页 I/O 请求而返回 FALSE；否则，I/O 管理器就标志这是一个同步分页 I/O 请求而返回 TRUE。

NT I/O 管理器提供各种方法来在异步 I/O 操作完成的时候通知调用者。这里是可能的方法：

- 在 WINDOWS NT 中文件对象结构是一个可等待的对象。当在一个文件对象上开始一个

I/O 操作的时候，这个文件对象首先被设置为无信号状态，当 I/O 操作完成的时候，文件对象被设置为有信号状态。

- I/O 管理器提供的异步的 NT 系统服务接受一个可选的事件对象，这个事件对象首先被设置为无信号状态，当 I/O 操作完成的时候，事件对象被设置为有信号状态。在讨论 IRP 的完成的时候，我提到 IRP 完成的时候执行的后处理的时候 I/O 管理器会在调用线程上下文中通知一个用户提供的事件对象。但是，注意，如果提供了这样一个事件对象，那么文件对象就不会被通知。
- I/O 管理器提供的异步的 NT 系统服务还接受一个可选的用户提供的 APC 例程，这个例程作为 I/O 管理器在调用线程上下文中执行后处理的一部分通过一个 APC 调用。

关于同步请求的最后一点是；使用同一个文件对象的所有同步请求将会被序列化，不管他们是同一个线程发出的还是相同进程中的其他线程发出的。文件系统驱动也有责任为每个文件对象维护一个当前指针，每当一个文件对象同步 I/O 打开的是更新这个指针。

操作用户空间缓冲区指针

当你创建一个设备对象来接受和处理 I/O 请求的时候，你就有机会来指定怎样操作用户提供的缓冲区地址指针了。直到你读到下一章的 NT 虚拟内存管理器之前你不会完全理解为什么这个信息是必要的。但是，现在要知道的是在 WINDOWS NT 中一个用户模式线程能够访问的地址范围限制是任何进程 4GB 地址空间中的底 2GB。此外，这个 2GB 虚拟地址空间范围对于每个进程是唯一的（也就是说，线程 A 和线程 B 使用的同一个地址不必指向相同的物理内存位置）。当然，属于同一进程的线程是共享相同的地址空间的。

一个用户模式的应用程序通常使用在自己线程上下文中分配的临时缓冲区来执行和辅助存储器的数据交互。现在我们忽略应用程序使用的其他代替的方法，包括使用共享内存或者内存映射文件。

例如，考虑一个应用程序需要从磁盘上一个文件中读一些数据。这个应用程序通常会分配一个足够容纳请求的数据的缓冲区。应用程序然后在他打开的希望得到数据的文件上调用一个读操作，指定读取位置的偏移量和要读取的信息数量。

这个应用程序发出的读请求最终会被 NT I/O 管理器转化为一个 NT 系统服务调用。I/O 管理器收到的参数之一是用户模式应用程序提供的缓冲区的指针。这个读请求现在被 I/O 管理器发送给管理被打开的文件对象所在的卷的文件系统驱动（注：在你读了本书的其他部分的时候，你会发现这里这个声明不完全正确，因为 I/O 管理器经常会完全绕过文件系统驱动而直接从系统缓存中取数据。现在这里我们为了简化和易于理解，将会忽略那种数据传输的方法）。在这时候 I/O 管理器会知道文件系统驱动会怎样处理用户提供的缓冲区指针。这个缓冲区只有在用户模式线程上下文中是有效的，而且并不是指向锁定（非分页）内存。文件系统可以选择下面可能的选项：

- I/O 管理器总是分配一个系统缓冲区，随后用于文件系统驱动数据传输。I/O 管理器在分派这个 IRP 给文件系统驱动之前要负责把要写出到磁盘的驱动从用户提供的缓冲区中复制到这个系统缓冲区中。类似的，对于用户要从文件系统驱动取数据或者从磁盘上读数据的 I/O 操作，I/O 管理器必须在 IRP 完成的时候从系统缓冲区复制数据到用户提供的缓冲区中。

  这种指示 I/O 管理器分配相应的系统缓冲区的处理用户模式缓冲区的方法叫在有缓冲的 I/O 的方法。

  系统缓冲区指针在 IRP 结构中的 AssociatedIrp.SystemBuffer 域中传给你的驱动。注意 I/O 管理器经常会在 IRP 的 UserBuffer 域中存放调用者提供的缓冲区的地址。但是不要试图在你的内核驱动中使用这个域的内容，因为在 SystemBuffer 域中已经包含了你可以使用

的系统缓冲区的指针。

使用有缓冲的 I/O 的缺点是 I/O 管理器需要执行额外的内存复制。这在你希望得到最大化系统性能的时候是不希望的。但是，有缓冲的 I/O 是最简单的因此是处理用户缓冲区中使用的最多的方法。

使用有缓冲的 I/O 的另一个缺点是 I/O 管理器分配的系统缓冲区内存的不分页的。这就导致给为系统保留的非分页池内存不必要的损耗。第三个问题是，虽然这个内存是不被分页的，但是如果你希望使用 DMA 来和外部设备直接写入/读出数据，你的驱动或者是低层驱动将不得不创建一个 MDL（内存描述符列表）来描述存储这个分配的缓冲区的物理页。

- 如果你的驱动希望避免分配和从系统缓冲区复制进出数据的负载，你可以使用指定你的驱动使用直接 I/O 的方法。如果指定这个方法，I/O 管理器会从 VMM 请求一个 MDL 直接来描述用户缓冲区，他还会请求 VMM 为这个用户缓冲区分配和锁定物理页。这个 MDL 指针将在 IRP 的 MdlAddress 域中传给你的驱动。

  直接 I/O 相对于分配一个额外的缓冲区然后执行必要的复制操作的方法是更有效率。也就是说，你的驱动在传输数据的时候没有虚拟地址指针可用。现在，当你简单地传递这个 MDL 给低层驱动的时候会良好的工作，这个 MDL 随后在 DMA 数据传输中被使用。但是，如果你需要一个在你处理 IRP 的线程上下文中可访问的虚拟地址指针，你的驱动不得不使用 VMM 的支持例程 MmGetSystemAddressForMdl。在使用这个例程的时候必须小心，释放 MDL 将导致系统中的所有处理器刷新他们的缓存。这样做的原因是复杂的；但是简单的说就是，得到一个 MDL 的系统虚拟地址是通过“双重映射”来做的，这也就是叫做“别名”的技术，如果没有正确地处理，会导致许多缓存一致性问题和使 VMM 头痛。如果你的驱动不使用直接 I/O 来处理用户提供的缓冲区，那么尽可能的避免使用 MmGetSystemAddressForMdl 例程。
- 第三个方法是不指定直接 I/O 或者有缓冲的 I/O 做为首选的处理用户缓冲区的方法。如果你没有指定这两个方式。I/O 管理器会简单地在 IRP 的 UserBuffer 域中传递用户地址给你的驱动。

  如果你选择这个方法，那么操作用户缓冲区的责任就就在你的驱动上。文件系统驱动常常使用这个方法，然后当处理请求的时候在他们的分派例程中决定是否是自己创建一个 MAL 或者系统缓冲区。但是，大多数的低层驱动更喜欢使用上面描述的直接 I/O 方式。

这些方式不适用于在设备或者文件系统IOCTL（I/O控制）请求中传递的缓冲区。我会在第三部分中讨论IOCTL（I/O控制）请求和I/O管理器操作这种请求的缓冲区。

系统启动顺序

在你继续阅读本书的其他章节之前，理解从你加电你的WINDOWS NT系统开始到你看见登陆屏幕之间要执行的步骤可能是有用的。

在你设计你的驱动的时候这些信息可以提供很多帮助，因为他决定你驱动将会在什么时候加载和在这个过程中会调用你的驱动的那些部分。但是，你也应该知道启动进程是高度依赖系统，处理器，操作系统版本和系统架构的，在下面列出的步骤中的基本的目的仅仅是给你提供在系统驱动时的“真实发生的”情况一个一般的信息，而不准备适用于一个新的处理器架构（我描述了在X86架构处理器上执行的顺序，尽管我上面警告了，在系统启动期间执行的许多代码是被设计为依赖于不同的架构的；因此，使用的方法和原理是很相似的）。因此要警告的是下面的描述是高度简化的，虽然绝大多数都正确。

解释系统启动顺序的一个主要问题是确定开始的地方。为了这一节的目的，我们的“开始点”将是 NT 操作系统的代码开始执行的地方：

1， NT 系统启动例程由系统启动模块调用。这个例程传递一个 BootRecord 结构，这个结构包含基本的机器和环境信息，随后将由 OS 加载组件使用。

NT 系统启动例程执行一些全局的初始化和确定磁盘驱动器和系统将从中启动的分区。全局初始化的一部分包括初始化在系统启动最初阶段使用的内存描述符。NT 系统启动例程还要启动加载器堆初始化例程，这个例程建立适当的内存描述符以便启动加载器在系统加载进程过程中能够使用。

迄今为止描述的是 NT 系统启动进程的 8 个阶段中的一个组成阶段。

2， 系统启动例程现在调用启动加载器启动例程。注意系统启动例程并不期望调用启动加载器启动例程会返回，因为那样会指示系统启动序列失败了。但是，如果这真的发生了，你可能会看见一个挂起的系统，这时候可能需要强行进行电源重启动。

启动加载器启动例程打开启动分区，这个分区先前已经被他的调用者确定了。读取文件 boot.ini。作为试图读取这个文件的一部分，启动加载器启动例程要使用已经便宜进来的代码确定启动分区包含的是一个 NTFS，CDFS 或者 HPFS 分区。注意这时候标准文件系统驱动还没有被加载，启动加载器启动例程硬编码来支持这些 MICROSOFT 选择提供启动支持的文件系统。这些是发生在标准 NT 文件系统实现中的。因为支持启动文件系统必须内建在 NT 启动加载器启动代码中，所以没有 MICROSOFT 的积极协助要提供一个第三方可启动文件系统实现是几乎不可能的。

在这时候，启动加载器启动例程调用一个中断调用来设置视频适配器为 80*50，16 色的字符模式。他还通过输出空格到屏幕来清除显示。

启动加载器启动例程读取整个 boot.ini 文件的内容，显示在 boot.ini 文件中的可用启动内核列表给用户。为了读这个文件，启动加载器启动例程再次使用能够识别 NTFS，CDFS，FAT 和 HPFS 数据结构，而且能够成功浏览整个磁盘上文件系统布局的例程。如果文件 boot.ini 是空的，默认选项是 NT，默认的启动目录路径是 C:\winnt（注：注意当前的启动加载器启动例程有个 BUG 就是不能处理 boot.ini 文件中超过 10 个项的情况。所有超过这个限制的项会被简单地丢弃，显然，这个 BUG 从 NT3.5 就已经存在了（很可能更早））。

默认启动位置和可能的选择被启动加载器启动例程用视频显示支持例程显示给用户。如果用户选择导入内核路径位置是 C:\，NT 加载启动代码假设用户希望启动到 DOS，WINDOWS 3.X，WINDOWS 95 或者 OS2；因此，他试图读取文件 bootsect.dos，然后重新启动系统到出现的无论哪个选择的操作系统中。

如果启动位置指示出用户希望启动到 WINDOWS NT（这可能在选择超时或者用户选择了特定的启动系统的时候发生），启动加载器启动例程从启动分区的启动目录试图读取 ntdetect.com 可执行文件。如果 ntdetect.com 找不到或者这个文件的大小看起来不正确，或者任何 OS 加载器启动代码进行的一致性检查失败，启动进程会失败，你必须重启动系统。但是，如果找到一个有效的可执行体，他就会被读进内存，系统试图使用硬件制造商提供的服务来探测当前硬件配置。

注意在这时候我们将进入系统启动初始化的第二阶段。如果需要，OS 加载器启动例程现在初始化 SCSI 启动驱动器。ntldr.exe 操作系统加载器现在被加载到内存中。

3， OS 加载器打开控制台输入输出设备，还有系统和启动分区。他还在控制台上显示 OS 加载器的标志器，OS Loader V4.0。

加载器用启动分区信息来为 NT 核心系统印象文件 ntoskrnl.exe 产生一个完全路径名。注意系统总是期望在启动分区的 System32 目录里找到这个文件。一旦系统印象被加载到内存中，OS 加载器加载系统文件 hal.dll 到内存中，HAL（硬件抽象层）为 NT 执行体的其他组件提供平台依赖的隔离。

在这时候，这两个加载的系统文件的所有引入 DLL 被定位出来并加载到内存中。现在 OS 加载器试图加载 NT 注册表的 SYSTEM 子键下的内容。这时候，加载已经作出决定是加载注册表中的 LastKnownGood 控制集还是 Default 控制集。这个决定很重要，因为控制集决定了将被加载到系统中的启动驱动。为了加载 SYSTEM 子键到内存中，OS 加载器试图从启动分区的 System32\config 目录打开和读取 SYSTEM 文件。如果打开和读取文件 SYSTEM 文件失败，就试图读取 SYSTEM. ALT 文件。如果读取都不成功，OS 加载器将使启动失败。如果文件能够成功读取，将验证文件的内容，然后内存中的数据结构将被以这个磁盘上文件的内容来初始化。还有，注意系统加载器块被适当的修改为指向在内存中的 SYSTEM 子键，最后被传递给加载的系统印象。

此时，OS加载器决定要加载到系统内存中的启动驱动程序的列表。包括在这个列表中之一是负责启动分区的文件系统驱动程序。注意启动驱动程序用加载到内存中的控制集中的驱动程序的键的Start值项（应该等于0）来标志。一旦启动驱动的列表被标志出来，OS加载器根据注册表中指定的ServiceGroupOrder来排序这些驱动。同一个组中的驱动基于在注册表中指定的GroupOrderList和与注册表中每个驱动程序键关联的标签（Tag）值项来排序。

当驱动加载顺序被确定了以后，所有的启动驱动都被加载。在加载驱动错误的情况下，就检查注册表中与这个驱动关联的 ErrorControl 值项，如果这个驱动被标记为系统启动进程中的临界驱动，当前启动将失败；否则，OS 加载器继续加载其他启动驱动程序。

最后，OS 加载器准备执行加载的系统印象，把控制传递给 NT 核心的入口。

4， 在系统启动的第 3-5 阶段，各种 NT 执行体组件和 NT 内核被初始化。那些 Start 值项为 1 的驱动也在系统启动进程的阶段 5 被自动加载。

NT 内核初始化例程 KilnitializeKernel 在初始化阶段 3 中被内核系统启动例程（他是 OS 加载器加载到内存中的系统印象的入口）调用。这个例程初始化处理器控制块数据结构，内核数据结构，还有空转线程和进程对象，调用 NT 执行体初始化例程。各种保护内核数据结构的自旋锁和内核链表结构在这里初始化。各种内核链表头（DPC 队列链表头，定时器通知链表头，各种线程表链表，还有其他类似的内核数据结构）也在这里初始化。

当内核空转线程数据结构初始化以后，执行体初始化例程现在就在这个空转线程上下文中调用。初始化 NT 执行体和各种 NT 执行体的子组件在两个阶段中发生（注：不要把这些阶段和系统启动顺序的 8 个阶段混淆，这两个阶段是特定于初始化 NT 执行体和他的各种子组件的内部的阶段），在执行体初始化阶段 0 中，下列的子组件初始化他们自己的内部状态：

- 硬件抽象层（HAL）
- NT 执行体组件
- 虚拟内存管理器（VMM）

  内存管理器分页和非分页池，页帧数据库（在下一章中解释），页表项（PTE）管理结构，和各种 VMM 资源，例如互斥体和自旋锁数据结构，在这个时候初始化。VMM 在此时还初始化 NT 系统缓存相关数据结构，包括系统缓存工作集和各种用于管理系统缓存的 VMM 数据结构。
- NT 对象管理器
- 安全子系统
- 进程管理器

  在 NT 执行体初始化的阶段 0 中，初始系统进程被创建了。注意空转进程是 NT 内核在任何其他的执行体初始化开始之前手工创建的。这里创建的系统进程和前面创建的空转进程是有区别的。此时一个系统线程也被创建在初始进程

的上下文中。在阶段 1 或者剩下的 NT 执行体的初始化，现在就执行在这个属于初始进程的新创建的线程上下文中执行。

在 NT 执行体和各种子组件初始化的阶段 1 中，所有中断被禁止，初始化执行的线程上下文的线程优先级被提升到高优先级，有效禁止任何抢占。还有，在执行体初始化阶段 1 中，系统被认为是健全的，子组件被允许执行任何要求的操作来完成他们的初始化。在执行体初始化阶段 1 中，下面的子组件将被调用（或者他们的操作被执行）：

— 硬件抽象层（HAL）被调用来完成初始化
— 系统日期和时间被初始化
— 在多处理器系统中，其他的处理器此时被启动
— 对象管理器，执行子系统，还有安全子系统被调用来执行他们剩下的初始化
— 虚拟内存管理器（VMM）的初始化阶段 1 执行

此时，内存映射功能被初始化，对于系统的其他足见可用。VMM 线程可在现在开始。VMM 此时可以被认为是完全的而准备为在初始化阶段 1 后为其他的系统服务。

— 在 VMM 初始化完成以后 NT 缓存管理器被初始化

你会在本书的后面读到更多关于 NT 缓存管理器和他提供的功能。现在要知道的是，在缓存管理器初始化中，为异步操作而要求的多个工作者线程被确定和创建，还有缓存管理器链接的链表结构和同步资源也被初始化。

— 配置管理器被调用开始他的初始化

配置管理器管理 NT 注册表。在初始化的这个阶段，配置管理器（CM）是注册表中的\REGISTRY\MACHINE\SYSTEM 和\REGISTRY\MACHINE\HARDWARE 子键可用。为了这个，ntdetect.com 在先前得到的所有信息，还有 OS 加载器读进内存的信息被填充进 SYSTEM 和 HARDWARE 子键的适当的项中。一旦这个阶段的初始化完成，部分注册表名字空间对于其他的系统组件就可用了，特定的内核驱动将很快被加载。但是，CM 在这时候不会把注册表的修改写到注册表中。内核驱动将很快被调用，通过执行驱动特定的初始化，可以使用标准的注册表例程来访问这个信息。

— 调用 NT I/O 管理器来执行他的初始化

I/O 管理器首先初始化内部状态对象，包括同步数据结构，链表和内存池（比如 IRP 池或者旁视列表）。接着，I/O 管理器使用一个叫做 ObCreateObjectType 的内部例程向对象管理器注册他的内部定义的对象类型（也就是说，适配器对象，控制器对象，设备对象，驱动对象，I/O 完成对象和文件对象）（注：记住对象管理器是不知道这些对象的类型的（也就是说，关于 I/O 管理器定义的对象信息是不会编码到对象管理器的设计中的），这也说明了一个分层的，基于对象的系统，NT 开发组所遵循的）。此时 NT I/O 管理器还要在对象名字空间中创建\Device, \DosDevices,和\DriverRoot 目录。

接着，OS 加载器所加载的启动驱动由 I/O 管理器来初始化。这包括为每个驱动程序调用驱动入口例程来执行驱动程序特定的初始化。“原始文件系统（raw file system）”驱动也在此时加载。其他唯一要加载的文件系统驱动是“启动文件系统（boot file system）”驱动。驱动必须坚持限制的和 NT 注册表的交互。最后，那些 Start 值项为 1 的驱动被加载，他们的驱动入口例程被调用执行驱动特定的初始化。

接着为那些要求重新初始化的驱动调用重新初始化例程。接下来，NT I/O 管

理器为识别出的磁盘分区赋上驱动器盘符。盘符 A: 和 B:是为软盘驱动器保留的。I/O 管理器检查注册表，寻找那些需要为 CD-ROM 和硬盘分区保留的“固有的（sticky）”驱动器盘符分配。这些驱动器盘符被内部保留因此不能用在接下来的动态确定 DOS 驱动器盘符分配中。

注意在为每个硬盘驱动器保留驱动器盘符之前，I/O 管理器在物理驱动器上执行一个打开操作。因此，如果你开发硬盘设备驱动或者低层过滤驱动或者是中间层驱动，你应该在这时候期待一个打开请求。如果这个打开请求成功了，就在 NT 对象名字空间中创建这个设备对象的符号连接；指派给这个符号连接的名字是这样的格式：\DosDevices\PhysicalDrive%d，这里的%d 表示硬盘驱动器的序号。NT I/O 管理器这时候还要从这个硬盘驱动中查询分区信息。下面是 I/O 管理器用来确定给固定磁盘分区指派的驱动器盘符顺序的方法：

— NT I/O 管理器在注册表中查询保留的“固有的（sticky）”驱动器盘符。
— 可启动分区被首先赋值为 DOS 兼容的驱动器盘符（也即是说，为设备对象创建连接符号来表示这个分区，使用\DosDevices\%c:形式的名字，这里%c:表示 NT I/O 管理器为这个分区选择的驱动器盘符）。
— 接着选择主要分区来动态地指派驱动器盘符。
— 接着为扩展分区指派 DOS 驱动器盘符
— 现在为其他（增强）分区分配 DOS 驱动器盘符。在为硬盘驱动器和可移动磁盘驱动器分区分配了驱动器盘符之后，NT I/O 管理器为在硬件检测期间探测到的所有 CD-ROM 分配驱动器盘符。
— 本地过程调用（LPC）子系统和进程管理器子系统现在完成他们的初始化
— 接着是调用引用监视器和会话管理器子系统来完成他们的初始化。

5， 在这时候，系统启动进程的 3-5 阶段已经完成。记得 NT 执行体组件是在属于系统进程的系统工作者线程上下文中初始化的，系统工作者线程是由 NT 内核创建的。这些线程现在担当内存管理器的“页清零线程”的角色，这是一个非常底优先级的用来异步清零那些被 VMM 放入到自由列表中的页面的线程。正如你在下一章中将读到的，为了使系统符合美国国防部（DOD）定义的 C2 安全等级，所有页面在重新使用之前必须要清零。

6， 此时系统已经被初始化了。在系统启动进程的 6-8 阶段，各种子系统被初始化，其他服务被服务管理器加载。这包括加载注册表中 Start 值为 2 的内核驱动程序。

我们结束了我们的 NT 系统启动进程的“鸟览”，这应该能够给你一个对从系统驱动开始到 NT 系统达到一个稳定状态而能够响应用户请求需要执行的步骤有相当的理解。

本章给你介绍了 WINDOWS NT I/O 管理器和 WINDOWS NT I/O 子系统。NT 执行体的另一个重要组件是 NT 虚拟内存管理器（VMM），这将是下一章的主题。