In this chapter:
- ***The NT I/O Subsystem***
- ***Common Data Structures***
- ***I/O Requests: A Discussion***
- ***System Boot Sequence***

4

The NT I/O Manager

Successfully interfacing with external devices is essential for any computing system. A general-purpose commercial operating system like Windows NT must also interact with a variety of peripherals, the common ones most of us use each day, as well as the more uncommon external devices that might be useful in some specific settings. For example, we expect the NT operating system to provide us with built-in support for our hard disks, keyboard, mouse, and video monitor. If, however, I wish to attach a programmable toaster device to my system (my new invention), and I would like to control this device using my computer, which is running Windows NT, I suspect that I will have to develop a driver to control the device. Furthermore, if I expect to be successful in developing this driver, I will obviously have to look to the operating system to provide an appropriate environment and support structure that makes developing, installing, testing, and using this driver a task that might be difficult but not insurmountable.

Although some might argue that such expectations of support from an operating system are unreasonable, the Windows NT operating system does provide such a framework, so that mere mortals like you and me can develop necessary drivers to control such esoteric devices as a programmable toaster. In fact, the NT operating system provides a consistent, well-defined I/O subsystem within which all code required to interface with external devices can reside. The I/O subsystem is extensive, encompassing file system drivers, intermediate drivers, device drivers, and services to support and interface with such drivers. It is also consistent in its treatment of external devices.

In this chapter, I will present an introduction to the NT I/O Manager, the component responsible for creating, maintaining, and managing the NT I/O subsystem. To develop any kind of driver for the Windows NT operating system, an understanding of the framework provided by the I/O Manager is extremely important. First, I will describe some of the services provided by the I/O Manager. Next, I will present an overview of the components comprising the I/O subsystem, including a discussion of the various types of drivers that can exist within the I/O subsystem. I will then describe some common data structures that kernel-mode developers should be familiar with. Following this is a discussion on some common issues involving I/O requests sent to kernel-mode drivers. Finally, I will present a description of the system boot sequence, with emphasis on the activities of the I/O Manager and the drivers within the kernel.

The NT I/O Subsystem

The NT I/O subsystem is the framework within which all kernel-mode drivers controlling and interfacing with peripheral devices reside. This subsystem is composed of the following components (see Figure 4-1):

- The NT I/O Manager, which defines and manages the entire framework.
- File system drivers that are responsible for local, disk-based file systems.
- Network redirectors that accept I/O requests and issue them over the network to a file server. The redirectors are implemented similarly to other file system drivers.
- Network file servers that accept requests sent to them by redirectors on other nodes, and reissue these requests to local file system drivers. Although file servers do not need to be implemented as kernel-mode drivers, typically they are implemented as such for performance reasons.
- Intermediate drivers, such as SCSI class drivers. These drivers provide generic functionality that is common to a set of devices. Intermediate drivers also include drivers that provide added functionality, such as software mirroring or fault tolerance, by using the services of device drivers.
- Device drivers that interface directly with hardware, such as controller cards, network interface cards, and disk drives. These are typically the lowest-level kernel-mode drivers.
- Filter drivers that insert themselves into the driver hierarchy to perform functionality that is not directly available using the existing set of drivers. For example, a filter driver can layer itself above a file system driver, intercepting all requests that are issued to the file system driver. A filter driver could just as well layer itself below the file system driver, but above a device driver, intercepting all requests targeted to the device driver. Note that conceptually, the only tangible difference between filter drivers and other intermediate drivers is that filter drivers typically intercept requests targeted to some existing

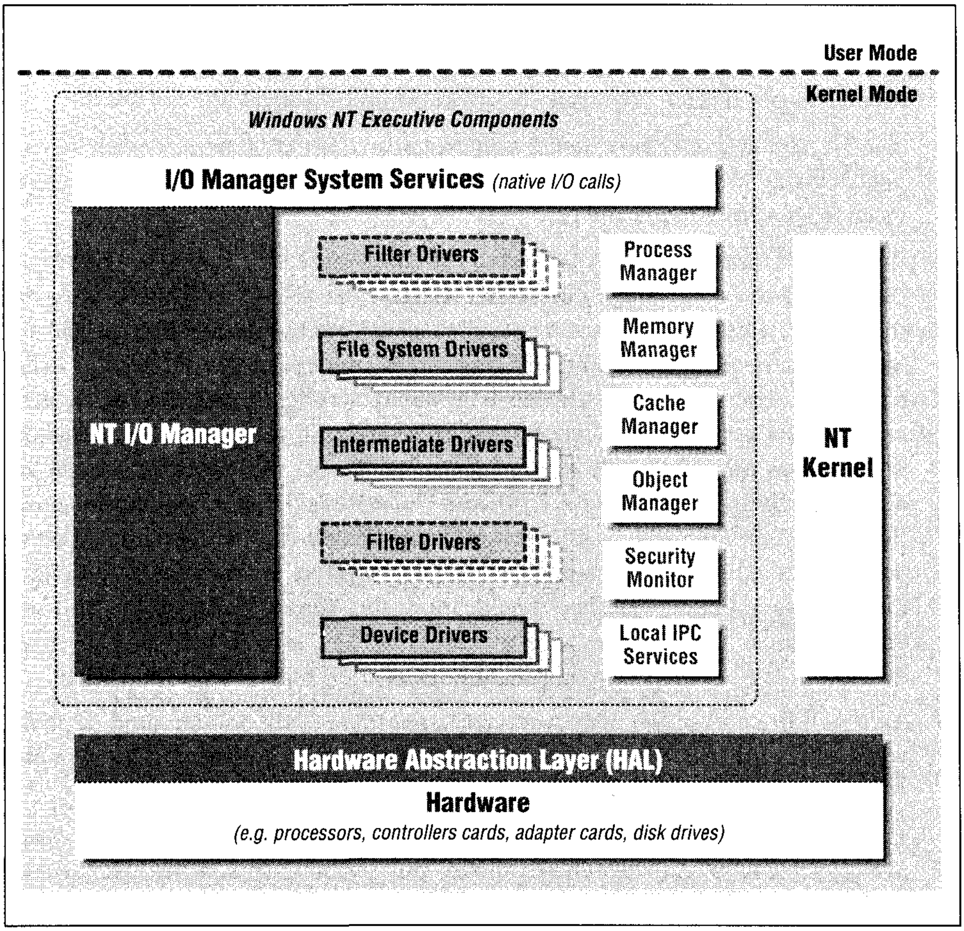

Figure 4-1. Kernel-mode components, including the I/O subsystem

device and then provide their own functionality, either in lieu of or in addition to the functionality provided by the driver that was the original recipient of the request.

Functionality Provided by the NT I/O Manager

The NT I/O Manager oversees the NT I/O subsystem. The following is a list of some of the functionality provided by the I/O Manager:

- The I/O Manager defines and supports a framework that allows the operating system to use peripherals connected to the system.

  The type and number of peripherals that can potentially be used with a Windows NT system is not limited, since new types of peripheral devices are constanding of the framework provided by the I/O Manager is extremely important. First, I will describe some of the services provided by the I/O Manager. Next, I will present an overview of the components comprising the I/O subsystem, including a discussion of the various types of drivers that can exist within the I/O subsystem. I will then describe some common data structures that kernel-mode developers should be familiar with. Following this is a discussion on some common issues involving I/O requests sent to kernel-mode drivers. Finally, I will present a description of the system boot sequence, with emphasis on the activities of the I/O Manager and the drivers within the kernel.

The NT I/O Subsystem

The NT I/O subsystem is the framework within which all kernel-mode drivers controlling and interfacing with peripheral devices reside. This subsystem is composed of the following components (see Figure 4-1):

- The NT I/O Manager, which defines and manages the entire framework.
- File system drivers that are responsible for local, disk-based file systems.
- Network redirectors that accept I/O requests and issue them over the network to a file server. The redirectors are implemented similarly to other file system drivers.
- Network file servers that accept requests sent to them by redirectors on other nodes, and reissue these requests to local file system drivers. Although file servers do not need to be implemented as kernel-mode drivers, typically they are implemented as such for performance reasons.
- Intermediate drivers, such as SCSI class drivers. These drivers provide generic functionality that is common to a set of devices. Intermediate drivers also include drivers that provide added functionality, such as software mirroring or fault tolerance, by using the services of device drivers.
- Device drivers that interface directly with hardware, such as controller cards, network interface cards, and disk drives. These are typically the lowest-level kernel-mode drivers.
- Filter drivers that insert themselves into the driver hierarchy to perform functionality that is not directly available using the existing set of drivers. For example, a filter driver can layer itself above a file system driver, intercepting all requests that are issued to the file system driver. A filter driver could just as well layer itself below the file system driver, but above a device driver, intercepting all requests targeted to the device driver. Note that conceptually, the only tangible difference between filter drivers and other intermediate drivers is that filter drivers typically intercept requests targeted to some existing

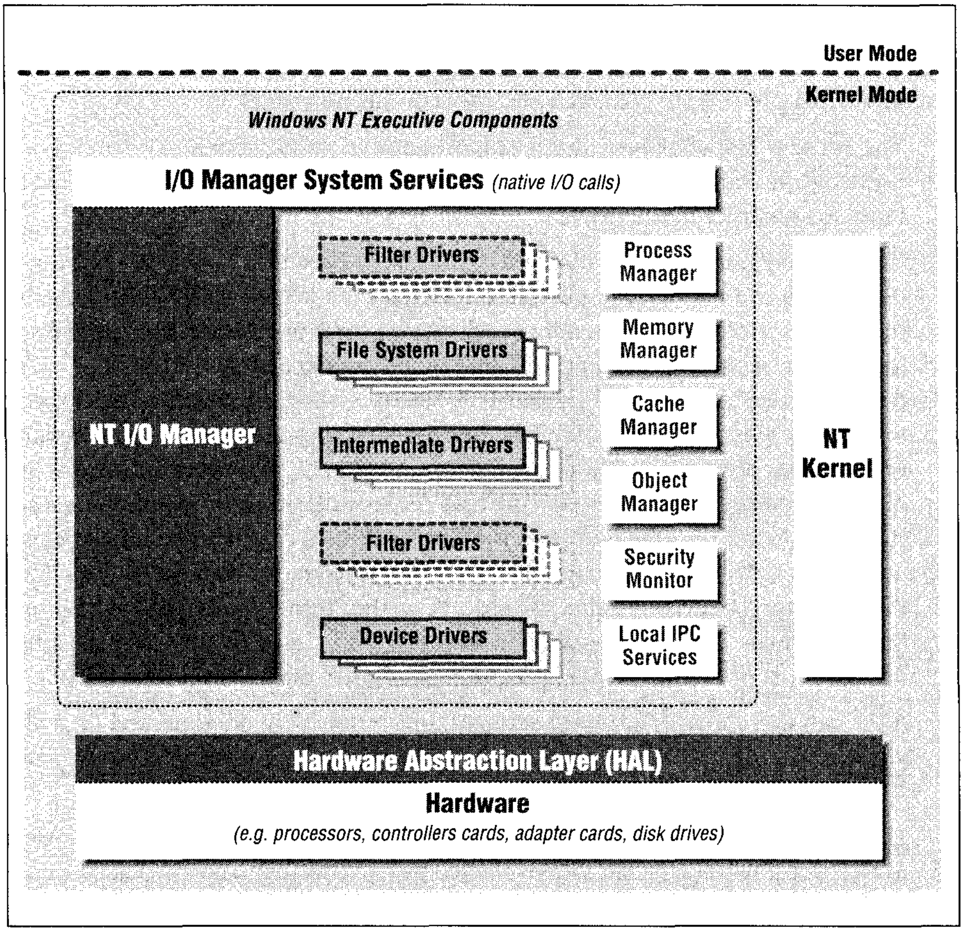

Figure 4-1. Kernel-mode components, including the I/O subsystem

device and then provide their own functionality, either in lieu of or in addition to the functionality provided by the driver that was the original recipient of the request.

Functionality Provided by the NT I/O Manager

The NT I/O Manager oversees the NT I/O subsystem. The following is a list of some of the functionality provided by the I/O Manager:

- The I/O Manager defines and supports a framework that allows the operating system to use peripherals connected to the system.

  The type and number of peripherals that can potentially be used with a Windows NT system is not limited, since new types of peripheral devices are continuously being designed and developed. Therefore, the I/O subsystem for a commercial operating system like Windows NT must be well-designed and extensible, such that it can easily accommodate the myriad devices, each with its own set of unique characteristics, that could be used.

- The NT I/O Manager provides a comprehensive set of generic system services used by the various subsystems to actually perform I/O or request other services from kernel-mode drivers.

  Consider a read request initiated by a user process. This read request is directed to the controlling subsystem, such as the Win32 subsystem. Note that the Win32 subsystem does not actually direct the read request to the file system driver or device driver itself; instead it invokes a system service called NtReadFileO , supplied by the I/O Manager. The NtReadFileO system service then assumes the responsibility for directing the request to the appropriate driver and conveying the results to the Win32 subsystem. Also note that the buffer supplied by the user process requesting the read operation usually cannot be used directly by the kernel-mode drivers that will eventually satisfy the request. The I/O Manager provides the support to automatically perform the necessary operations that would allow the kernel-mode drivers to use a buffer address that is accessible in kernel-mode. Later in this chapter, I will describe this operation of manipulating user-mode buffers in further detail.

  Although the native NT system services are very poorly documented (if at all), you can find a detailed description of these services in Appendix A, *Windows NTSystem Services,* in this book.

- The NT I/O Manager defines a single I/O model that all drivers in the system must conform to. As mentioned above, this model consists of objects and a set of associated methods used to manipulate the objects. Kernel-mode drivers do not need to be concerned with the originator of an I/O request, since they respond to all I/O requests in the same manner.

  This results in a consistent interface provided to users of the I/O subsystem, such as the Win32 or POSIX subsystem, and also protects the kernel-mode drivers from having to worry about the vagaries associated with the particular subsystem that issued the I/O request.

  Furthermore, since every kernel-mode driver must conform to this single I/O model, kernel-mode drivers can use services provided by each other, since a kernel-mode driver does not really care whether the I/O request originates in kernel-mode or user-mode. That said, if you do invoke the services of another kernel-mode driver from your kernel-mode driver, there are certain considerations that you must be aware of. These will be described later in this chapter.

Finally, the single I/O model allows for the implementation of layered kernel-mode drivers, which are supported by the NT I/O Manager. Each kernel-mode driver in a layered hierarchy can utilize the services of the underlying driver to complete a specific operation. In turn, the underlying driver can satisfy the issued request without concerning itself with whether the request came to it directly from some user process or from a driver that resides above it in the hierarchy of layered drivers.

- The I/O Manager supports installable file system implementations that use the peripheral devices connected to the system.

  The NT operating system includes support for the CD-ROM file system, the NTFS log-based file system, the legacy FAT file system, the LAN Manager File System Redirector, as well as the HPFS file system. In addition to supporting such native local- and network-based file systems, the I/O Manager provides the infrastructure for development of external, installable file systems, i.e., file system implementations from third-party vendors. You can purchase commercial implementations of NFS (the Network File System), DPS (the Distributed File System), and other file system and network redirector implementations.

- The NT I/O Manager supports dynamically loadable kernel-mode drivers.
- The I/O Manager provides support for device-independent services that can be utilized by other components of the NT operating system, as well as by kernel-mode drivers that are implemented by third-party vendors.

  If a kernel-mode driver needs to invoke the dispatch routine for another kernel-mode driver, it can use the `loCallDriver()` service provided by the I/O Manager. Similarly, if a kernel-mode driver has to allocate a Memory Descriptor List (MDL) structure, the `loAllocateMdl()` routine, can be used. There are other such services that are commonly used by kernel-mode components (including kernel-mode drivers), provided by the NT I/O Manager. The list of services is available in the Windows NT Device Drivers Kit (DDK).

- The NT I/O Manager interacts with the NT Cache Manager to support virtual block caching of file data.

  Later in this book, you will learn more about the functionality provided by the NT Cache Manager.

- The NT I/O Manager interacts with the NT Virtual Memory Manager and file system implementations to support memory-mapped files.

  In the next chapter, you will read in detail about memory-mapped files. Support for memory-mapped files is provided jointly by the NT I/O Manager, the NT Virtual Memory Manager, and the appropriate file system driver.

If you wish to develop kernel-mode drivers for Windows NT, your driver must conform to the specifications provided by the NT I/O Manager. This includes creating and maintaining some data structures defined by the I/O Manager and also supplying the methods that manipulate such objects. Furthermore, your driver must respond appropriately to requests issued by the NT I/O Manager, and your driver must return results of each operation back to the I/O Manager. It is extremely unlikely that you can successfully develop a kernel-mode driver that does not use any of the services provided by the NT I/O Manager. Therefore, you will need to understand well the framework provided by the NT I/O Manager. The remainder of this chapter addresses some of these issues in further detail.

Concepts in I/O Manager Design

The design of the NT I/O subsystem exhibits a number of characteristics described in the following sections.

Packet-based I/O

The I/O subsystem is *packet-based;* i.e., all I/O requests are submitted using I/O Request Packets (IRPs). IRPs are typically constructed by the I/O Manager in response to user requests and sent to the targeted kernel-mode driver. However, any kernel-mode component can create an IRP and issue it to a kernel-mode driver using the IoAllocateIrp() and IoCallDriver() I/O Manager routines described in the DDK.

The I/O Request Packet is the only method you can use to request services from an I/O subsystem driver. By strictly conforming to this packet-based I/O model, the NT I/O Manager ensures consistency across the I/O subsystem and enables the layered driver model, described later in this section.

Each IRP sent to a kernel-mode driver represents a pending I/O request to that driver. An IRP will continue to be outstanding until the recipient of the IRP invokes the IoCompleteRequest() service routine for that particular IRP. Invoking IoCompleteRequest() results in that I/O operation being marked as completed, and the I/O Manager then triggers any post-completion processing that was awaiting completion of the I/O request. A particular IRP can be completed only once; i.e., only one kernel-mode driver can invoke IoCompleteRequest() for any outstanding IRP in the system.

You should be aware that, although packet-based I/O is the rule in Windows NT, the NT I/O Manager, NT Cache Manager, and the various NT file system implementations collaborate to implement functionality called the *fast I/O path,* which is an exception to this rule. The fast I/O method of I/O operations is only valid for file system drivers. These operations are implemented using direct function calls into the file system drivers and the NT Cache Manager instead of using the normal IRP method. The fast I/O path is described in detail later in this book.

NT object model

The I/O Manager conforms to the NT Object Model defined and implemented by the Object Manager component of the NT Executive.

Kernel-mode drivers, peripheral devices, controller cards, adapter cards, interrupts, and instances of open files are all represented in memory as *objects* that can be manipulated. These objects also have a set of *methods,* a set of operations that can be performed on the object, associated with them. For example, each controller card in the system is represented by a *controller object,* while each instance of an open file is represented by the *file object* data structure. The controller object can only be accessed using one of the methods associated with the object. This same restriction also applies to the file object structure, as well as to all other object types defined by the I/O Manager.

Note that kernel-mode drivers developed for Windows NT have to conform to this object-based model along with the rest of the I/O subsystem. All drivers must initialize a driver object structure representing the loaded instance of the device driver itself. In addition, if the driver manages devices or peripherals attached to the system, it must create and initialize one or more device object structures.

Since the I/O Manager uses the NT object model, it can also use the services of the Security Subsystem to control access to objects. The I/O Manager supports named object structures. For example, file objects have a name associated with them indicating the on-disk file that they represent. You can also create other named objects, such as device objects, that can then be opened by other processes or kernel-mode drivers.

Layered drivers

The I/O Manager supports layered kernel-mode drivers. Each driver in the hierarchy accepts an *I/O Request Packet,* processes it, and then invokes the next driver in the hierarchy.

Drivers lower in the hierarchy are closer to the actual hardware. However, only the lowest drivers typically interact directly with hardware devices or cards. The layered driver model is a boon to designers who wish to provide value-added functionality not supplied with the base operating system. This feature enables intermediate and filter drivers to be inserted into the driver hierarchy whenever required, and therefore allows new functionality to be easily added to the system. Furthermore, since each driver in the hierarchy interacts with drivers above and below it in a consistent fashion, development, debugging, and maintenance of kernel-mode drivers is a lot easier than on most other operating system implementations.

Asynchronous I/O

The NT I/O Manager supports *asynchronous I/O** allowing a thread to request I/O operations and continue performing other computational tasks until the previously requested I/O operations have been completed. This makes for greater parallelism in completing computational tasks as opposed to the purely sequential model in which a thread must wait for an I/O operation to proceed before it proceeds with other activity.

Figure 4-2 graphically illustrates the sequence of activities that occur when performing synchronous and asynchronous I/O operations. As you can see from the illustration, the thread using asynchronous I/O can continue performing computational activity in parallel with the servicing of the I/O request that it has initiated. This results in higher performance and higher net throughput for the system. Note that the default I/O mechanism is the synchronous model.

Preemptible and interruptible

The I/O subsystem is preemptible and interruptible. It is extremely important for all kernel-mode driver developers to understand these two concepts.

Every thread executing in kernel mode executes at a certain system-defined Interrupt Request Level (IRQL). Each IRQL has an interrupt vector assigned to it by the system, and there are a total of 32 different IRQLs defined by Windows NT. Any thread can have its execution interrupted due to an interrupt at a higher IRQL than the IRQL at which that thread is executing. When such an interrupt occurs, the Interrupt Service Routines (ISRs) associated with that particular interrupt are executed in the context of the currently executing thread. This results in a suspension of the current flow of execution so that thread can execute the ISR code.t

IRQ levels range from PASSIVE_LEVEL (defined as numeric value 0), which is the default level at which all user threads and system worker threads execute, to IRQL HIGH_LEVEL (defined as numeric value 31), which is the highest possible hardware IRQL in the system. Most file system dispatch routines are executed at IRQL PASSIVE_LEVEL. However, most lower-level device driver routines (for example, SCSI class driver read/write dispatch entry points) are executed at higher IRQ levels — typically at IRQL DISPATCH_LEVEL (defined as numeric value 2).

---

* The term *overlapped I/O* used by the Win32 subsystem refers to the same concept as that of asynchronous I/O supported by the NT I/O Manager.

t ISR execution can be interrupted as well if another, even higher-level interrupt occurs.

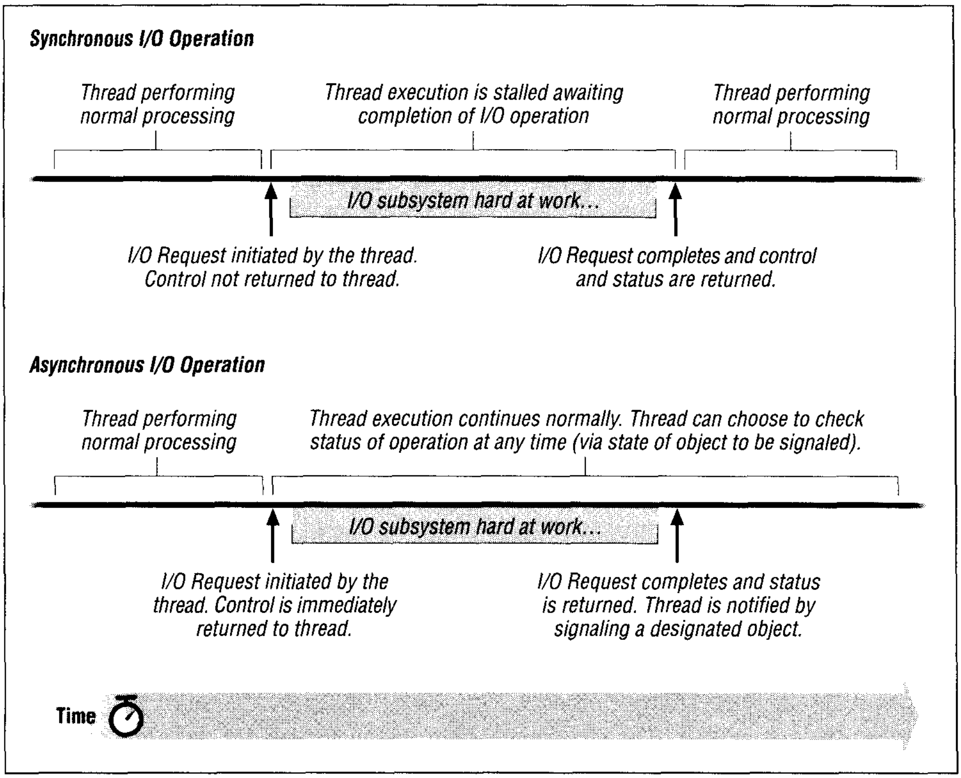
Synchronous I/O Operation
Thread performing normal processing
Thread execution is stalled awaiting completion of I/O operation
Thread performing normal processing
I/O subsystem hard at work...
I/O Request initiated by the thread. Control not returned to thread.
I/O Request completes and control and status are returned.
Asynchronous I/O Operation
Thread performing normal processing
Thread execution continues normally. Thread can choose to check status of operation at any time (via state of object to be signaled).
I/O subsystem hard at work...
I/O Request initiated by the thread. Control is immediately returned to thread.
I/O Request completes and status is returned. Thread is notified by signaling a designated object.
Time

Figure 4-2. Synchronous/asynchronous processing

Since all code in the I/O subsystem is interruptible, drivers developed for the NT operating system must use appropriate synchronization and protection mechanisms to prevent data corruption for data accessed at different IRQ levels. For example, if your kernel-mode driver accesses a data structure at IRQL PASSIVE_LEVEL in the context of a system worker thread, and if this driver also needs to access this same data structure at IRQL DISPATCH_LEVEL when servicing an interrupt request, the driver will have to use a spin lock that is always acquired at IRQL DISPATCH_LEVEL, which is the highest-level IRQL at which the spin lock could possibly be acquired, to provide mutually exclusive access to the data structure.*

Threads executing I/O subsystem code in the kernel are also *preemptible.* The Windows NT operating system associates execution priorities with threads. These priorities are typically variable, and most user-level threads and system worker

---

* Chapter 3, *Structured Driver Development,* provides a description of the available locking and synchronization primitives in the Windows NT kernel environment.

threads execute at relatively lower priorities, which allow them to *be* preempted by the NT scheduling code (in the NT Kernel) when a higher-priority thread is scheduled to run.

The fact that such threads could be preempted while executing kernel-mode code also necessitates synchronization mechanisms to ensure data consistency. This requirement is not present in other operating systems, such as the Windows 3.1 operating environment, or some versions of UNIX (e.g., HPUX, or SunOS), which currently do not allow preemption of threads or processes executing in kernel mode.

Kernel-mode driver designers must be extremely careful when acquiring common resources (e.g., read/write locks, semaphores) from within the context of different threads, because the Windows NT Kernel does not provide any built-in safeguards against programming errors resulting in situations like the priority inversion scenario described in Chapter 1, *Windows NT System Components.*

If you develop a driver that needs to acquire more than one synchronization resource at an IRQL that is less than or equal to DISPATCH_LEVEL, you must also be careful to define a strict locking hierarchy. For example, assume that your kernel-mode driver has to lock two FAST_MUTEX objects, *fast_mutex_l* and *fast_mutex_2.* You must define the order in which all threads in your driver can acquire both of these mutex objects. This order could be "acquire *fast_mutex_l* followed by *fast_mutex_2* or vice-versa. The reason for strictly defining and maintaining a locking hierarchy is to avoid a situation like one where *thread-a* acquired *fast_mutex_l,* wants to acquire *fast_mutex_2,* and gets preempted. *Thread-b* in the meantime gets scheduled to execute, acquires *fast_mutex_2,* and now needs to acquire *fast_mutex_l.* This scenario would cause a deadlock condition.

Portable and hardware independent

The I/O subsystem is portable and hardware independent. Kernel-mode drivers developed for Windows NT environments are also required to be portable and hardware independent.

The NT Hardware Abstraction Layer (HAL) is responsible for providing an abstraction of the underlying processor and bus characteristics to the rest of the system. NT drivers must be careful to use the appropriate HAL, NT Executive, and I/O Manager support routines to ensure portability across Alpha, MIPS, PowerPC, and Intel platforms.

The vast majority of the code in the NT I/O subsystem is written in C, a high-level and portable language. NT currently also requires kernel-mode driver developers to write their code in the C language, though it is possible with some extra work to write and link drivers in assembly. However, development in low-level languages, such as assembly, is highly discouraged, because assembly languages are inherently processor/architecture specific, and therefore such drivers cannot execute on more than one type of processor architecture.*

Multiprocessor safe

The I/O subsystem is multiprocessor safe. Windows NT was designed from the ground up to be able to execute on symmetric multiprocessing environments.

Execution of NT kernel-mode code and drivers on multiprocessor machines requires careful synchronization by kernel designers to avoid data consistency problems. For example, on uniprocessor machines, a common practice used to avoid data consistency problems while servicing an interrupt is to disable all other interrupts on the same machine (e.g., via a cli assembly instruction on x86 architectures). However, this same mechanism will fail on symmetric multiprocessor systems, because it is possible to encounter an interrupt on another processor, even though all interrupts had been disabled on the current processor. Similarly, on uniprocessor systems, it can be guaranteed (e.g., via usage of a critical section) that only one thread at a time can access a particular data structure. However, on symmetric multiprocessor architectures, even if preemption of a thread from a single processor were temporarily suspended, other threads executing on other processors could conceivably try to simultaneously access the same data structure.

Typically, spin locks and other higher level (Executive) synchronization mechanisms must be used consistently and correctly in Windows NT drivers to ensure correct functionality on multiprocessor systems.

Modular

The NT I/O subsystem is modular. Any driver within the NT I/O subsystem can be easily replaced by another driver that provides support for the same dispatch entry points supported by the original driver. The use of I/O Request Packets to submit I/O requests and an object-based model where all I/O operations are invoked via standard methods (or well-defined dispatch routine entry points) allows easy replacement of one kernel-mode driver with another that responds appropriately to the same dispatch routines.

All drivers also invoke the services of the I/O Manager using a well-defined and consistent set of service and utility functions. Theoretically, therefore, the I/O Manager is also easily replaceable. In practice, however, the I/O Manager is an extremely complex and integral component of the core NT operating system, and

---

* There are third-party-provided libraries that claim to assist you in developing Windows NT device drivers in C++.

would be extremely difficult to replace easily, even by developers at Microsoft itself.

One obvious benefit of the modularity in the I/O subsystem, however, is the relative ease with which I/O Manager support functions and driver functionality can be reimplemented without affecting any clients that use the services of the I/O Manager or such drivers. As long as the interfaces are maintained consistently, the internals of any implementation can be changed whenever required.

Configurable

All components of the I/O subsystem are configurable. The I/O Manager and all components that comprise the I/O subsystem try to maximize run-time configurability. The NT I/O Manager works with the HAL to determine the set of peripherals connected to the system at boot time. It then initializes the appropriate data structures to support these connected devices. This process avoids any requirements for hardcoding device configurations into the operating system. Windows NT does not as yet support true plug-and-play, though it should in the near future.

Kernel-mode drivers can be developed to manipulate devices; each driver is dynamically loadable and unloadable, minimizing unnecessary kernel overhead. The I/O Manager determines the drivers to be loaded, and the order in which they should be loaded, based upon the entries in the Windows NT Registry. I/O Manager configuration parameters, as well as those required by kernel-mode drivers, are obtained from the Windows NT Registry.

Any drivers that you develop should be as configurable as possible. This includes avoiding any hardcoded values in the driver code and instead obtaining these values from the system Registry, maximizing user configurability.

Process and Thread Context

Before discussing other details specific to the I/O Manager and the I/O subsystem, it would be useful for you to understand the concepts underlying thread/process contexts and to realize why a good grasp of these concepts is essential to understanding the operation of the various components in the Windows NT Kernel. To design and develop kernel-mode drivers under Windows NT successfully, you will need a solid grasp of these issues.

Every process in a Windows NT operating environment is represented by a *process object* structure and has an *execution context* that is unique to that process. The execution context for the process includes the process virtual address space (described in greater detail in the next chapter), a set of resources visible to that process, and a set of threads that belong to the process. Examples of resources owned by a process include file handles for files opened by that process, any synchronization objects created by that process, and any other objects that are created either by the process or on behalf of that process. Each process has at least one thread that is created and belongs to the process, although the process certainly could have numerous threads that belong to it. Note that in Windows NT, the fundamental scheduleable entity is a thread object and not the process object.

Each process is described internally by the Windows NT Kernel by a Process Environment Block (PEB) structure, which is opaque to the rest of the system. The PEB contains process global context, such as startup parameters, image base address, synchronization objects for process-wide synchronization, and loader data structures. Upon creation, the process is also assigned an access token called the primary token of the process. This token is used, by default, by threads associated with the process to validate themselves when they access any Windows NT object.

An *object table* is created for each new process object structure. This object table is either empty or a clone of the parent process object table, depending upon the arguments supplied to the system's create process routine and the inheritance attributes (OBJ_INHERIT) for each of the objects contained within the object table for the parent process. The default access token and the base priority for a new process is the same as that of the parent process.

A *thread object* is the entity that actually executes program code and is scheduled for execution by the Windows NT Kernel. Every thread object is associated with a process object; several threads can be associated with a single process object, which enables concurrent execution of multiple threads in a single address space. On uniprocessor systems, threads can never be executed concurrently; however, on multiprocessor systems, concurrent execution is possible and does occur.

Each thread object has a *thread context* unique to it. This context is architecture-dependent and is typically composed of the following:

- Distinct user and kernel stacks for the thread, identified by a user stack pointer and a kernel stack pointer
- Program counter
- Processor status
- Integer and floating-point registers
- Architecture-dependent registers

You will notice that object handles and other related information about open object structures stored in the process' object table are global to all threads associated with the process. Therefore, all threads in a process can access all open handles for the process, even those opened by other threads within the process. Threads belonging to other processes can only access objects that belong to the process to which they are affiliated; any attempt to access a resource owned by another process will result in an error returned by the Object Manager component in Windows NT.*

Threads are typically referred to as *user-mode* or *kernel-mode* threads. Note that there is no difference in the internal representation of such threads, as far as the Windows NT operating system is concerned. The only conceptual difference between such threads is the mode of the processor when the thread typically executes code, and the virtual address range that is therefore accessible by the thread. For example, a Win32 application process contains threads that execute code while the processor is in user mode and therefore are referred to as user-mode threads. On the other hand, there is a global pool of worker threads created by the Windows NT Executive in the context of a special system process that are used to execute operating system or driver code when the processor is in kernel mode; these threads are typically referred to as kernel-mode threads.

Although user-mode threads typically execute code with the processor in user mode, they often request system services, such as file I/O, which result in the processor executing a *trap* and entering kernel mode to execute the file system code that will service the I/O request. Notice that the user-mode thread is now executing operating system (file system driver) code with the processor in kernel mode, with all the rights and privileges that exist while the processor in this state. While executing in kernel mode, the thread can access kernel virtual addresses and perform operations that are otherwise always denied while the processor is in user mode.

Execution contexts

Consider a kernel-mode driver that you develop. The fact that this is a *kernel-mode driver* tells us that, while the code is being executed, the processor will be in kernel mode and will therefore be able to access the kernel virtual address range. You might wonder which set of threads will execute the code that you develop. Will it be some special thread that you would have to create, or will it be a user-mode thread that requests services from your driver, or will it be a thread on loan from the pool of system worker threads I referred to earlier?

The answer is, it depends. Your driver might always execute code in the context of a special thread that you may have created at driver initialization time, or it

---

* Typically, if you write a kernel-mode driver that attempts to use a handle that is not valid within the execution context of the currently executing process, you will see an error status of STATUS_INVALID_HANDLE returned to you.

might execute code in the context of a user thread that has requested I/O services, or it might be invoked in the context of system worker threads. It is quite possible that, if you develop a file system driver, your driver will execute code in the context of all three types of threads. Furthermore, if you develop device drivers or other lower-level drivers that have their dispatch routines invoked in response to interrupts, your code will execute in the context of whichever thread was executing on that processor at the particular instant when the interrupt occurs. This is referred to as execution of code in the context of an *arbitrary thread,* i.e., a thread whose context is unknown to your driver. The operating system temporarily "borrows" the execution context of this thread to execute your driver routines simply because this thread happened to be executing code on the processor at the time the interrupt occurred.

As a kernel-mode driver designer, you must, therefore, always be aware of the execution context in which your code will execute. This execution context is always one of the following:

The context of a user-mode thread that has requested system services

> If you develop a file system driver or a filter driver that resides above the file system in the driver hierarchy, then your code will often execute in the context of the user-mode thread that requested, say, a read operation. Your code will then be able to access the kernel virtual address range, as well as the virtual addresses in the lower 2GB of the virtual address space belonging to the user-mode process to which the requesting user-mode thread belongs.[*]
>
> Typically, only file system drivers or filter drivers that intercept file system requests should expect that their dispatch routines[†] will be executed directly in the context of user-mode threads. Other drivers cannot expect this, simply because higher-level drivers might have posted the user request to be executed asynchronously in the context of a worker thread, or your driver code might be executed in response to an interrupt as discussed previously.

The context of a dedicated worker thread created by your driver or by some kernel-mode component (typically a component belonging to the I/O subsystem)

> File system drivers sometimes create special threads in the context of the system process (using the PsCreateSystemThread() system service routine described in the DDK) that they subsequently use to perform operations that cannot otherwise be performed in the context of user-mode threads requesting I/O services. Filter drivers might also choose to create such dedi-

---

[*] See the next chapter for a detailed discussion on virtual address spaces.

[†] Dispatch routines are the entry points into a kernel-mode driver. Later in this chapter, I will describe the possible dispatch routines that a kernel-mode driver could have.

cated worker threads; or for that matter, any kernel-mode component can choose to create one or more worker threads.

If you write a file system driver, you might occasionally request that certain operations be carried out by such threads created by you. Your code will then execute in the context of your special threads. If, however, you write lower-level drivers, and if the file system uses a special thread to process I/O requests, your driver might now be invoked in the context of the special thread created by the file system driver. Either way, you can see that the code executes in the context of specially created threads belonging to the system process.

The context of system worker threads specially created by the I/O Manager to serve I/O subsystem components

It is possible for certain I/O operations to be performed in the context of system worker threads that are created by the I/O Manager. These worker threads are often used by file system driver implementations, or by device drivers or other kernel-mode components that need thread context to perform their operations. For example, consider asynchronous I/O requests from user-mode applications. Typically, a file system driver will service such a request by "posting" the request to be picked up and handled by a system worker thread. Control is immediately returned to the calling application once the request has been posted, and the I/O Manager will notify the application once the request has been serviced in the context of the system worker thread. In such a situation, all lower-level drivers will have their dispatch routines invoked in the context of the system worker thread. Note that a system worker thread belongs to the system process, just like the dedicated worker threads created by kernel-mode components described earlier.

The important point to note here is that once the request has been posted to the system worker thread, the virtual address space now accessible in the context of the system worker thread is not the same as the virtual address space that was accessible in the context of the original, user-mode thread that requested the I/O operation. Similarly, the resources that were valid in the context of the original user-mode thread are no longer valid in the context of the system worker thread. The reason for this is obvious: the system worker thread executes in the context of the system process, and the user-mode thread that requested the I/O operation belongs to a distinct application process with its own object table, virtual address space, and process environment block.

The context of some arbitrary thread

Consider now a device driver able to service one IRP at any given point in time. Typically, most device drivers respond to I/O requests by queuing the

IRP for delayed processing, and by returning control immediately to the driver above it in the hierarchy. The IRP will be processed later when the driver can get to it, which is when I/O Request Packets before it in the queue have been processed.

So how is an IRP taken off the queue? Once the current I/O operation is completed by the target device, the device informs the operating system via a hardware interrupt. The operating system responds to this interrupt by invoking the Interrupt Service Routines that various drivers have associated with that specific interrupt. One of these Interrupt Service Routines will be the ISR specified by your driver. As part of ISR execution, the current IRP will complete, and the next IRP will be taken off the device queue and scheduled for actual I/O.*

The point to note here is that the ISR is executed asynchronously, in the context of the currently executing thread—an arbitrary thread. Therefore, when responding to such an interrupt, the driver cannot assume that the virtual address space accessible to it is the same as that of the user thread that requested the IRP now being completed. Resources associated with that thread are not available to the driver code either, because the driver does not know which thread's context is being borrowed to execute the ISR code.

Importance of thread and process contexts

Your kernel-mode driver code will be invoked in one of the execution contexts described previously. The code you develop should be aware of the execution context in which it will be invoked, since that determines the restrictions under which your driver must operate.

Consider the case where you develop a kernel-mode driver that needs to open some object; for example, your driver may perform file I/O itself and may therefore open a file and receive a file handle in return.† If you open this file in your driver initialization code (the `DriverEntry()` routine that every kernel-mode driver must have), you should be aware that this handle will only be valid in the

---

* If you do develop device drivers, you will note that most processing described above is actually performed as a Deferred Procedure Call (DPC) initiated by the ISR. However, the DPC is also executed in the context of an arbitrary thread. Although I will not focus on DPCs and device driver development in this hook, you can consult the DDK for more information.

† Although it may seem strange that a kernel-mode driver might want to perform file I/O, there are filter drivers that provide functionality that requires such capabilities. A strength of the object-based, layered model followed by Windows NT components is that kernel-mode drivers have a tremendous amount of flexibility in terms of services available to them. This leads to the design of very robust, and useful, kernel-mode drivers.

context of the kernel process and the threads associated with the kernel process. So, if you use this handle in the context of system worker threads, the handle will be valid. However, if you attempt to use the handle in the context of a user thread, or an arbitrary thread context, your handle will not be valid. Similarly, if your driver opens an object while servicing a read request in the context of a user thread, the handle can be used only in the context of that thread. Any attempt to use the handle in the context of a system worker thread, for example, will result in an error.

You must be also be aware of when you can safely use the user buffer address, passed to your driver, for a read or write I/O operation. The user specifies a virtual address pointer that is perfectly valid in the context of that particular user thread. However, if the I/O operation is not performed in the context of that user thread (e.g., the I/O operation is performed asynchronously), the virtual address passed in by the user application will no longer be valid and therefore cannot be used by the kernel-mode driver. The I/O Manager provides support for accessing user buffers in other contexts besides that of the requesting thread. I will discuss this support in detail later in this chapter.

As discussed above, there are certain restrictions on the resources that can be used by your driver, depending on the thread context in which your code executes. This thread context depends on the circumstances under which your code is invoked, and this context will determine the resources that your driver can utilize.

Objects and handles

All objects created by kernel-mode components in the Windows NT Executive can be referred to in two ways, either by using an object handle returned by the NT Object Manager when the object is created or opened, or by using a pointer to the object. Note that the pointer to an object allocated by a kernel-mode component will typically be valid in all execution contexts, because the virtual address referring to the object will be from the kernel virtual address range (more on this in the next chapter). However, as mentioned earlier, object handles are specific to the execution context in which the handle is obtained and hence are valid only in that particular execution context.

Remember that each object created by the NT Object Manager has a reference count associated with it. When the object is initially created, this reference count is set to 1. The reference count is incremented whenever a kernel-mode component requests the Object Manager to do so, typically via an invocation of ObReferenceObjectByHandle (), which is described in the DDK. The reference count is decremented whenever a close operation is performed on the object handle. Kernel-mode drivers use the ZwClose ( ) system service routine to close a handle to any system-created object. The reference count is also decremented when a kernel-mode component invokes `ObDereferenceObject()`, which requires the object pointer to be passed in. When the object count goes to zero, the object will be deleted by the NT Object Manager.

In the course of this book, you will often find places where we open an object and receive a handle, then obtain a pointer to the object and stash it away someplace (possibly in global memory), reference the object, and close the handle. This allows us two advantages:

- By saving a pointer to the object, we can always reobtain a handle to the same object in the context of a thread other than the one that originally opened the object. You can find concrete examples of this later in the book.
- By referencing the object and closing the original handle, we are assured the object will not be deleted (until we finally dereference it for the last time), yet we are also assured that, once the last dereference operation is performed, the object will automatically be deleted.

Keep the above discussion in mind as you go through the discussion and code presented throughout this book. This methodology of working with objects and object handles will probably be used extensively by you when you develop your own kernel-mode driver.

Common Data Structures

Data structures are the heart of any computer application or operating system. The NT I/O Manager defines certain data structures that are important to kernel-mode driver designers and developers. Often, your driver will have to create and maintain one or more instances of these data structures to provide driver functionality. In this section, I will briefly discuss the structure and uses of some of the data structures that are important to file system driver and filter driver developers. Note that all of these structures are well documented in the Windows NT DDK. However, our objective here is to understand the reason for creating and working with these data structures, as well as to get a good understanding of the important fields that comprise these data structures.

Driver Object

The DRIVER_OBJECT structure represents an instance of a loaded driver in memory. Note that a kernel-mode driver can only be loaded once; i.e., multiple instances of the same driver will not be loaded by the Windows NT I/O Manager. The driver object structure is defined as follows:

```
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_DISPATCH              MajorFunction[IRP_MJ_MAXIMUM_FUNCTION + 11;
} DRIVER_OBJECT;
```

Earlier in this chapter, I discussed the NT packet-based I/O model. Each I/O Request Packet describes an I/O request. The major function of an I/O request packet is to request functionality from a driver.

We know that the IRPs will have to be dispatched to some I/O driver routines. If you examine the driver object structure, you will notice that it contains memory allocated for an array of function pointers called the MajorFunction array. It is the responsibility of the kernel-mode driver to initialize the contents of this array for each major function that the kernel-mode driver supports. There are no restrictions on the number of functions that your driver must support, nor are there any restrictions specifying that each function pointer should point to a unique function; you could initialize the entry points for all major functions to point to a single routine and this would work perfectly (as long as your driver routine handled all the IRPs that would be directed to it). If you develop a kernel-mode driver, you will probably support at least one major function and should therefore initialize the function pointers appropriately.

The DriverStartlo and the DriverUnload fields are also left for the driver to initialize. Lower-level Windows NT drivers typically provide a Startle function, which is invoked either when an IRP is dispatched to the driver, or when an IRP has just been popped off a queue. The DriverStartlo field is initialized by lower-level drivers to point to this driver-supplied StartIO function. Typically, as you will see in code presented later in this book, file system drivers and filter drivers will not need a `DriverStartIo` routine, because such drivers manage their pending I/O Request Packets via other internal queue management implementations. The `DriverUnload` field should point to a routine that is executed just before the driver is unloaded. This allows your kernel-mode driver an opportunity to ensure that any on-disk information is in a consistent state, as well as to allow lower-level drivers to put the device(s) they control into a known state. Note that it is not required that your driver be unloadable; in particular, file system drivers are extremely difficult to design so that they can be unloaded on demand. If your driver cannot be unloaded, you must not initialize the `DriverUnload` field in the driver object structure (the field is initialized to NULL by the I/O Manager and therefore your driver entry routine need not do anything to this field).

Many kernel-mode drivers create one or more device object structures. These structures are linked in the `DeviceObject` field in the driver object structure. At driver load time, this linked list is empty. However, the NT I/O Manager fills the list with pointers to device objects created by your driver as such device objects are created using the `IoCreateDevice()` service routine.

To load a driver, the I/O Manager executes an internal routine called `IopLoadDriver()`. This routine performs the following functionality:

- Determines the name of the driver to be loaded and checks whether the driver has already been loaded by the system.

  The I/O Manager checks to see whether the driver has already been loaded by examining a global linked list of loaded kernel modules. If the driver is already loaded, the I/O Manager immediately returns success; otherwise, it continues with the process of loading the driver. To have your driver loaded, your installation utility must have created an appropriate entry in the Registry. See Part 3 for more information on how the Registry must be configured for kernel-mode file system and filter drivers.

- If the driver is not loaded, the I/O Manager requests the Virtual Memory Manager (VMM) to map in the driver executable. As part of mapping in the driver code, the VMM checks to see that the file contains a valid Windows NT executable format. If the driver was built incorrectly, the VMM will fail the map request and the I/O Manager, in turn, will fail the driver load request.
- Now the I/O Manager invokes the Object Manager, requesting that a new driver object be created. Note that the DRIVER_OBJECT type is an I/O Manager-defined object type, which was previously created by the I/O Manager at system initialization time; it is therefore recognized as a valid object type by the NT Object Manager. Note also that the returned driver object structure is

allocated from nonpaged system memory and is, therefore, accessible at all IRQ levels.

- The I/O Manager zeroes out the driver object structure returned by the Object Manager. Each entry in the `MajorFunction` array is initialized to `IopInvalidDeviceRequest` (). This is the default dispatch routine for the various entry points. This routine simply sets a return status of STATUS_INVALID_DEVICE_REQUEST and returns control to the calling process.
- The I/O Manager initializes the `DriverInit` field to refer to the initialization routine in your driver (the `DriverEntry` routine). `DriverSection` is initialized to the section object pointer* for the mapped executable, `DriverStart` is initialized to the base address to which the driver image was mapped, and `DriverSize` is initialized to the size of the driver image.
- The I/O Manager requests that the object be inserted into the linked list of driver objects maintained by the NT Object Manager. In return, the I/O Manager gets a handle to the object. This handle is referenced by the I/O Manager and closed, thereby ensuring that the object will be deleted when dereferenced at driver unload time.
- The `HardwareDatabase` field is initialized with a pointer to the Configuration Manager's hardware configuration information; this field could be used by lower-level drivers to determine the hardware configuration for the current boot cycle. The I/O Manager also initializes the DriverName field so that it can be used by the error logging component when required.
- Finally, the I/O Manager invokes the driver initialization routine, which is where your driver gets the opportunity to initialize itself, including initializing the function pointers in the driver object structure. You should note that your driver initialization routine is always invoked at IRQL PASSIVE_LEVEL, allowing you to use pretty much all of the system services available. Furthermore, your initialization routine will be invoked in the context of the system process; this is especially important to keep in mind if you open any objects or create any objects resulting in a handle being returned to you. Any such handles will only be valid in the context of the system process. In order to be able to use such objects in the context of other threads, you will have to use the methodology described earlier in the chapter, where you obtain a pointer to the object and then subsequently obtain handles in the context of other threads as and when required.

  If your driver fails the initialization routine it will automatically be unloaded by the Windows NT I/O Manager. Remember to deallocate any allocated

---

* Chapter 5, *The NT Virtual Memory Manager,* explains section objects and the process of mapping files in greater detail.

memory prior to returning control to the I/O Manager and also to close and dereference any open objects, or else you will leave a trail behind you that could lead to degraded or impaired system behavior.

The driver entry routine is the initialization routine for a kernel-mode driver and is invoked by the I/O Manager. Each kernel-mode driver can also register a reinitialization routine that is invoked after all other drivers have been loaded and the rest of the I/O subsystem, as well as other kernel-mode components, have been initialized. In NT 3.51 and earlier, the Count field in the driver object structure contained a count of the number of times the reinitialization routine had been invoked.

Beginning with NT 4.0 and later, the NT I/O Manager allocates an additional structure that is an extension of the original driver object structure. This *driver extension* structure is defined below and contains fields to support plug-and-play for lower-level drivers that manage hardware devices and peripherals. The Count field has been moved to the driver extension structure with the new release; however, it still provides the same functionality as it did in earlier releases. Plug-and-play support is provided by lower-level drivers and will not be covered in this book.

```
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;
```

Finally, notice that there is a pointer to a *fast I/O dispatch table* in the driver entry structure. Currently, only file system driver implementations provide support via the fast I/O path. Essentially, the fast path is simply a way to avoid the abstract, clean, modular, yet relatively slow method of using packet-based I/O. Using the function pointers provided by the file system driver in this structure, the NT I/O Manager can either directly invoke the file system dispatch routines or call directly into the NT Cache Manager to request I/O without having to set up an IRP structure. The FastIoDispatch field should be initialized by the driver entry routine to refer to an appropriate structure containing initialized file system entry points. In the coverage of the NT Cache Manager, provided later in this book, you will see a detailed discussion of the entry points that comprise the fast I/O method of I/O.

Device Object

Device object structures are created by kernel-mode drivers to represent logical, virtual, or physical devices. For example, a physical device, such as a disk drive, is represented in memory by a device object. Similarly, consider the situation where you develop an intermediate driver that presents a large physical disk as three smaller disks or partitions. Now, there will be one device object, representing a large physical disk, that is created by the lower-level disk driver, and your intermediate driver should create three additional device objects, each of which represents a virtual disk. Finally, a driver might choose to create a device object to represent a logical device; for example, the file system drivers create a device object to represent the file system implementation. This device object can be opened by other processes and can be used to send specific commands targeted to the file system driver itself.

Without a device object, a kernel-mode driver will not receive any I/O requests, since there must be a target device for every I/O request dispatched by the I/O Manager. For example, if you develop a disk driver and do not create a device object structure representing this particular disk device, no user process can access this disk. Once you do create a device object for the disk, however, file system drivers can potentially mount any volumes present on the physical media and user-mode processes can try to read and write data from the disk.

Unnamed device objects are rarely created by kernel-mode drivers, since such device objects are not easily accessible to other kernel-mode or user-mode components. If you create an unnamed device object, none of the other components in the system will be able to open it, and therefore, no component will direct any I/O to it. However, one common example of unnamed device objects are those created by file system drivers to represent mounted file system volumes. In this case, there is a device object, created by the disk driver representing the physical or virtual disk, on which the file system volume resides, and a Volume Parameter Block (VPB) structure (described later) performs the association between the named physical disk device object and the unnamed logical volume device object created by the file system driver. I/O requests are sent to the device object representing the physical disk. However, the I/O Manger checks to see whether the disk has a mounted volume on it (mounted volumes are identified by an appropriate flag in the VPB structure for the device object that represents the physical disk), and if so, it redirects the I/O to the unnamed device object representing the instance of the mounted volume.

When your driver issues a call to loCreateDevice() to request creation of a device object, it can specify an additional amount of nonpaged memory to be allocated and associated with the newly created device object. The reason is to have a global memory area reserved for and associated with that particular device object. This memory is called the device object extension and will be allocated by the I/O Manager on behalf of your driver. The I/O Manager initializes the DeviceExtension field to point to this allocated memory. There are no constraints mandated by the I/O Manager on how this memory object should be used by your driver. You may wonder what the difference is between requesting a device extension and declaring global static variables. The answer can *be* summed up as potentially cleaner code design. Another important benefit is that device-specific global variables stored in a device object extension become logically associated with the device object immediately, and therefore you can avoid unnecessary acquisition of synchronization resources before accessing this device-object-specific data.

Any static variables declared by your kernel-mode driver are global to the entire Windows NT operating system. They are also not logically associated with any particular device object, so if your driver creates and manages multiple device object structures, you will have to design some method where the global structures can be associated with specific device objects. Note, however, that both statically declared global variables and the device extensions are allocated from nonpaged pool, although you can request that your static variables be made pageable (typically, this is never done). Many kernel-mode drivers make use of both statically declared global variables that are required by the entire driver, and a driver extension containing global variables that are specific to the context of a certain device object structure.

The device object structure is defined as follows:

```
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;
    DEVICE_TYPE                      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;
```

Any kernel-mode driver can direct the I/O Manager to create a device object using the loCreateDevice () routine. This routine, if successful, will return a pointer to the device object structure that is allocated from nonpaged memory. Many of the fields in the device object structure are reserved for use by the I/O Manager. A brief description of the important fields is given below:

- As long as the ReferenceCount field is nonnull, two invariants hold true. First, the device object will never be deleted. Second, the driver object representing the driver that created this device object will never be deleted (i.e., the driver will never be unloaded as long as any of the device objects created by the driver has a positive reference count). The ReferenceCount field is manipulated at various times by the I/O Manager and can also be manipulated by the driver.* An example of this field being incremented by the I/O Manager is whenever a new file stream is opened on a mounted volume; the reference count for the device object representing the mounted volume is incremented by 1 to ensure that the volume is not dismounted as long as any file is open. This also ensures that the file system driver is not unloaded as long as any file is open, since unloading the driver could lead to a system crash. Similarly, whenever a new volume is mounted, the device object representing the logical volume has its reference count incremented to ensure that both the device object and the corresponding driver object are not deleted.
- The I/O Manager initializes the DriverObject field to refer to the driver object representing the loaded instance of the kernel-mode driver that invoked the loCreateDevice ( ) routine.
- All device objects created by a kernel-mode driver are linked together using the NextDevice field in the device object. Note that there is no particular order in which a kernel-mode driver, traversing this linked list, should expect to find created device objects. As it happens, the I/O Manager adds new

---

* Be careful if your driver manipulates the ReferenceCount field in the device object, because there is no method with which you can synchronize your operation with that of the I/O Manager. This could lead to inconsistent behavior.

device objects to the head of the linked list; therefore, you will probably find the last device object inserted at the beginning of the list.

- In this chapter, as well as in Chapter 12, *Filter Drivers,* you will be exposed to more detail about how filter drivers can be developed for Windows NT environments. These filter drivers are intermediate-level drivers that intercept I/O requests targeted to certain device objects by interjecting themselves into the driver hierarchy and by attaching themselves to the target device objects. The concept of attaching to a device object is simple, as illustrated in Figure 4-3.

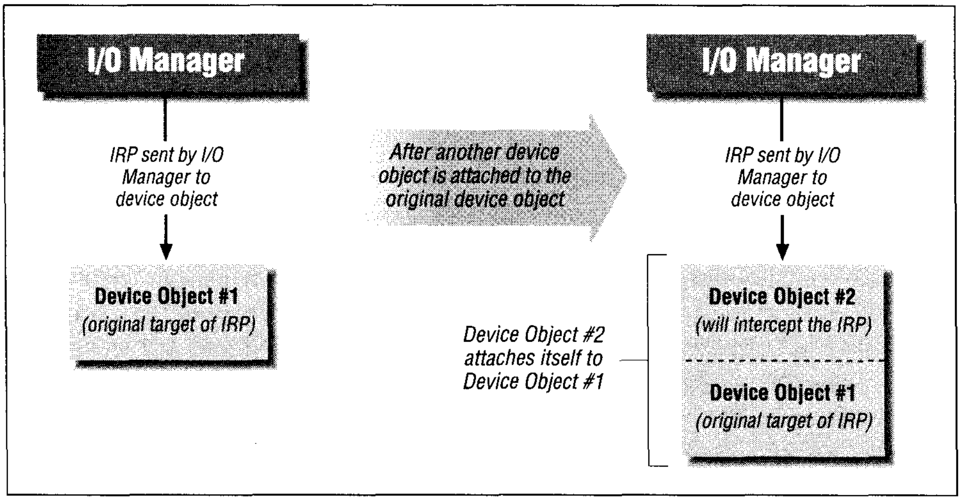
I/O Manager
IRP sent by I/O Manager to device object
Device Object #1
(original target of IRP)
After another device object is attached to the original device object
I/O Manager
IRP sent by I/O Manager to device object
Device Object #2
(will intercept the IRP)
Device Object #1
(original target of IRP)
Device Object #2 attaches itself to Device Object #1

Figure 4-3. Illustration ofa device object being attached to another

When a device object is attached to another (via the I/O-Manager-provided `loAttachDevice ( )` or the `loAttachDeviceByPointer ( )` routines), the `AttachedDevice` field in the device being attached to (device object #\ in Figure 4-3) will be set to the address of the device object being attached (device object #2).

- The `CurrentIrp` field is of interest to designers of device drivers or other lower-level drivers. Such drivers typically use the I/O-manager-supplied `loStartNextPacket ()` or `loStartPacket ()` routines to queue and dequeue an IRP from the driver queue of pending IRPs. Once the I/O manager dequeues a new IRP, it makes the dequeued IRP the current IRP to be processed by the driver. To do this, it inserts the IRP pointer in the `CurrentIrp` field of the device object. The I/O manager subsequently passes a pointer to `DeviceObject->CurrentIrp` when invoking the device driver `Startlo ()` dispatch routine.

  This field is typically not of much interest to higher-level drivers.

- The `Timer` field is initialized when the driver invokes `IoInitializeTimer()`. This allows the I/O Manager to invoke the driver-supplied timer routine every second.
- The device object `Characteristics` field describes some additional attributes for the physical, logical, or virtual device that the object represents. The possible values are FILE_REMOVABLE_MEDIA, FILE_READ_ONLY_DISK, FILE_FLOPPY_DISK, FILE_WRITE_ONCE_MEDIA, FILE_REMOTE_DEVICE, FILE_DEVICE_IS_MOUNTED, or FILE_VIRTUAL_VOLUME. This field is manipulated by the I/O Manager, as well as by the file system or kernel-mode driver that manages the device object.
- The `DeviceLock` is a synchronization-type event object allocated by the I/O Manager. Currently, this object is acquired by the I/O Manager prior to dispatching a mount request to a file system driver. This allows synchronization of multiple requests to mount the volume. You should only be concerned with this event object if you design a file system driver that uses the I/O-Manager-supplied `IoVerifyVolume()` routine (described in Part 3). In that case, you should be careful not to invoke that routine when you get a mount request from the I/O Manager, since the `DeviceLock` would have been previously acquired by the I/O Manager prior to sending you the mount IRP; invoking the verify routine would cause the I/O Manager to try to reacquire this resource and cause a deadlock.
- The I/O Manager allocates memory for the device extension and initializes the `DeviceExtension` field to point to this allocated memory.

I/O Request Packets (IRP)

As described earlier, the Windows NT I/O subsystem is packet-based. Kernel-mode drivers that comprise the I/O subsystem receive I/O Request Packets (IRP), which contain details of the operation being requested. The recipient of the IRP is responsible for processing the IRP, and either forwarding it on to another kernel-mode driver for additional processing, or completing the `IRP`, indicating that processing of the request described in the IRP has been terminated.

IRP allocation

All I/O requests are routed through the NT I/O Manager. Most often, a user process executes a Win32- or other subsystem-specific I/O request (e.g., `CreateFile()`), and this request gets translated to an NT system service call to the I/O Manager. Upon receiving the I/O request, the I/O Manager identifies the driver that should service the I/O request. Most likely, this will be a file system driver that will have mounted the file system on the physical device to which the I/O request is targeted.

To dispatch the request to the kernel-mode driver, the I/O Manager allocates an I/O Request Packet using the routine IoAllocateIrp().* This structure is always allocated from nonpaged pool. The method of allocation'differs slightly in the various versions of Windows NT.

> *NOTE* A *zone* is a system-defined structure supported by the Windows NT Executive and is used to efficiently manage allocation and deallocation of fixed-sized chunks of memory. Allocating and freeing memory using zones is more efficient than asking for small chunks of memory from the VMM, which could also lead to some internal memory fragmentation. Using a zone requires your driver to perform two steps: first, allocate the memory that will comprise the zone and inform the NT Executive about this allocated pool, as well as the size of entries you will allocate from the zone; second, use the available ExAllocateFromZone() and other related support routines to allocate and free entries using the zone.
>
> Read Chapter 2, *File System Driver Development,* for a discussion on how to use zones in your driver.

In NT version 3.51 and earlier, the I/O Manager first attempts to allocate the IRP from a zone composed of fixed-sized IRP structures. As you will read later in this discussion of IRPs, the size of the IRP depends upon the number of *stack locations* that are required for the IRP. Therefore, the I/O Manager keeps two zones available, one for IRPs with relatively fewer stack locations, and the other for I/O Request Packets with a larger number of stack locations. If the zone from which allocation is attempted is found empty (this can happen in high-load situations where an extremely large amount of concurrent I/O is in progress), the I/O Manager requests memory for the IRP directly from the VMM (actually, the I/O Manager uses the ExAllocatePool() support routine provided by the NT Executive). For I/O requests that originate in user-mode, if no memory is currently available, an error is returned to the user application indicating that the system is out of available resources. However, for I/O requests that originate in kernel-mode, the I/O Manager attempts to allocate memory for the IRP from the NonPagedPoolMustSucceed memory pool. If this memory allocation request does not succeed, the attempt will result in a system bugcheck.

The methodology used in NT version 4.0 is similar with one slight variation: the I/O Manager uses lookaside lists, a new structure used to manage fixed-sized pools of memory introduced in this new release, instead of zones. The reason for

---

* The IoAllocateIrp() routine is documented in the DDK. It can also be used by other kernel-mode drivers to request an IRP to be allocated. Supply a FALSE for the ChargeQuota argument required with this routine invocation.

this new structure is to gain some efficiency, because lookaside lists do not always use spin locks to perform synchronization; instead they use an atomic 8-byte compare exchange instruction on architectures where such support is possible.

Other kernel-mode components besides the I/O Manager can use the I/O-Manager-supplied routine loAllocatelrp () to request a new IRP structure. This IRP can subsequently be used to send a I/O request to a kernel-mode driver. Other routines provided by the I/O Manager that also use loAllocatelrp () to obtain a new IRP structure and then return these newly allocated IRPs after the initialization of certain fields are loMakeAssociatedlrp (), loBuildSynchronousFsdRequest ( ) , loBuildDeviceloControlRecjuest ( ) , and loBuildAsynchronousFsdRecruest ( ) . Consult the DDK for more information on these routines. Part 3 also uses some of these routines in implementing filter drivers.

IRP structure

Logically, each I/O Request Packet is composed of the following:

- The IRP header
- I/O Stack Locations

The IRP header contains general information about the I/O request, useful to the I/O Manager as well as to the kernel-mode driver that is the target of the request. Many of the fields in the IRP header can be accessed by a kernel-mode driver; other fields exist solely for the convenience of the I/O Manager and should be considered off-limits by the drivers processing the IRP.

Here is a brief explanation of important fields that comprise the IRP header:

`MdlAddress`

A Memory Descriptor List (MDL) is a system-defined structure that describes a buffer in terms of the physical memory pages that back up the virtual address range comprising the buffer. There are different ways in which buffers used for I/O request handling can be passed down to the kernel-mode driver. Descriptions for the three methods will appear shortly. Remember for now, though, that if the *Directlo* method is used, the MdlAddress field will contain a pointer to the MDL structure that can then be used in data transfer operations.

`Associatedlrp`

This field is an union of three elements, defined as follows:

```
union {
    struct _IRP             *MasterIrp;
    LONG                     IrpCount;
```

```
        PVOID                       SystemBuffer;
    } Associatedlrp;
```

Any IRP structure that has been allocated can be categorized as either a *master IRP* or an *associated IRP*. An associated IRP is, by definition, associated with some master IRP, and can be created only by a higher-level kernel-mode driver. By creating one or more associated IRPs, the highest-level driver can split up the original I/O request and send each associated IRP to lower-level drivers in the hierarchy for further processing.

For example, higher-level drivers sometimes execute the following loop:

```
while (more processing is required) {
    create an associated IRP using loMakeAssociatedlrp ( ) ;
    send the associated IRP to a lower-level driver using
    loCallDriver ( ) ;
    if ( STATUS_PENDING is returned) {
        wait on an event for the completion of the associated IRP;
    } else {
        associated IRP was completed;
        check result and determine whether to continue;
    }
}
```

For an associated IRP, the union described here contains a pointer to the master IRP. For a master IRP, however, this union contains the count of the number of associated IRPs for this master IRP; or, if no associated IRPs have been created, the `SystemBuffer` pointer might be initialized to a buffer allocated in kernel virtual address space for data transfer. System buffers are allocated by the I/O Manager when a kernel-mode driver requests *buffered I/O* (described later in this book).

Note that the `IrpCount` field is manipulated under the protection of an internal I/O Manager resource. Therefore, external kernel-mode drivers must not attempt to manipulate or access the contents of this field directly.

ThreadListEntry

This field is typically manipulated by the I/O Manager. Before invoking a driver dispatch routine via `loCallDriver ( )`, all I/O Manager routines insert the IRP into a linked list of IRPs for the thread in whose context the I/O operation is taking place. For example, if a user thread invokes a read request, the I/O Manager will allocate a new IRP structure, and insert it into the list of IRPs being processed by the user thread prior to invoking the file system read dispatch routine.

NOTE There is a field in each thread structure called `IrpList`, which serves as the head of a linked list of pending I/O Request Packets. The `ThreadListEntry` field, described earlier, is used to queue the IRP to this linked list. This list is used to track all pending I/O Request Packets for the thread in question; this is especially useful when the I/O subsystem tries to cancel IRPs for a particular thread.

Note that the `IoAllocateIrp()` routine does not queue the returned IRP to the linked list of outstanding IRPs for the current thread. Therefore, when a cancel request is posted, that IRP will not be found among the list of IRPs for the thread.

`IoStatus`

This field should be appropriately updated by your kernel-mode driver before completing the I/O Request Packet. A description of the structure is provided later in this chapter. Note that this field is part of the IRP structure, and not part of the I/O status block structure passed in to the I/O Manager by the thread requesting the I/O operation. It is the I/O Manager's responsibility to transfer the results of the I/O operation from this field to the I/O status structure submitted by the requesting thread. This operation is performed by the I/O Manager as part of the postprocessing of the IRP, once the IRP has been completed by kernel-mode drivers.

`RequestorMode`

When code in your driver is executed, it would be useful if you knew whether the caller was a user-mode thread (e.g., an application requesting an I/O operation), or if the caller was a kernel component (some other driver requesting your services in the context of a system worker thread).

You may wonder why such information could be useful. Think about the case where the caller is a user-mode thread; you know then that you cannot blindly assume that the arguments passed in to your driver are legitimate. If your driver uses the direct-IO method of passing buffer pointers (explained later), you will need to convert the passed-in addresses to something usable by your kernel-mode code. This is especially true if the request will be handled asynchronously by your driver.

On the other hand, if your driver is invoked from a system worker thread, you could bypass these argument checks, because you could assume that addresses passed in to you are legitimate and usable directly by your driver.

Similarly, the NT I/O Manager, as well as other kernel components such as the Virtual Memory Manager, need to identify and differentiate whether clients of their services are executing kernel-mode (operating system) code, or whether the request came from a user-space component. This information is used to check the legitimacy of the arguments passed in to these kernel-mode components.[*]

The solution used throughout the NT Executive is to identify the processor mode in which the calling thread executed prior to invoking the services of the kernel-mode component. Note that the key concept here is that the previous mode of the calling thread is important; the very fact that the thread is executing kernel-mode code at the instant when the check is made tells us that the current mode will always be kernel mode. To obtain the previous mode information, the I/O Manager directly accesses a field in the thread structure. The `ExGetPreviousMode()` function, declared in the DDK, provides the same functionality to third-party driver developers. This routine returns the previous mode of the thread being checked: user or kernel mode.

The I/O Manager puts the information about the previous mode of the requesting thread into the `RequestorMode` field prior to invoking the `IoCallDriver()` routine, which, in turn, invokes one of your driver dispatch routines. You should use this information both internally in your driver, as well as in invocations to system service routines such as `MmProbeAndLockPages()`.

`PendingReturned`

Each IRP is typically handled by more than one driver in the hierarchy. To process an IRP asynchronously, a kernel-mode driver must execute the following steps:

a. Mark the IRP pending by invoking the `IoMarkIrpPending()` function.

b. Queue the IRP internally.

Lower-level drivers may use a `StartIo()` function instead.

c. Return a status code of STATUS_PENDING.

The `IoMarkIrpPending()` call (implemented as a macro) simply sets the SL_PENDING_RETURNED flag in the `Control` field of the current I/O stack location.[†]

At the time of IRP completion processing, during the execution of the `IoCompleteRequest()` function, the I/O Manager traverses each stack location that had been used by drivers in the hierarchy, looking for any completion routines that may need to be invoked. This traversal of stack locations happens in reverse order from that used in processing the IRP. The most recently used stack location is processed first (the one used by the lowest-

---

[*] If the I/O Manager read system service (`NtReadFile()`) blindly assumed that the passed-in buffer address was a legitimate kernel-mode usable address, malicious users could have a field day overwriting operating system data with their own!

[†] Stack locations are discussed in detail later in this chapter. You may skip this discussion for the moment and come back to it after you have read that section.

level driver in the hierarchy that processed the IRP), followed by the next one, and so on.

As each stack location is unwound, the I/O Manager notes whether the SL_PENDING_RETUKNED flag had been set in the I/O stack location, and sets the PendingReturned flag to TRUE if the flag had been set. However, if the flag was not set in the stack location, the I/O Manager sets the PendingReturned field to FALSE.

WARNING The value of the PendingReturned field may change as the I/O stack locations are being traversed, -while the I/O Manager looks for completion routines that may need to be invoked.

So why is the value of this field important? Well, later on in the loCompleteRequest() function, the I/O Manager checks the value of the PendingReturned field to determine whether or not to queue a special kernel Asynchronous Procedure Call (APC) to the thread that originally requested the I/O operation. Your file system or filter driver will have to cooperate with the I/O Manager to ensure that the right course of action is adopted. You will see how your driver's actions affect the behavior of the I/O Manager later in this chapter.

Cancel, Cancellrql, and CancelRoutine

Kernel-mode drivers that process I/O Request Packets that might potentially require an indefinite time interval to be completed should provide appropriate IRP cancellation support. Our perspective is that of a file system driver or that of a filter driver. We would need to provide this functionality if we do not pass on IRPs to lower-level disk or network drivers but perform our own processing instead. Note that all three fields listed above are manipulated by either the driver or the I/O Manager to provide the capability to cancel pending I/O Request Packets when required.

ApcEnvironment

When an IRP is completed, the I/O Manager performs postprocessing on the IRP, the details of which are given below. The ApcEnvironment field is used internally by the I/O Manager in performing postprocessing on the IRP in the context of the thread that originally requested the I/O operation. This field is initialized by the I/O Manager when allocating the IRP and should not be accessed by driver designers.

Zoned/AllocationFlags

The Zoned field was replaced with the AllocationFlags field in NT version 4.0. Fundamentally, the field (called by whatever name) records internal bookkeeping information used by the I/O Manager during IRP completion to determine whether the IRP was allocated from a *zone/lookaside list,* or from system *nonpaged pool,* or from system *nonpaged-must-succeed pool.* This information is not useful from the kernel driver's perspective, except when debugging the driver and trying to locate all IRP structures allocated out of the global lookaside list or zone.

Caller-supplied arguments

The following are part of the IRP:

```
PIO_STATUS_BLOCK              Userlosb;
PREVENT                       UserEvent;

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

The Userlosb field in the IRP is set by the I/O Manager to point to the I/O status block supplied by the thread requesting I/O. As part of the postprocessing performed by the NT I/O Manager upon completion of an IRP, the I/O Manager copies the contents of the loStatus field to the I/O status block pointed to by the Userlosb field.

Most NT I/O system service routines (documented in Appendix A) accept an optional event argument. This argument (if supplied by the caller) is initialized by the NT I/O Manager to the not-signaled state and is set to the signaled state by the I/O Manager upon completion of I/O. The I/O Manager fills in the UserEvent field with the address of the caller-supplied event object.

The AllocationSize field in the Overlay structure is only valid for file create requests. The user is allowed to specify an optional initial size for a file being created. The I/O Manager initializes the AllocationSize field with this caller-supplied size prior to invoking the file system driver create/open dispatch routine.

Many of the NT system services provided for I/O operations by the NT I/O Manager allow asynchronous operations. The caller thread can request that I/O be performed asynchronously and can also specify an APC to be invoked upon completion of the IRP. For these system services, the I/O Manager dutifully invokes the user-supplied APC, passing it the supplied APC context, as part of the postprocessing performed by the I/O Manager upon completion of the IRP by a kernel-mode driver. The I/O Manager stores the calling-thread-supplied APC function pointer in the UserApcRoutine field. The context is this point, none of the I/O Manager routines use this field to pass information to a kernel-mode driver.[*]

The CurrentStackLocation field is simply a pointer to the current stack location for the IRP. Stack locations are discussed later in this chapter. The important point to note for kernel-mode drivers is to always use I/O Manager-provided access functions to get the pointer to the current and the next stack locations in the IRP. To maintain portability, your driver should never try to access the contents of this field directly.

The OriginalFileObject field is initialized by the I/O Manager to the address of the file object to which an I/O operation is being targeted. The same information is available to the highest-level driver (typically, the file system driver) to which the I/O operation is sent from the current stack location. However, the I/O Manager keeps this information in the IRP header and can therefore access it independently of the manner in which stack locations are manipulated by lower-level drivers. The file object is used in the postprocessing of the IRP after it has been completed. For example, if the file object pointer is not NULL (i.e., the OriginalFileObject field is initialized at IRP allocation), the I/O Manager checks whether it needs to send a message to a completion port,[t] or dereference any event objects, or perform any similar notification or cleanup operation related to that file object. It is legitimate for this field to be NULL, in which case the I/O Manager will skip some of the postprocessing that it would otherwise perform.

The Ape field is used internally by the I/O Manager after the IRP has been completed, to queue an APC request for final postprocessing of the IRP in the context of the thread that issued the I/O request.

As mentioned earlier, each I/O Request Packet is composed of the IRP header, and the stack locations for that IRP. Some of the fields in the IRP structure such as StackCount, CurrentLocation, and CurrentStackLocation are related to stack location manipulation. IRP stack locations are discussed next.

Stack locations and reusing IRP structures

Windows NT I/O request packets are reusable. In a layered driver environment, such as in the Windows NT I/O subsystem, each higher-level driver in the hierarchy invokes the next lower-level driver, until some driver actually completes the

---

[*] If you write a file system driver, you might notice the value of this field is nonnull for directory-control IRPs. However, the same buffer pointer containing the directory name is accessible via the information obtained from the current stack location for the IRP, stored in the Parameters. QueryDirectory.FileName field.

[t] Consult the Win32 SDK for further information about I/O Completion Ports.

original IRP. It is quite possible, and is often the case, that the same IRP is passed down from driver to driver until it is completed.

Completing the IRP requires invoking `IoCompleteRequest()`; after such a call is issued, no component, other than the I/O Manager, can touch that IRP, since it can be deallocated at any time.

So how can a single IRP structure be reused cleanly? The solution provided by the NT I/O Manager is to use stack locations that contain descriptions of the I/O requests to the target device objects. When initially dispatching the IRP to a kernel-mode driver, the I/O Manager fills in one stack location with the parameters for the desired operation. Later, the driver to which the IRP is sent determines whether it can complete the IRP itself, or whether it needs to invoke another driver lower in the hierarchy. If it needs to invoke a lower-level driver, the current holder of the IRP can simply initialize the next IRP stack location, and then invoke the lower-level driver via `IoCallDriver()`, passing it the IRP. This process is repeated until a driver in the chain performs all of the required processing and decides to complete the IRP.

The NT I/O Manager allocates space for multiple associated stack locations when an IRP structure is allocated. Each of these stack locations can contain a complete description of an I/O request. For example, an IRP allocated for a *read* request should contain the following information:

- A function code, which will be examined by the kernel-mode driver to determine the type of request issued. In this example, the function code indicates a read request.
- An offset from which data should be read.
- The number of bytes that are requested.
- A pointer to the output buffer.

In addition to the above, other information relevant to the read request might also be passed to the driver that manages the device object that is the target of the read operation. All of this information is encapsulated into a single stack location structure.

The number of stack locations allocated for an IRP depends upon the `StackSize` field in the target device object to which the IRP is being issued. The `StackSize` field is initialized to 1 when the device object is created; it can then be set to any value by the driver managing the device object. The `StackSize` field is also changed when a device object is attached to another device object. As part of the attach process, the `StackSize` value is set to the value obtained from the device object being attached to, incremented by 1. The logic here is simple: an IRP sent to a device object needs one stack location for the initial target device object; it also needs one stack location for each filter and/or driver in the hierarchy that will perform some processing on the I/O Request Packet.

As shown in Figure 4-4, if a read request is sent to the file system driver that has a volume mounted on disk A, the I/O Manager will allocate four stack locations when creating the read IRP. These stack locations are used in reverse order, similar to the last-in-first-out usage of a stack structure. When invoking a driver, the I/O Manager always pushes the stack location pointer to point to the next stack location; when the called driver releases the IRP, the stack location pointer is popped to once again point to the previous stack location. Therefore, when invoking the filter driver dispatch routine in Figure 4-4 below, the I/O Manager uses stack location #4, the last stack location allocated.

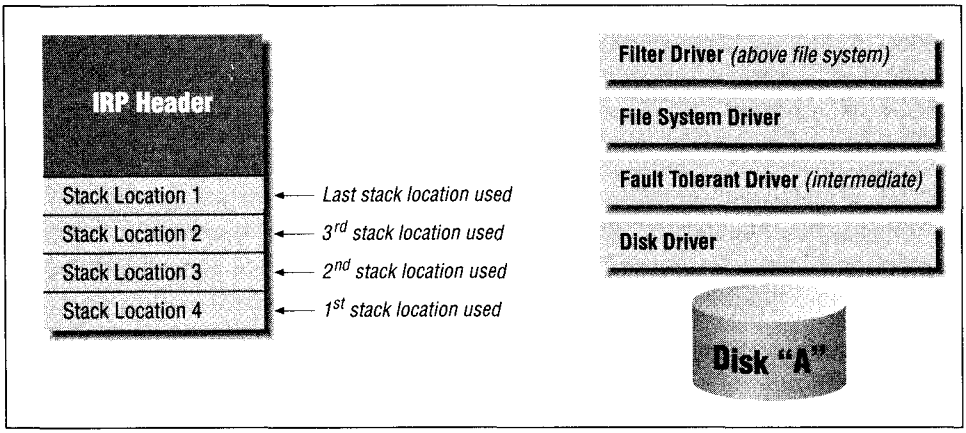

Figure 4-4. IRP stack locations usedfor a driver hierarchy

The NT I/O Manager initializes the `StackCount` field in the IRP header with the total number of stack locations allocated for that IRP. The CurrentLocation field in IRP header is initialized by the I/O Manager to (StackCount + 1). This value is decremented each time a driver dispatch routine is invoked via `loCallDriver()`.

Therefore, if the `StackCount` is 4, the initial value of `CurrentLocation` is set to 5, which is an invalid stack location pointer value. The reason for this, however, is that to dispatch an IRP to the next driver in the hierarchy, the kernel component must always get a pointer to the next stack location and then fill in appropriate parameters for the request.

When an IRP is being dispatched to the first driver in the hierarchy, the next stack location will be (CurrentStackLocation—1) equal to 4, the correct value for the stack location used for the filter driver above.

The I/O Manager often performs sanity checks using this value to ensure that the IRP is being routed correctly through the I/O subsystem. For example, in IoCallDriver (), the I/O Manager first decrements the CurrentLocation field (since a new driver is being invoked, it requires the next IRP stack location), then checks to see if the CurrentLocation value is less than or equal to 0. If the value does become less than or equal to 0, it is obvious that IoCallDriver ( ) is being invoked once too often for the number of stack locations that were initially allocated (or that there is some stray pointer corrupting memory), and therefore the I/O Manager performs a bugcheck with the error code of NO_MORE IRP STACK_LOCATIONS.

NOTE The reason for a bugcheck is that, by the time the IoCallDriver ( ) is invoked, critical damage may have already been done, since the caller will in all likelihood have filled in the contents of the next stack location for the use of the driver being called. However, in this situation, the next stack location is some unallocated memory at the end of the IRP structure, which could literally be anything.

Continuing execution at this time could lead to all sorts of problems, including the possible corruption of user data.

The I/O Manager maintains a pointer to the current stack location, in addition to the CurrentLocation value mentioned previously. This pointer is maintained in the CurrentStackLocation field in the Tail.Overlay structure that is contained in the IRP header. Kernel-mode drivers should never try to manipulate the contents of either the CurrentLocation or the CurrentStackLocation fields themselves.* The I/O Manager does provide routines for a driver to get a pointer to the current stack location, via a call to IoGetCurrentIrpStackLocation ( ) , to get a pointer to the next stack location using IoGetNextIrpStackLocation ( ) so that the driver can set up the contents of the stack location appropriately for the next driver in the hierarchy, and in rare cases to use IoSetNextIrpStackLocation() to set the stack location value.

The stack location structure defined in the NT DDK is composed of some fields that are independent of the nature of the I/O request being described by the stack location. Here are these fields:

MajorFunction

The NT I/O Manager defines a set of major functions, each of which identifies a generic function that a kernel-mode driver can implement. Functions are identified by function codes or numbers, and the set of functions is deliber-

---

* That being said, it is true that NT file systems themselves perform some underhanded operations on these fields. However, for most kernel-mode drivers, it is far more preferable to stick with the I/O Manager-supplied access methods to view and modify the contents of these fields.

ately comprehensive, since the function codes serve all types of NT kernel-mode drivers, including file system drivers, intermediate drivers, device drivers, and other lower level drivers.

When an IRP is delivered to a kernel-mode driver, the driver must examine the MajorFunction field in the current stack location to find out the functionality expected from the driver. The possible major function codes are shown below:

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

Function codes beginning at IRP_MJ_DEVICE_CHANGE and higher were introduced in NT version 4.0. Also, not all of the major function codes are implemented yet; for example, the quota-related function codes do not yet have any support from native NT file system drivers.

None of the major functions listed above is mandatory for a kernel-mode driver to implement, except for the ability to open and close objects managed by the driver. Open and close operations are very important because, if open operations fail, no I/O requests can be submitted, since there does not exist any object that would be the target of the requests. Similarly, if opens succeed, the close operations will eventually be invoked, and close operations cannot fail (the I/O Manager does not check the return code from a close operation). Therefore, if you do not implement a close operation to complement your open, the system might eventually run out of resources, depending on what operations were previously performed during the open, and also depending on the data structures created during the open operation.

The major function codes in the context of a file system driver and a filter driver are discussed in Part 3-

MinorFunction

Minor function codes provide more information specific to the major function code in the I/O stack location. For example, consider the IRP_MJ_DIRECTORY_CONTROL major function code above. An IRP containing this major function code is sent by the I/O Manager to file system drivers. The intent is to perform some file directory operation. The question, however, is what directory control operation does the I/O Manager want the file system driver to perform?

The available operations include obtaining information about directory contents (IRP_MN_QUERY_DIRECTORY) and notifying the I/O Manager when certain attributes of files or directories contained within the target directorychange(IRP_MN_NOTIFY_CHANGE_DIRECTORY).

Currently, only a few of the major functions have minor functions associated with them. However, for those few, the kernel-mode driver developer must examine this field to correctly determine the functionality it is expected to provide.

Flags

The `Flags` field also provides additional information that qualifies the functionality expected from the target driver. For example, consider the IRP_MJ_DIRECTORY_CONTROL major function code previously discussed. If the minor function is IRP_MN_QUERY_DIRECTORY, the `Flags` field could contain additional information that might cause the file system to behave differently when returning the contents of the directory being queried.

For example, if the SL_RESTART_SCAN flag is set, the file system driver will restart the scan from the beginning of the directory being queried. Or if the SL_RETURN_SINGLE_ENTRY flag is set, the file system driver will return only the first entry matching the specified search criteria.

Lower-level drivers also have an interest in the settings for this flag. For example, removable media drivers will perform a read request dispatched to them from a file system driver if the SL_OVERRIDE_VERIFY_VOLUME flag has been set. If, however, the flag has not been set, and the device driver has recognized a media change (and informed the file system about it), it will fail all I/O requests, including all read requests.

`Control`

When a kernel-mode driver must process an IRP asynchronously, the driver can queue the IRP, mark it "pending" via a call to IoMarkIrpPending() and subsequently return control back to the caller. The call to loMarklrpPending () simply sets the SL_PENDING_RETUKNED flag in the Control field for the current stack location. Any kernel-mode driver can examine the Control field for the existence of this flag.

This flag is also used internally by the NT I/O Manager to store information about whether a completion routine associated with the current stack location should be invoked if the return code supplied at IRP completion indicates a success, a failure, or a cancel operation. These flags are designated as SL_INVOKE_ON_SUCCESS, SL_INVOKE_ON_FAILURE, and SL_INVOKE_ON_CANCEL. Kernel-mode drivers typically should not need to be directly concerned with the state of these flags.

`DeviceObject`

This field is set by the NT I/O Manager as part of the processing performed in the loCallDriver () routine. The contents are set to the device object pointer for the target device object (i.e., the device object to which the IRP is being dispatched).

`FileObject`

The I/O Manager sets this field to point to the file object that is the target of an I/O operation. Note that just calling loAllocatelrpO from your driver will not result in this field being set. If you intend to use the returned IRP for an operation on a specific file object, your driver must set the field itself.

`CompletionRoutine`

The contents of this field are set by the I/O Manager when the loSetCompletionRoutine () macro is invoked. The I/O Manager checks for a completion routine as part of the postprocessing performed during IRP completion. If a completion routine is specified, the routine is invoked in the context of the thread performing the postprocessing; typically this is in the context of the thread that invoked the loCompleteRequest () routine. Since IRP completion is often performed by lower level drivers at a high IRQL, it is quite likely that the completion routine will be invoked at some high IRQL.

You should also note that completion routines are invoked in a *last-specified-first-invoked* order. Therefore, the highest-level driver's completion routine will be invoked after all other completion routines have been invoked. If any driver returns STATUS_MORE_PROCESSING_REQUIRED from an invocation to the driver-supplied completion routine, the I/O Manager immediately stops all postprocessing of the IRP. Freeing the memory for that IRP will then become the responsibility of the driver that returns the STATUS_MORE_PROCESSING_REQUIRED status.

If you develop a higher-level driver, like a file system driver or a filter driver, and if you specify a completion routine, always execute the following sequence of code in your completion routine:

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

If you fail to do this and if there are other drivers layered above yours in the calling hierarchy, the IRP may be processed incorrectly and you could experience a driver or process hang. The reason for the potential hang will be further explained later in this chapter.

Context

This field contains the context supplied by the kernel-mode driver when it specifies a completion routine for the IRP.

If you develop an intermediate driver, you will have to be careful about copying some of the values contained in the current I/O stack location into the next I/O stack location when you prepare to forward the IRP to the next driver in the hierarchy. For example, you must copy the contents of the `Flags` field, so the lower-level driver will know that it should perform an I/O read operation requested by a file system even though it had previously informed the file system about a media change.

Processing an IRP

Handling an IRP sent to your driver can be quite straightforward. The next four figures illustrate some of the common methods employed to handle an IRP dispatched to a kernel-mode driver.

In Figure 4-5, you can see that the target kernel-mode driver receives an IRP, obtains a pointer to the current stack location, performs some processing based on the contents of the I/O stack location, and, finally, completes the I/O request packet. Note, however, that there could be a delay between receiving the request and beginning the processing, since the driver might queue the IRP if it is currently busy processing other requests. The queued IRP would subsequently get processed asynchronously in the context of a worker thread.

Also note that once the driver gets control back from an invocation to `loCompleteRequest ( )`, it must not touch the IRP or any of the fields contained within the IRP again. Doing so could lead to data corruption and system crashes.

Figure 4-6 illustrates how a kernel-mode driver receives an IRP, obtains a pointer to the current stack location, and performs processing based upon the contents of

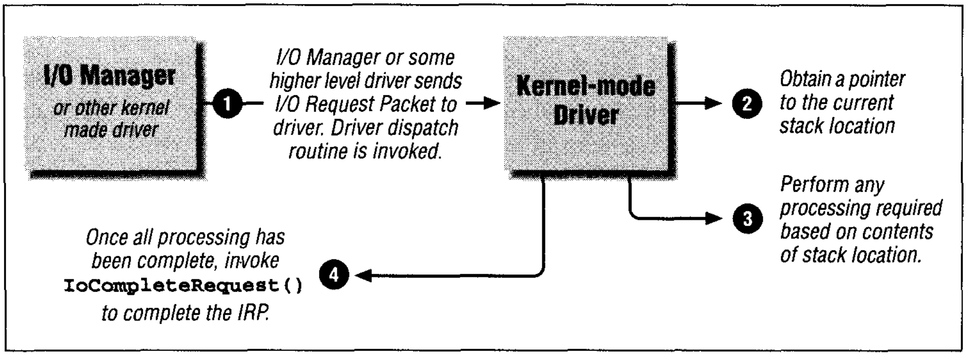
I/O Manager
or other kernel made driver
1
I/O Manager or some higher level driver sends I/O Request Packet to driver. Driver dispatch routine is invoked.
Kernel-mode Driver
2
Obtain a pointer to the current stack location
3
Perform any processing required based on contents of stack location.
4
Once all processing has been complete, invoke IoCompleteRequest() to complete the IRP.

Figure 4-5. Simple IRP processing where invoked driver completes the IRP

the stack location. However, the driver might need to invoke the services of a lower-level driver before the requested functionality is declared completed. Therefore, the recipient of the IRP can initialize the next stack location in the IRP and forward the IRP to the next kernel-mode driver in the layered driver hierarchy.

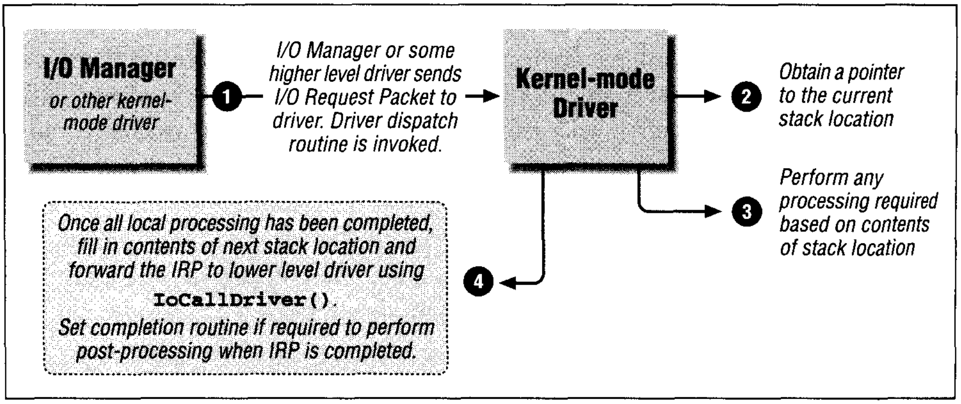
I/O Manager
or other kernel-mode driver
1
I/O Manager or some higher level driver sends I/O Request Packet to driver. Driver dispatch routine is invoked.
Kernel-mode Driver
2
Obtain a pointer to the current stack location
3
Perform any processing required based on contents of stack location
4
Once all local processing has been completed, fill in contents of next stack location and forward the IRP to lower level driver using IoCallDriver().
Set completion routine if required to perform post-processing when IRP is completed.

Figure 4-6. IRP processing where IRP is reused and sent to lower-level driver

If your driver forwards an IRP to another driver, it is no longer allowed to try to access that IRP, since it does not know when the lower-level driver will complete that particular IRP. Typically, forwarding of the IRP is done via a call to `IoCallDriver()`. The I/O Manager will invoke the lower-level driver in the context of the thread that makes the call to `IoCallDriver()`; however, the lower-level driver that now receives the IRP might return STATUS_PENDING and complete the IRP asynchronously.

Figure 4-7 illustrates a sequence where a higher-level kernel-mode driver (e.g., a file system driver) uses associated I/O request packets to issue I/O requests to other lower-level drivers. This might be done if, for example, the higher-level driver wishes to split up an I/O request; it might even be required if the higher-level driver needs processing to be performed by more than one set of lower-level drivers.

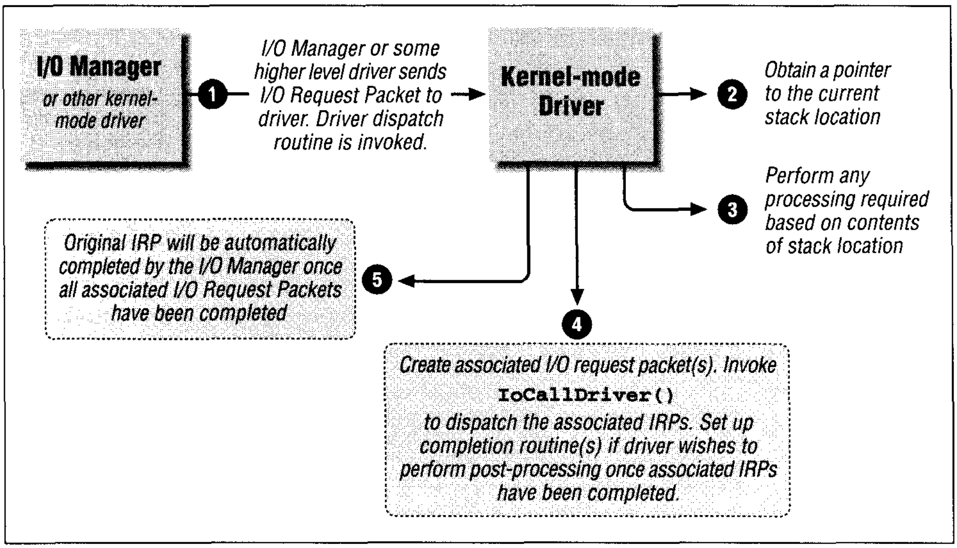
I/O Manager
or other kernel-mode driver
1
I/O Manager or some higher level driver sends I/O Request Packet to driver. Driver dispatch routine is invoked.
Kernel-mode Driver
2
Obtain a pointer to the current stack location
3
Perform any processing required based on contents of stack location
5
Original IRP will be automatically completed by the I/O Manager once all associated I/O Request Packets have been completed
4
Create associated I/O request packet(s). Invoke
IoCallDriver()
to dispatch the associated IRPs. Set up completion routine(s) if driver wishes to perform post-processing once associated IRPs have been completed.

Figure 4-7. Using associated IRP structures to process an IRP

Note that the higher-level driver does not need to invoke loCompleteReguest ( ) on the original IRP; the I/O Manager will automatically complete the original IRP once all associated IRPs have been completed by lower-level drivers. However, the higher-level driver can request that a completion routine be invoked when the associated IRP completes, thereby giving it the opportunity to perform some postprocessing, and also allowing itself the opportunity to complete the original IRP at its own convenience.

Figure 4-8 illustrates a variation of the method using associated IRPs; here the kernel-mode driver uses one of the I/O Manager-supplied functions to create new I/O Request Packets, which are then dispatched to other kernel-mode drivers. Once the newly created I/O Request Packets have been completed, the original IRP can be redispatched to lower-level drivers for further processing, or it can be immediately completed.

IRP completion and deallocation

Every I/O Request Packet must be completed in order for the I/O Manager to be informed that the request contained within the IRP has been completely processed. To complete an IRP, a kernel-mode driver has to invoke the loCompleteRequest ( ) I/O Manager support routine.

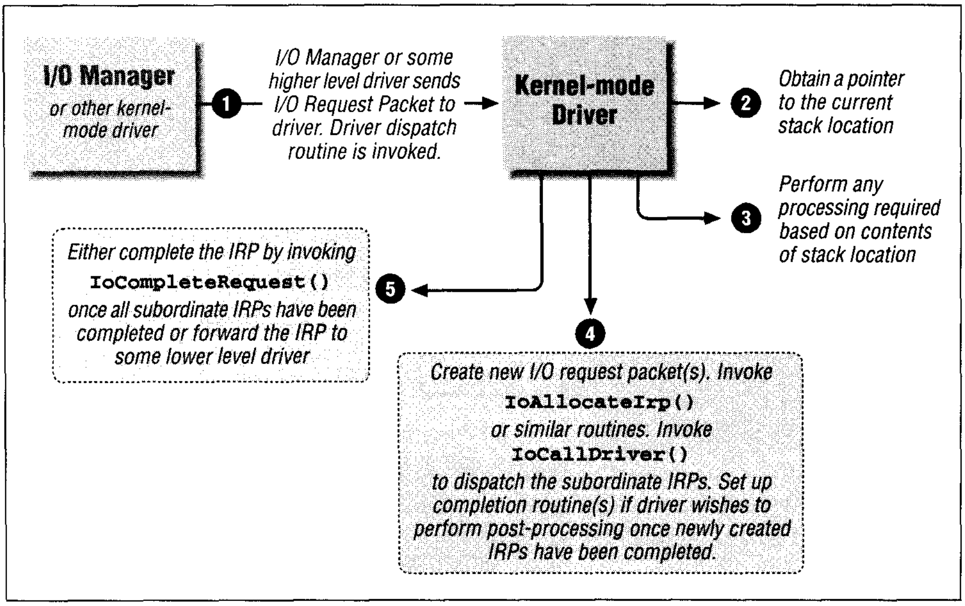
I/O Manager
or other kernel-mode driver
1
I/O Manager or some higher level driver sends I/O Request Packet to driver. Driver dispatch routine is invoked.
Kernel-mode Driver
2
Obtain a pointer to the current stack location
3
Perform any processing required based on contents of stack location
5
Either complete the IRP by invoking IoCompleteRequest() once all subordinate IRPs have been completed or forward the IRP to some lower level driver
4
Create new I/O request packet(s). Invoke IoAllocateIrp() or similar routines. Invoke IoCallDriver() to dispatch the subordinate IRPs. Set up completion routine(s) if driver wishes to perform post-processing once newly created IRPs have been completed.

Figure 4-8. Using newly allocated IRPs to help in processing of an IRP

Once this routine is invoked, the NT I/O Manager performs some postprocessing on the I/O request packet being completed, as follows:

1. The I/O Manager performs some basic checks to ensure that the IRP is in a valid state. The value of the current stack location pointer is verified to ensure that it is less than or equal to (total number of stacks + 1). If the value is not valid, the system will bugcheck with an error code of MULTIPLE_IRP_COMPLETE_REQUESTS. If you install the debug build of the operating system, the I/O Manager will execute some additional assertions, such as checking for a returned status code of STATUS_PENDING when completing the IRP, and checking other invalid return codes from the driver.
2. Now, the I/O Manager starts scanning through all stack locations contained in the IRP looking for completion routines that need to be invoked. Each stack location can have a completion routine associated with it, which should be called depending on whether the final return code was a success or a failure, or if the IRP was canceled. The I/O stack locations are scanned in reverse order, with the highest-valued I/O stack location being checked first. This results in completion routines invoked such that a completion routine supplied by a disk driver (the lowest-level driver) will be invoked first, while the completion routine for the highest-level driver (typically, the file system driver) will be invoked last.

   Completion routines are invoked in the context of the same thread that calls IoCompleteRequest(). If any completion routine returns STATUS_MORE_

PROCESSING_REQUIRED, the I/O Manager immediately stops all further postprocessing and returns control back to the routine that invoked `IoCompleteRequest()`. Now, it is the responsibility of the driver that returned STATUS_MORE_PROCESSING_REQUIRED to invoke `IoFreeIrp()` later.[*]

3. If the IRP being completed is an associated IRP, the I/O Manager will decrement the `AssociatedIrp.IrpCount` field in the master IRP. Then, the I/O Manager invokes an internal routine, `IopFreeIrpAndMdls()`, to free up memory allocated for the associated IRP and also to free any MDL structures allocated for the associated IRP. Finally, if this happens to be the last associated IRP outstanding for the master IRP, the I/O Manager recursively invokes `IoCompleteRequest()` on the master IRP itself.

4. A lot of the postprocessing performed by the I/O Manager occurs in the context of the thread that had originally requested the I/O operation. To do this, the I/O Manager queues a kernel-mode APC, which is subsequently executed in the context of the requesting thread. However, this methodology cannot be employed for certain types of IRP structures, used for the following types of operations:

   Close operations

   An IRP describing a close operation is generated by the I/O Manager and sent to the affected kernel-mode driver whenever the last reference to a kernel-mode object is removed. This might just as well occur while a special kernel-mode APC was already executing. To perform a close operation on objects defined by the I/O Manager, a special internal I/O Manager routine called `IopCloseFile()` is always invoked. `IopCloseFile()` is synchronous and therefore blocking. It allocates and issues a close IRP to the target kernel-mode driver and waits for an event object to complete the close operation. Therefore, when completing an IRP for a close operation, the I/O Manager simply copies over the return status (which incidentally is never checked by the requesting thread for a close operation), signals the event object for which the thread executing `IopCloseFile()` is waiting, and then returns control immediately. The IRP is subsequently deleted in `IopCloseFile()`.

---

[*] The DDK assumes that STATUS_MORE_PROCESSING_REQUIRED will only be invoked by kernel-mode drivers for associated IRPs that they have created. There is nothing, however, that prevents your driver from returning this status code for a normal IRP request that was dispatched to you by the I/O Manager. The problem, though, is that there is a lot of postprocessing required on that IRP that will have been abruptly interrupted due to your returning such a status code from your completion routine. Your driver will then have to devise a method whereby such postprocessing can be resumed later; this is not a trivial task.

Paging I/O requests

Paging I/O requests are issued on behalf of the NT Virtual Memory Manager (VMM). In Chapters 5-8, you will read about the functionality provided by the NT VMM and the NT Cache Manager. For now, simply understand that the I/O Manager cannot afford to incur a page fault while completing a paging I/O request. That would cause a system crash. Therefore, the I/O Manager will do one of two things when completing a paging I/O request:

- For a synchronous paging I/O request, the I/O Manager will copy the returned I/O status into the caller-supplied I/O status block structure, signal the kernel event object for which the caller might be waiting, then free the IRP and return control, since there is no additional postprocessing to be performed.
- For an asynchronous paging I/O request, the I/O Manager will queue a special kernel APC to be executed in the context of the thread that requested paging I/O. This is the Modified Page Writer (MPW) thread, which is a component of the VMM subsystem. In the next chapter you will read a lot more about the MPW thread. For now, it is enough for you to know that the routine that executes in the context of the MPW thread (once the APC has been delivered), copies the status from the paging read operation to the I/O status block provided by the modified page writer, and subsequently invokes an MPW completion routine using another kernel APC.

Later, you will see that the I/O Manager typically frees up any Memory Descriptor Lists that are associated with the IRP, before freeing up the IRP itself. However, for paging I/O operations, the MDL structures that are used belong to the VMM (i.e., they are allocated by the VMM and will therefore be freed only by the VMM upon completion of I/O). That is the reason why the I/O Manager does not free up the MDL structures used in paging I/O requests.

Mount requests

If you examine the flags supplied in the NT DDK, indicating paging I/O requests and mount requests (IRP_PAGING_IO and IRP_MOTJNT_COMPLETION, respectively), you will notice that they are both defined to the same value. This is because the I/O Manager treats the mount request exactly the same as a synchronous, paging I/O read request. Therefore, the I/O Manager performs exactly the same postprocessing for mount requests as described for a synchronous, paging I/O request.

5. If the IRP did not describe either a paging I/O, a close, or a mount request, the I/O Manager next unlocks any locked pages described by Memory

Descriptor Lists (MDLs) associated with the I/O Request Packet. Note that the MDL structures are not freed at this time; they are freed as part of the postprocessing performed in the context of the requesting thread.

6. At this point, the I/O Manager has completed as much of the postprocessing it can, without being in the context of the requesting thread. Therefore, the I/O Manager queues a special kernel APC to the thread that requested the I/O operation. The internal I/O Manager routine that is invoked in the context of the calling thread is called lopCompleteRecjuest ( ). It could happen, however, that there might not be any thread context to send the APC request to. This happens if the thread exited after starting an asynchronous I/O operation, the request had already been initiated by the lower level driver, and the driver could not complete the request within a fixed time-out period. In this scenario, the I/O Manager has given up on the request, and therefore, it simply frees up the memory allocated for the IRP at this point since no further postprocessing can be performed.

   For synchronous I/O operations, the I/O Manager does not queue the special kernel APC but simply returns control immediately at this point. These IRP structures have the IRP_DEFER_IO_COMPLETTON flag set in the Flags field in the IRP. Examples of IRP major functions for which IRP completion can be deferred are directory control operations, read operations, write operations, create/open requests, query file information, and set file information requests. By returning control immediately, the I/O Manager avoids the overhead associated with queuing kernel-mode APCs and the overhead of serving APC interrupts. Instead, the thread that originally requested the I/O operation by invoking loCallDriver () invokes lopCompleteRequest () directly once control is returned to it. This is simply an optimization performed by the NT I/O Manager.

   Note that the I/O Manager will perform two checks to determine whether the APC should be queued or not for the above situation:

   — The IRP_DEFER_IO_COMPLETION flag should be set to TRUE.

   — The Irp->PendingReturned field should be set to FALSE.

   Only if both of the conditions above are TRUE will the I/O Manager simply return from the loCompleteRequest () function at this stage.

   The following situation may result in a problem if you are not careful in your driver:

   — Your driver specifies a completion routine before forwarding a request to a lower-level driver.

- There is a driver layered above you in the calling hierarchy (e.g., a filter/intermediate driver).
- Your driver does not execute the instructions listed earlier about invoking `IoMarkIrpPending()` if `Irp->PendingReturned` is set to TRUE.

Now the I/O Manager may incorrectly believe that an APC should not be queued (thinking that the completion was being performed in the context of the requesting thread) and the original thread will stay blocked forever.

The other situation where the I/O Manager does not queue an APC is if the file object has a completion port associated with it, in which case the I/O Manager sends a message to this completion port instead.

At this time, all processing that could have been performed in `IoCompleteRequest()` is complete.

The remaining steps described below occur in the context of the thread that had originally requested the I/O operation. The NT I/O Manager routine that performs these steps is the `IopCompleteRequest()` routine previously mentioned.

1. For buffered I/O operations, the I/O Manager copies any data returned as a result of the successful execution of the I/O request back into the caller's buffer. Details of buffered I/O operations are provided later in this chapter; however, note for now that if the driver returns an error or if the driver returns a code indicating that a verify operation is required in the IRP `IoStatus` structure, no copy will be performed.*

   Also, the number of bytes copied into the caller's buffer equals the value of the `Information` field in the `IoStatus` structure; therefore, if that field is not set correctly, the caller will not get back all or any of the returned data.

   The I/O Manager-allocated buffer is also deallocated once the copy operation is performed.

2. Any Memory Descriptor Lists associated with the IRP are freed at this time.

---

* You should understand that the NT I/O Manager treats warning status codes as if the operation succeeded; i.e., the I/O Manager will copy data into the caller's buffer even if the status code was not STATUS_SUCCESS, as long as it does not indicate an error.

TIP It is possible for a file system driver to deliberately return a pointer to an MDL allocated by the Cache Manager when requested to do so by a caller for either a read or a write I/O request. Such requests are distinguished by the presence of the IRP_MN_MDL flag in the MinorFunction field of the IRP stack location in the IRP sent to the file system driver. Since all MDLs associated -with an IRP are blindly freed at this point, it appears that there is not much point to a file system driver returning an MDL to the caller. However, currently the only kernel-mode client using the IRP_MN_MDL flag is the LAN Manager Server module, and this module typically circumvents the problem by returning STATUS_MORE_PROCESSING_REQUIRED from a completion routine. See Chapter 9, *Writing a File System Driver I*, for a discussion on how the file system driver processes MDL-read and MDL-write requests.

---

3. The I/O Manager copies the Status and Information fields into the caller-supplied I/O status block structure.
4. If the caller supplied an event object to be signaled, the I/O Manager signals that event object. The I/O Manager signals the event object in the Event field for any file object associated with the I/O Request Packet if either no event object was supplied by the caller or the I/O operation was executed synchronously because the file object was opened for synchronous access only.
5. Typically, the NT I/O Manager increments the reference count of any caller-supplied event object or any file object associated with an IRP before forwarding the IRP to a driver for processing. At this time, the I/O Manager dereferences both of these objects if they had been referenced before.
6. The I/O Manager dequeues the IRP from the list of I/O Request Packets pending for the current thread.
7. Memory for the I/O Request Packet is finally freed; if the I/O Request Packet has been allocated from a zone/lookaside list, memory for that packet is returned to the zone/lookaside list for reuse; otherwise, memory is returned back to the system.

Working ***with I/O requestpackets***

There are a few key concepts that you must understand very well with regard to handling I/O Request Packets sent to your kernel-mode driver:

- Once your driver receives the IRP, no other component in the system, including the I/O Manager, can be concurrently accessing the same IRP. Until your driver either forwards the IRP to another kernel-mode driver, or completes the IRP, processing of the request described by the I/O Request Packet is solely the responsibility of your driver.

- Once your driver completes the IRP, or forwards it to another kernel-mode driver, your driver must give up control of the IRP and not attempt to access any of the fields contained within it again. The only time you can touch that IRP again is if you had specified a completion routine prior to forwarding the IRP. In that case, the I/O Manager will invoke your completion routine as part of its postprocessing performed during IRP completion.
- If you specify a completion routine to be invoked at the time of IRP completion, it can perform any postprocessing necessary. Keep in mind, though, that your completion routine might be called at an IRQL less than or equal to DISPATCH_LEVEL. If your completion routine is invoked at a high IRQL, you cannot incur any page faults while your code is executing. You do have the option of stopping any postprocessing of the IRP by returning STATUS_MORE_PROCESSING_REQUIRED from your completion routine. Be careful, though, when doing this, especially from a lower-level driver, because some of the completion routines specified by other drivers higher in the chain, which normally would be invoked, will now not be called unless you play some tricks with the IRP later.
- No IRP can be completed more than once.* If you do try to do this either deliberately or erroneously, you might cause data corruption and/or system crashes. Although the I/O Manager checks for the possibility that an IRP is being completed more than once, the check is not completely foolproof, so be aware of this requirement when designing your driver.
- Your driver cannot blindly assume that it is being invoked to process an IRP in the context of the thread that originally requested the I/O operation. As a matter of fact, lower-level drivers, such as intermediate drivers and device drivers, will probably never have their dispatch routines invoked in the context of the issuing thread. Therefore, your driver must be careful when trying to access objects, handles, resources, and memory when processing the I/O Request Packet. Understand the context in which your dispatch routines can be invoked and only use resources that are available to you and that are valid in that particular context.
- Kernel-mode drivers have tremendous freedom in what they are able to do. At the same time, the responsibilities that are placed upon kernel-mode code are greater than for user-space applications. If your driver uses pointers to

---

* It is possible for a completion routine to return STATUS_MORE_PROCESSING_REQUIRED, perform some specialized postprocessing with the IRP, and then reissue the IoCompleteRequest() function on the IRP to make the I/O Manager correctly dispose of the IRP. This is the single exception to the rule mentioned above and results in the situation where an IRP is completed more than once.

buffers sent by user-space code, be careful about how you use such buffers. It is possible for kernel-mode drivers to easily compromise system integrity by misusing, or not carefully validating, any buffers and data contained within them, sent by unprotected, user-mode applications. Determine the mode of the caller in deciding whether or not to validate pointers sent to you. Use the previous mode of the caller in making your decision on whether or not to validate user-supplied buffers.

- Use only the I/O Manager-provided access methods to manipulate stack locations in an IRP. It is possible for a kernel-mode driver to modify IRP stack locations, which can affect both how IRP processing is done initially, as well as how IRP postprocessing is performed once the IRP has been completed. Try to resist the temptation to manipulate the contents of the stack locations in any undocumented fashion.
- Use your own I/O Request Packets if you wish to utilize services of other drivers above or below you in the hierarchy. Avoid using private communication channels that are not extensible. To create IRP structures, use one of the I/O Manager-supplied support routines (i.e., IoAllocateIrp(), loBuildSynchronousFsdRequest( ), IoBuildAsynchronou.sFsdRecru.est ( ), loBuildDeviceloControlRequest(), and IoMakeAssociatedIrp()). Use lolnitializelrp (), in conjunction with loAllocatelrpO , to initialize the common fields in the IRP header. Be careful, and reread the previous section to determine which additional fields you might wish to initialize. Also, realize that loFreelrp () may or may not need to be invoked, depending on the status code you return from any completion routine you may have specified.
- Some kernel-mode components, such as the LAN Manager server, allocate I/O Request Packets from internal pools, instead of requesting them from the NT I/O Manager. Be aware that these components may use some of the fields in the IRP in a manner different from the standard manner in which those fields are manipulated by the I/O Manager. Therefore, be careful when depending upon the contents of fields that the I/O Manager wants to keep private and that are not documented in the DDK, since there are no guarantees made by the system that the fields will always contain consistent values.

  Furthermore, components like the LAN Manager server often have a maximum number of stack locations that they typically allocate for an I/O Request Packet. If you add one or more additional filter or intermediate drivers to the driver hierarchy, the number of stack locations required may then exceed the maximum that the LAN Manager server can deal with. There is a workaround to this problem, where you can instruct the user to specify additional stack locations that the LAN Manager server should allocate via a Registry parameter.

Volume Parameter Block (VPB)

The VPB is the link between the file system device object representing the mounted volume and the device object representing the physical or virtual disk that contains the physical file system data structures. Each time a file open request for an on-disk file stream is sent to a device object for a physical or virtual device,* the I/O Manager invokes an internal routine called lopCheckVpbMounted(). This routine is responsible for initiating a logical volume mount operation, if the VPB associated with the physical/virtual device that is the target of the request indicates that the volume has not been mounted. If, however, the volume is previously mounted, the I/O Manager redirects the open operation to the device object whose pointer is obtained using the `DeviceObject` field in the VPB.

Memory for a volume parameter block is automatically allocated from nonpaged pool by the Windows NT I/O Manager when a device object is created through a `loCreateDevice()` call or when a file system driver invokes the `loVerifyVolume()` call, for the following types of device objects:

- `FILE_DEVICE_DISK`
- `FILE_DEVICE_CD_ROM`
- `FILE_DEVICE_TAPE`
- FILE_DEVICE_VTRTUAL_DISK (used for RAM disks or any similar virtual disk structures that can hold a mountable volume)

Note that each of the these types of device objects can have a logical volume present on the device object, and each of these device objects typically also represents a single mountable partition for a device. The volume parameter block is used to map the file system (logical) volume device object to the physical device also represented by a device object. This structure is initially zeroed by the I/O Manager upon allocation. The following definition describes the VPB:

---

* Since the most commonly used subsystem on Windows NT platforms is the Win32 subsystem, consider the case when a user performs a file open operation on a file stream on drive letter *C:*. This drive letter is nothing but a Win32 subsystem-visible name that is actually a symbolic link to a Windows NT name, such as *\Device\HardDiskO\Partitionl*. Therefore, accessing a file stream on *C:* is the same as accessing an on-disk file stream on the physical disk device object with the name *\Device\HardDiskO\Partitionl*. Note that the Windows NT named object is not the device object representing the mounted volume; rather, it is the device object representing the physical/virtual disk drive. The VPB is used to perform the association between the named physical/virtual disk device object and the unnamed logical volume device object.

```
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;
```

Each mounted volume can have a label associated with it with a maximum length of 32 characters. The VolumeLabelLength field is initialized by file system drivers to the actual length of the label for the volume, which is stored in the VolumeLabel field. Each file system volume can also have a serial number associated with it that should be read off the volume by the file system driver and placed in the SerialNumber field. As long as the reference count for the VPB is nonzero, the I/O Manager will not deallocate the VPB structure. The RealDevice field is initialized by the I/O Manager to point to the physical or virtual device object that contains the mountable logical volume. The DeviceObject field is initialized by the file system driver whenever a mount operation takes place. This field contains the address of the device object of type FILE_DEVICE_DISK_FILE_SYSTEM, created by the file system to represent the mounted volume.

The Flags field in the VPB can have one of three values:

VPB_MOUNTED
: This bit is set by the I/O Manager once a file system mounts the logical volume represented by the VPB. This happens after a file system driver returns STATUS_SUCCESS from an IRP sent to it with a major function of IRP_MOUNT_COMPLETION.

VPB_LOCKED
: This field can be set or cleared by the file system driver that has mounted the logical volume represented by the VPB. While this field is set, the NT I/O Manager will fail all subsequent open/create requests targeted to that logical volume. File systems may choose to set this field in response to application requests to lock the logical volume, or if they temporarily wish to prevent any create/open requests from proceeding. The FASTFAT file system responds to application IOCTL requests to lock a volume (FSCTL_LOCK_VOLUME) by setting this field in the VPB.

VPB_PERSISTENT

This field is also manipulated by file system drivers. If this field is set, the I/O Manager will not delete the VPB structure, even if the `ReferenceCount` in the VPB is 0.

The NT I/O Manager provides two routines that should be used by filter drivers and file system drivers to synchronize access to a VPB structure. These support routines are defined as follows:

```
VOID
loAcquireVpbSpinLock(
    OUT PKIRQL        Irql
);

VOID
loReleaseVpbSpinLock(
    IN KIRQL           Irql
);
```

Parameters:

Irql

For the `loAcquireVpbSpinLock()` routine, this is a pointer that, upon return, will contain the IRQL to which the thread must be restored when the corresponding release function is invoked.

For the routine `loReleaseVpbSpinLock()`, this argument contains the IRQL value returned when the spin lock was acquired.

Functionality Provided:

There is a global spin lock structure that is acquired by the I/O Manager internally while manipulating contents of the VPB. If your driver wishes to check or manipulate the `Flags`, `DeviceObject`, or `ReferenceCount` fields in any VPB, you should first invoke the `loAcquireVpbSpinLock()` support routine to ensure data consistency. Note that this is a global spin lock and that, while this spin lock is acquired, not many I/O operations can continue (e.g., new create and open operations will be blocked). Therefore, be careful to acquire the lock only for the short period required while accessing the specified fields.

For more detailed information on the flow of execution leading to a mount operation, as well as for a detailed explanation of handling VPB structures for volumes mounted on removable media, consult Part 3-

I/O Status Block

The I/O Status Block is used to convey the results of an I/O operation. This structure is defined as follows:

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

Every I/O Request Packet (IRP) has an I/O Status Block associated with it. A kernel-mode driver should always insert the return code describing the results of processing the request in the `Status` field in the I/O status block structure. This field will, therefore, contain a return code denoting success (STATUS_SUCCESS), a return code denoting a warning, an informational message, or an error. Error status codes also include those indicating that an exception (which was handled by the driver) occurred while processing an I/O request. Consult the previous chapter for a discussion of the structure of NT return codes.

The `Information` field is typically filled with any additional information related to the requested I/O operation. For example, for a read request of 1024 bytes, the `Information` field upon return will contain the actual number of bytes read even if the Status field indicates STATUS_SUCCESS. Therefore, the `Information` field in this case would contain a value between 0 and 1024 bytes.

File Object

If you develop file system drivers in Windows NT, or if you develop filter drivers that reside above the file system driver in the driver hierarchy, you should become very familiar with the structure of a *file object.* A file object is the I/O Manager's in-memory representation of an open object. For example, if an open operation is successfully performed on an on-disk file, the I/O Manager creates a file object structure to represent that particular instance of the open operation. If another open operation is performed on the same file stream, the I/O Manager will allocate a new file object to represent this second open operation, even though both open operations were performed on the same underlying, on-disk file stream.

You should conceptualize a file object as the kernel equivalent of a *handle* created as a result of a successful open/create request. File objects are not limited to representing open file streams; rather, they are an abstraction used to represent any object opened by the NT I/O Manager. Therefore, if you open a logical volume or a disk drive device object, the open operation will result in the creation and initialization of a file object data structure.

All I/O operations targeted to on-disk file streams or logical volumes require a file object structure as the target for the request (you cannot perform a read request in a vacuum; you must have a target file object representing a previous successful open operation to which you can direct the read operation). The responsibility for creating and maintaining a file object data structure is jointly shared by the NT I/O Manager and the file system driver.

The file object structure is allocated by the I/O Manager before it passes the open or a create request to a kernel-mode file system driver. The create/open IRP contains a pointer to this newly allocated file object structure; it is the responsibility of the kernel-mode file system driver that processes the create/open request to initialize certain fields in the file object structure.

The file object structure is defined by the NT I/O Manager:

```
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;
```

The DeviceObject and Vpb fields in the file object structure are initialized by the I/O Manager before sending a create or an open request to the file system driver. The DeviceObject is initialized to the address of the target physical or virtual device object to which the request is directed. The Vpb field is initialized to the mounted VPB associated with the target device object.

The FsContext, FsContext2, SectionObjectPointer, and PrivateCacheMap fields are initialized and/or maintained by the file system driver implementation and the NT Cache Manager. They will be discussed in greater detail later in this book. The NT I/O Manager does not maintain the contents of these fields, though it does check for and use the contents of the FsContext field; this will be discussed in Part 3.

The FileName field is initialized by the I/O manager to a string representing the file, volume, or physical device to be opened. This name can either be a *relative* pathname or an *absolute* pathname. A relative pathname is indicated by the presence of a nonnull value in the RelatedFileObject field. This field contains a pointer to a previously opened file object data structure. The relative pathname supplied in the FileName field must now be considered relative to the name of the file represented by the RelatedFileObject. Note that the RelatedFileObject field is only valid in the context of a create request. At all other times, the contents of this field are undefined.

The CurrentByteOffset field is maintained by file systems for those file objects that were opened for synchronous access only. This field contains the current pointer position for the file stream, which is updated upon the successful completion of read and/or write I/O operations.

The CompletionContext field is used by the NT I/O Manager to send a message to a Local Procedure Call (LPC) port upon completion of an IRP. The DeletePending flag is set in the file object structure when a file system receives a set information IRP specifying that the file stream should be deleted.

The LockOperation field is set to TRUE by the I/O Manager if the thread that owns the file object structure invoked a byte-range lock operation at least once while the file was open. This field is later checked when the thread closes the file object to determine whether or not to send an unlock IRP to the file system driver.

The various access fields (ReadAccess, WriteAccess, and DeleteAccess) are set and cleared by the I/O Manager. So are the various share access related fields (SharedRead, SharedWrite, and SharedDelete). The state of these fields determines how the file is currently opened and also determines whether subsequent opens requesting certain specific types of access will be allowed to proceed or will be denied with an error code of STATUS_SHARING_VIOLATION. There exists an I/O Manager support routine called loCheckShareAccess (), which maintains the state of these fields. This routine is typically only invoked by file system drivers and will be described later in this book.

Later in this chapter, you will read about synchronous and asynchronous I/O operations from the perspective of the file system drivers that must provide the code to implement such requests. A user can open a file object specifying that all operations performed on the opened object by that particular file object be executed synchronously. This is indicated by the presence of a FO_SYNCHRONOUS_FLAG in the Flags field of the file object structure, which is set by the I/O Manager as part of the create/open request. One of the effects of requesting synchronous I/O operations is that the I/O Manager always serializes all I/O operations performed using that particular file object. To implement this sequential behavior, the NT I/O Manager uses the `Busy` and the `Waiters` fields in the file object data structure, The `Busy` field is set when an I/O operation using that particular file object is in progress. The `Waiters` field denotes the number of threads waiting to perform I/O operations using the same file object. These fields should not be of much interest to other kernel-mode drivers.

The file object is a waitable kernel-mode object, i.e., threads can request asynchronous I/O, and subsequently wait for the completion of the I/O operation. The `Event` field in the file object is used by the I/O Manager to maintain the state of the wait object. This event object is set to the not-signaled state by the I/O Manager when an I/O operation begins using that file object. It is subsequently set to be signaled once the I/O is completed, though only if the caller had not explicitly supplied another event object to wait for.

The `Flags` field can reflect many values, one of has been described here, and each describes a state associated with the file object structure. I will defer discussion of each of the possible values of this field until later in the book, when the field is actually used in our code.

Determining Which Objects to Use

Here are a few simple rules to "put everything together" when developing your driver:

- When your driver loads, a driver object will be created and sent to your initialization routine by the I/O Manager. You must fill in certain fields in the driver object, such as the various dispatch routine function pointers, for the functionality you wish to support. If you do not fill in the function pointers, your driver will not receive any requests, because all requests will be handled by the default routine (`lopInvalidDeviceRequest()`).
- In order to provide any functionality, you will probably create at least one device object. More than likely, you will create one device object representing your driver and subsequent other device objects representing other virtual and/or physical devices you support. Most of the device objects you create will be named, unless you develop a file system driver, in which case, most of the device objects will represent logical volumes and will therefore be unnamed. When requesting a create operation for a device object, you should also specify a device extension in which you can store global data associated with each new device object.
- If you write a filter driver, you will create one device object for each target device object whose I/O requests you wish to intercept. You will then attach

your device object to the target device object. This procedure of attaching to the target will actually cause all I/O requests directed to the target to be rerouted to your device object.*

- If you develop a file system driver, you will have to manipulate the Volume Parameter Block (VPB) for the physical device object on which you perform a *mount* operation. Performing a mount will cause the I/O Manager to make the physical device object accessible for read/write requests and those requests will be sent to your device object representing the mounted logical volume.
- Once you make a device object available for receiving I/O requests, requests will be sent to you in the form of I/O Request Packets (IRPs). If you develop a file system driver, you will also receive requests via the fast path (more on that later in this book).
- When you receive an IRP, you will determine the nature of the I/O operation your driver is being asked to perform. To do this, you should get a pointer to the current stack location in the IRP and use it to extract information pertaining to the I/O request. Your driver will then perform appropriate processing of the IRP, either synchronously or asynchronously.
- Your driver may be able to complete the IRP, or it might determine that the IRP needs to be forwarded to a driver that is lower in the hierarchy for some additional processing. In the latter case, you should obtain a pointer to the next stack location in the IRP and fill in the information that the next driver in the hierarchy can subsequently extract to determine the nature of processing it has to perform.
- If your driver will complete the IRP, it must return results of the I/O operation in the I/O status block structure. The `Status` field should contain the result, while the `Information` field should contain any additional information you wish to return to the caller.
- Last, but not least, if you develop a file system driver, you will access and possibly modify the file object structure as part of processing an open request (and subsequently when processing most IRPs). Each such structure represents an instance of a successful open operation.

In addition to the objects mentioned in this chapter, if you develop a device driver, you will be concerned with other objects as well, including controller objects, adapter objects, and interrupt objects.

Furthermore, your driver will undoubtedly create one or more object types of its own. For example, file system drivers will create some internal representation of a

---

* The process of attaching to a target device object is described in detail in Chapter 12.

file stream in memory. For those familiar with UNIX operating system environments, think about the *vnode* structure that is created and maintained by all file systems. The NT equivalent of this structure is a *File Control Block*, an object we will discuss at length in Part 3. In addition, file systems will create a context to internally represent an instance of a file open operation (similar to the system-defined file object structure). In Windows NT parlance, this structure is called a *Context Control Block.*

Once you start using these objects in your code development, they should become second nature to you and you will no longer have to spend time trying to figure out what a device object represents.

I/O Requests: A Discussion

The following discussion provides some additional information that you should keep in mind as we develop a higher-level kernel-mode driver. This information will be used not only in the sample drivers provided in this book, but also in any commercial kernel-mode drivers you design and develop.

Synchronous/Asynchronous Operations

Some I/O operations are always performed synchronously; therefore, any file system driver that you develop only has to design a synchronous method of processing IRPs for such types of requests. Other operations can be handled either synchronously or asynchronously; your file system driver must, therefore, provide both synchronous and asynchronous code paths for processing such I/O request packets.

How does a kernel-mode driver determine whether an IRP should be handled synchronously or asynchronously?

Before we address that question, it might be useful to see why handling asynchronous requests correctly is important. Consider a file system driver that you design that does not honor asynchronous requests but performs all requests synchronously. Your implementation should work correctly most of the time. The one problem that might occur is when your driver receives asynchronous paging I/O write requests. These requests typically originate from the NT Modified Page Writer. The number of worker threads available to the Modified Page Writer is fixed. It may be that the MPW uses only two threads to perform such paging I/O, one to the page files and the other to memory-mapped files.

In low-memory and high-stress situations, the VMM tries to quickly flush modified pages out to secondary storage to make room for other data in the system memory. The MPW does this by rapidly issuing asynchronous page write requests to file systems that manage one or more of the modified pages, either in mapped files or in page files. If your driver blocks the MPW thread until the I/O is completed, it slows down the whole process of flushing data out to disk, which can result in unacceptably long delays to the users of the system.

Therefore, if you develop a higher level kernel-mode driver, it would be prudent to provide support for asynchronous I/O operations.

Only some I/O system services can be processed asynchronously:

- Read requests
- Write requests
- Directory control requests
- Byte range lock/unlock requests
- Device I/O control requests
- File system I/O control requests

As you may have noticed, all of the types of requests listed above can potentially take a significant amount of time to complete. Therefore, it is logical that the caller be allowed to request asynchronous processing for such requests. All of the other IRP major functions should complete reasonably quickly.

Therefore, if your file system or higher-level filter driver (layered above a file system) receives an IRP with a major function other than the ones listed here, you can assume that you are allowed to block in the context of the calling thread.

For the major functions listed, the caller has the option of specifying whether the request should be performed synchronously or asynchronously. To find out what the caller wants, your kernel-mode driver can invoke the following I/O Manager support routine:

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

Parameters:

Irp

> The I/O request packet sent to your driver. This IRP has flags set by the I/O Manager that determine whether the IRP can be processed synchronously or asynchronously. Note that asynchronous operations can always be performed synchronously (with the slight caveat discussed above); however, even if your driver performs a synchronous operation asynchronously and therefore returns STATUS_PENDING to the I/O Manager, the NT I/O Manager will perform a wait operation in the kernel on behalf of the calling thread.

Functionality Provided:

This simple function call performs the following checks:

- If the IRP_SYNCHRONOUS_IRP flag has been set, the IRP should be executed synchronously. All IRP structures that describe major functions other than the ones listed above will have this flag set in the IRP. The presence of this flag causes `IoIsOperationSynchronous` () to return TRUE.
- As described earlier in this chapter, the caller may have opened the target file object for synchronous access only. This is denoted by the F0_SYNCHRONOUS_IO flag being set in the file object data structure; the presence of this flag causes the loIsOperationSynchronous () routine to return TRUE.
- The IRP may be a paging I/O read or write request, denoted by the IRP_PAGING_IO flag in the IRP. Furthermore, even paging I/O requests can be synchronous or asynchronous. Synchronous paging I/O requests are indicated by the presence of the IRP_SYNCHRONOUS_PAGING_IO flag in the IRP. If the latter flag is not set, the I/O Manager knows that this is an asynchronous paging I/O request and returns FALSE; otherwise, the I/O Manager identifies the request as a synchronous paging I/O request and returns TRUE.

The NT I/O Manager provides different methods of informing callers when asynchronous I/O operations have been completed. Here are the possible methods:

- The file object structure is a waitable object in Windows NT. When an I/O operation is initiated on a file object, the object is initially set to the not-signaled state; when the I/O operation completes, the file object is set to the signaled state.
- The asynchronous NT system services provided by the I/O Manager accept an optional Event object that is initially set to the not-signaled state and is signaled when the I/O operation is completed. In the discussion on IRP completion, I mentioned that the I/O Manager signals a user-supplied event object when performing the final postprocessing upon IRP completion in the context of the calling thread. Note, however, that if an event object is supplied, the file object will not be signaled.
- Asynchronous NT system services provided by the I/O Manager also accept an optional caller-supplied APC routine. This routine is invoked via an Asynchronous Procedure Call by the I/O Manager as part of the postprocessing performed in the context of the calling thread.

One final note about synchronous requests; all synchronous requests made using the same file object structure are serialized, regardless of whether they are made by the same thread or by other threads that are part of the same process. The file system driver also has the responsibility of maintaining a current position pointer for each file object that is updated whenever a file object is opened for synchronous I/O.

Handling User-Space Buffer Pointers

When you create a device object that can receive and serve I/O requests, your driver gets the opportunity to specify how it will handle user-supplied buffer address pointers. You won't fully understand why this information is necessary until you read the next chapter on the NT Virtual Memory Manager. For now, however, note that the range of addresses that a user-mode thread can access is limited to the lower 2GB of the 4GB address space accessible to any process under Windows NT. Furthermore, this 2GB range of virtual address space is unique per process (i.e., the addresses used by *thread-A* do not necessarily refer to the same physical memory location as do similar addresses used by *thread-B).* Of course, threads belonging to the same process do share the same address space.

A user-mode application typically performs I/O to and from secondary storage using temporary buffers it has allocated in its own thread context. We will currently ignore the alternative method used by applications, which involves using shared memory or memory-mapped files.

For example, consider an application that needs to read some data for a file from disk. This application will typically allocate a buffer that should be large enough to contain the amount of requested data. The application will then invoke a read operation on the open file from which it wishes to obtain data, specifying the byte offset to read from and the amount of information to be read.

The read request from the application will eventually be translated into an NT system service call provided by the NT I/O Manager. Among the arguments received by the I/O Manager will be the pointer to the buffer, supplied by the user-mode application. This read request now is sent by the I/O Manager to the file system driver that manages the mounted logical volume on which the open file object resides.* It is at this point that the I/O Manager finds out how the file system driver will deal with the user-supplied buffer pointer. This buffer is valid only in the context of the user-mode thread and does not refer to locked (nonpaged) memory. The file system can choose from the following possible options:

' As you go through the rest of the book, you will find out that this statement is not completely true, since often the I/O Manager bypasses the file system driver completely and instead gets data directly from the system cache. Let us keep things simple and straightforward for now, though, and ignore that method of data transfer.

- Request that the I/O Manager always allocates a nonpaged system buffer that will subsequently be used by the file system driver in the data transfer. It would then be the responsibility of the I/O Manager to copy any data being written out to disk from the user-supplied buffer to the system buffer before dispatching the IRP to the file system driver. Similarly, for I/O operations where the user-mode application needs to obtain information from the file system driver or to read data from disk, the I/O Manager would have to copy the data back from the system buffer to the user-allocated buffer once the IRP had been completed.

  • This method of handling user-mode buffers by instructing the I/O Manager to always allocate a corresponding system buffer is called the *Buffered I/O* method.

  The system buffer pointer is passed down to your driver in the `AssociatedIrp.SystemBuffer` field in the IRP. Note that the I/O Manager will also often initialize the `UserBuffer` field in the IRP with the address of the caller-supplied buffer. Do not attempt to use the contents of this field in your kernel-mode driver, though, because the `SystemBuffer` field already contains the system buffer pointer you can use.

  The disadvantage of using buffered I/O is the requirement for extra memory copies to be performed by the I/O Manager. This is not desirable when you wish to maximize system performance. However, buffered I/O is the simplest and therefore most widely utilized method of handling user-supplied buffers.

  Another disadvantage of using the buffered I/O method is that the memory for the system buffer allocated by the I/O Manager is not paged. This results in unnecessary depletion of the nonpaged pool of memory reserved for the system. A third problem is that, although the memory is not paged out, if you wish to use Direct Memory Access to transfer data directly to/from memory and peripheral devices, a Memory Descriptor List will have to be created by either your driver or a lower-level driver to describe the physical pages that back the allocated buffer.

- If your driver wishes to avoid the overhead of allocating and copying data to and from a system buffer, you can instead specify that your driver will use the *direct I/O* method. If this method is specified, the I/O Manager will request an MDL from the VMM that describes the user buffer directly, and it will also request the VMM to allocate and lock physical pages for the user buffer. The resulting MDL pointer will be passed to your driver in the `MdlAddress` field in the IRP.

  The direct I/O method is more efficient than the allocation of an extra buffer and the resulting copy operations that must be performed. The downside is that your driver must be capable of working with the MDL directly; i.e., there is no virtual address pointer that your driver can use when transferring data.

  Now, this works fine when you simply pass the MDL down to a lower-level driver, which subsequently uses it in a DMA data transfer. However, if you need a virtual address pointer that is accessible in the context of the thread you process the IRP in, your driver will have to use the MmGetSystemAddressForMdl ( ) support routine from the VMM. You must be careful when using this routine; freeing the Memory Descriptor List will cause all processors in the system to flush their caches. The reason for this is complex; simply stated though, obtaining a system virtual address for the MDL is done by *doubly mapping* the physical pages. This is also known as *aliasing,* a technique which, if not handled correctly, causes many cache consistency problems and resulting headaches for the VMM. If your driver does use the direct I/O method for handling user-supplied buffers, try to avoid using the MmGetSystemAddressForMdl ( ) routine whenever possible.

- The third method is not to specify either direct I/O or buffered I/O as the preferred method for handling user-supplied buffers. If you do not specify either of these two methods, the I/O Manager will simply pass down the user address to your driver in the `UserBuffer` field in the IRP.

  The responsibility for manipulating the user buffer is on your driver if you choose this method. File system drivers often use this method, and then make a decision in their dispatch routines whether they will create an MDL themselves or internally allocate a system buffer they can use while processing the request. Most lower-level drivers, however, prefer to use the direct I/O method described above.

These methods do not apply to buffers passed in for device or file system IOCTL (I/O Control) requests. I will discuss IOCTL requests and the buffer manipulation performed by the I/O Manager for such requests in Part 3.

System Boot Sequence

Before you proceed to the remaining chapters in this book, it might be useful to understand the steps that are executed from the time you power-on your Windows NT system until the point where you see the logon screen on the console.

This information can prove quite useful when you design your driver, because it determines when your driver will be loaded and what part your driver might be called upon to play during this process. However, you should also note that the boot process is highly system-, processor-, operating-system-version-, and architecture-dependent, and the sole objective in listing some of the steps below is simply to provide you with generic information about "what really happens" when the system boots, not to prepare you to be able to adapt the boot sequence to a new processor architecture.* Therefore, be warned that the following description is highly simplified, though mostly correct.

The main problem in examining the system boot sequence is to determine the starting point. For the purposes of this section, our "beginning" will be the point at which code provided by Microsoft as part of the NT operating system gets executed:

1. The NT system startup routine is invoked by the system start-up module. This routine is passed a BootRecord structure, which contains basic machine and environment information used later by the OS Loader component.

   The NT system startup routine performs some global initialization and determines the disk drive and partition that the system is booting from. Part of the global initialization involves initializing memory descriptors for use during this initial system boot-up stage. The system startup routine also invokes a boot loader heap initialization routine, which sets up memory descriptors appropriately so that the boot loader can subsequently use that memory during the system load process.

   The boot sequence described so far comprises Phase 1 of the eight phases in the NT system boot process.

2. The boot loader startup routine is now invoked by the system startup routine. Note that system startup routine does not expect that the call to the boot loader startup routine will ever return, since that would indicate that system boot sequence has failed. However, if this does happen, you will probably see a hung system, where a hard power reset might be required to restart.

   The boot loader startup routine opens the boot partition, which had been previously identified by the caller, and reads the *boot.ini* file off it. As part of attempting to read this file, the boot loader startup routine uses code that has been compiled in to determine whether the boot partition contains an NTFS, CDFS, FAT, or HPFS partition. Note that the standard file system drivers have not been loaded yet, and the boot loader startup routine uses hardcoded support for only those file systems that Microsoft has chosen to provide boot support for; these happen to be the standard NT file system implementations. Since support for boot file systems has to be built into the NT boot loader startup code, providing a third-party bootable file system implementation is close to impossible without the active assistance of Microsoft.

---

* I have described the sequence that executes on the x86 processor architecture. Despite my warnings above, much of the code executed during system startup has been designed to be relatively portable across different architectures; therefore, the methodology and principles used are pretty much the same.

At this point, the boot loader startup routine makes a real-mode BIOS interrupt call to set the video adapter to 80*50, 16-color, alphanumeric mode. It also clears the display by writing blanks out to the screen.

The boot loader startup routine reads the entire contents of the *boot.ini* file and presents the list of bootable kernels available to the user, as listed in the *boot.ini* file. To read the file, the boot loader startup routine once again employs routines that can recognize NTFS, FAT, CDFS, and HPFS data structures, and can navigate successfully through the on-disk file system layout. If the *boot.ini* file is empty, the default option presented is NT (default) and the default directory path to boot from is *C:\winnt**

The boot loader startup routine now attempts to match the default boot location provided by the user in the `[boot loader]` section of the *boot.ini* file, with the options read from the `[operating systems]` section of the file. If no default option was specified, the default directory path is searched for. If the boot loader startup routine does not find a match between the default boot option and those options listed in the `[operating systems]` section, the default boot location chosen is *C:\winnt.*

The default boot location and the possible options are presented by the boot loader startup routine to the user using video display support routines. If the boot kernel path location selected by the user is *C:\*, the NT loader startup code assumes that the user wishes to boot into DOS, Windows 3.x, Windows 95, or OS/2; therefore, it attempts to read in the *bootsect.dos* file and then reboots the machine into whichever alternative operating system is present.

If the boot location indicates that the user wishes to boot into Windows NT (this can happen because of a time-out in the selection process, or because of the user selecting a specific boot system), the boot loader startup routine attempts to read in the *ntdetect.com* executable from the root directory of the boot partition. If *ntdetect.com* is not found, or if the size of the file seems incorrect, or if any of the other consistency checks made by the OS loader startup code fail, the boot process will fail and you will have to reboot the system. If, however, a valid executable is found, it is read into memory, and the system attempts to use the services provided by the hardware manufacturer to detect the current hardware configuration.

Note that we are well into Phase 2 of the system boot initialization at this point. The OS loader startup routine now initializes the SCSI boot driver if required. The *ntldr.exe* OS loader is now loaded into memory.

---

* Note that the boot loader startup routine currently has a bug in that it cannot handle more than 10 entries in the *boot.ini* file. All entries exceeding this limit are simply ignored. Apparently, this bug has existed since Windows NT Version 3.5 (and probably since well before that).

3. The OS loader opens the console input and output devices, and also the system and boot partitions. It also displays the OS loader identification message on the console, `OS Loader V4.0`.

   The loader uses the boot partition information to generate a complete pathname for the *ntoskrnl.exe* NT kernel system image file. Note that the system always expects to find this file in the *System32* directory under the boot partition location. Once the system image has been loaded into memory, the OS loader loads into memory the *hal.dll* system file. The HAL (Hardware Abstraction Layer) isolates platform dependent functionality for the rest of the Windows NT Executive.

   At this point, all DLLs imported by the two loaded system files are identified and loaded into memory. Now, the OS loader attempts to load the *SYSTEM* hive from the NT Registry. At this time, the loader has already made the determination whether it should load the LastKnownGood control set or the *Default* control set from the Registry. This determination is important because the control set determines the set of boot drivers that will be loaded into the system.

   To load the *SYSTEM* hive into memory, the OS loader attempts to open and read the *SYSTEM* file from the *System32\config* directory on the boot partition. If the attempt to open and read in the *SYSTEM* file fails, an attempt is made to read in the *SYSTEM.ALT file.* If neither of these attempts succeed, the OS loader fails the boot attempt. If the file can be successfully read, the contents of the file are verified, and in-memory data structures are initialized to reflect the contents of the on-disk file. Also, note that the system loader block, which is eventually passed to the loaded system image, is appropriately modified to point to the in-memory copy of the *SYSTEM hive.*

   At this time, the OS Loader determines the list of boot drivers that need to be loaded into memory. Included among this list is the driver responsible for the boot partition file system. Note that boot drivers are identified by the `Start` value entry (should be equal to 0) associated with the driver's key in the control set that was loaded into memory. Once the list of boot drivers has been identified, the OS Loader sorts these drivers based upon the `ServiceGroupOrder` specified in the Registry; subsequent drivers within a group are sorted based on the GroupOrderList specified in the Registry and the Tag value entry associated with each driver key in the Registry.

   Once the driver load order has been determined, all boot drivers are loaded. In the event of an error while loading boot drivers, the `ErrorControl` value entry associated with the driver in the Registry is examined. If the driver was marked as a critical driver for the system boot process, the current boot fails; otherwise, the OS loader continues loading other boot drivers.

Finally, the OS Loader prepares to execute the loaded system image and transfers control to the entry point in the NT kernel.

4. During Phases 3-5 of the system boot process, the various NT Executive components and the NT kernel are initialized. Drivers that should be automatically loaded with a *Start* value entry of 1 are also loaded during Phase 5 of the system boot process.

   The NT kernel initialization routine, `KiInitializeKernel ()`, is invoked during Phase 3 initialization by the kernel system startup routine (which is the entry point into the system image that was loaded into memory by the NT OS loader). This routine initializes the processor control block data structure, the kernel data structures, and the idle thread and process objects, and invokes the NT Executive initialization routine. Various spin locks protecting kernel data structures and kernel linked list structures are initialized here. The various kernel linked list heads (DPC queue list head, timer notification list head, various thread table lists, and other similar kernel data structures) are also initialized here.

   Once the kernel idle thread structure has been initialized, the Executive initialization routine is now invoked in the context of this idle thread. Initialization of the NT Executive and the various subcomponents of the Executive takes place in two phases.* During Phase 0 of the Executive initialization, the following subcomponents initialize their internal states:

   — The Hardware Abstraction Layer (HAL)

   — The NT Executive component

   — The Virtual Memory Manager (VMM)

      Memory Manager paged and nonpaged pools, the page frame database (explained in the next chapter), Page Table Entry (PTE) management structures, and various VMM resources, such as mutex and spin lock data structures, are initialized at this time. The VMM also initializes the NT system-cache-related data structures at this time, including the system cache working set and the various VMM data structures used to manage the system cache.

   — The NT Object Manager

   — The Security subsystem

---

* Do not confuse these phases with the system boot sequence Phases 1 through 8. These two phases are internal and specific to the initialization of the NT Executive and its various subcomponents.

— The Process Manager

During Phase 0 initialization of the NT Executive, the initial system process is created. Note that the idle process was hand crafted by the NT kernel before any of the Executive initialization began. The system process created at this time is distinct from the idle process that was created earlier. A system thread is also created in the context of the initial system process at this time. Phase 1, or the remainder of the NT Executive initialization, is now performed in the context of this newly created thread belonging to the initial system process.

During Phase 1 initialization of the NT Executive and various subcomponents, all interrupts are disabled and the priority of the thread in whose context the initialization is performed is raised to a high priority, effectively disabling any preemption. Also, during Phase 1 of the Executive initialization, the system is considered fully functional and subcomponents are now allowed to perform all required operations to complete their initialization. The following subcomponents are invoked (or their operations performed) during Phase 1 initialization of the NT Executive:

— The Hardware Abstraction Layer (HAL) is invoked to complete initialization.

— The system date and time are initialized.

— On an multiprocessor system, other processors are started at this time.

— The Object Manager, Executive subsystem, and the Security subsystem are invoked to perform the remainder of their initialization.

— The Virtual Memory Manager (VMM) Phase 1 initialization is performed.

At this time, the memory mapping functionality is initialized and becomes available to the rest of the system. VMM threads are also started now. The VMM can be considered fully functional and ready to service the remainder of the system after Phase 1 initialization.

— The NT Cache Manager is initialized after the VMM initialization has been completed.

You will read more about the NT Cache Manager and the functionality provided by it later in this book. Note for now, that during Cache Manager initialization, the number of worker threads required for asynchronous operations is determined and created, and the Cache Manager linked list structures and synchronization resources are initialized.

— The Configuration Manager is invoked to begin its initialization.

The Configuration Manager manages the NT Registry. During this phase of initialization, the Configuration Manager (CM) makes available the

\REGISTRY\MACHINE\SYSTEM and the *\REGISTRY\MACHINE\HARDWARE hives* in the registry. To do this, all of the information obtained by *ntdetect.com* earlier, as well as information read into memory by the OS loader is filled into appropriate entries in the *SYSTEM* or *HARDWARE* hives. Once this phase of initialization has been completed, part of the Registry name space is available to other system components, particularly kernel-mode drivers that will soon be loaded; however, the CM will not write out modifications to the Registry at this time. The kernel-mode drivers that will soon be called upon to perform driver-specific initialization can use the standard Registry routines to access this information.

— The NT I/O Manager is called upon to perform its initialization.

The I/O Manager first initializes internal state objects, including synchronization data structures, linked lists, and memory pools (e.g., the IRP zone/ lookaside lists). Then, the I/O Manager registers all of its internally defined object types (i.e. adapter objects, controller objects, device objects, driver objects, I/O completion objects, and file objects) with the Object Manager using an internal routine called ObCreateObjectType ( ) .* The I/O Manager also creates the *\Device*, *\DosDevices*, and the *\Driverroot* directories in the object name-space at this time.

Next, the boot drivers loaded by the OS loader are initialized by the I/O Manager. This includes invoking the *driver entry* routines for each of these drivers to perform driver-specific initialization. The *raw file system* driver is also loaded at this time. The only other file system driver loaded is the *boot file system* driver. Drivers must adhere to the restrictions on interacting with the NT Registry. Finally, the drivers with a *Start* value entry value of 1 are loaded and their driver entry routines invoked for driver-specific initialization.

Driver reinitialization routines are subsequently invoked for all loaded drivers that have requested reinitialization. Following this, the NT I/O Manager assigns drive letters to recognized disk partitions. The drive letters *A:* and *B:* are reserved for floppy drives. The I/O Manager examines the registry for any "sticky" drive letter assignments that need to be maintained for CD-ROM drives and for hard disk drive partitions. These drive-letter assignments are internally reserved so that they will not be used subsequently when determining dynamic DOS drive letter assignments.

' Note that the Object Manager is not aware of these object types otherwise (i.e., information about I/O Manager-defined objects is not coded into the Object Manager design). This illustrates the philosophy of a layered and object-based system, followed by the NT development team.

Note that before reserving a drive letter for each of the hard disk drives, the I/O Manager performs an open operation on the physical drive. Therefore, if you develop a hard disk device driver or a lower-level filter or intermediate driver, you should expect an open request at this time. If the open request succeeds, a symbolic link to the device object is created in NT object name space; the name assigned to this link is of the form *\DosDevices\PhysicalDrive°/od* where *%d* represents the disk drive number in sequence. The NT I/O Manager also queries partition information from the disk driver at this time. The following method is used by the I/O Manager to determine the order in which drive letters are assigned to fixed disk partitions:

— The NT I/O Manager queries the Registry for any "sticky" drive letter assignments that need to be maintained.

— Bootable partitions are first assigned dynamic DOS-compatible drive letters (i.e., a symbolic link is created to the device object representing the partition, with the name *\DosDevices\%c:* where *%c* represents the drive letter chosen by the NT I/O Manager for the partition).

— Primary partitions are next chosen for dynamically assigned drive letters.

— Extended partitions are subsequently assigned DOS drive letters.

— Other (enhanced) partitions are now assigned DOS drive letters. After drive letters have been assigned to hard disk drive and removable drive partitions, the NT I/O Manager assigns drive letters to all CD-ROM drives that were identified during hardware detection.

— The Local Procedure Call (LPC) subsystem and the Process Manager subsystem now complete their initialization.

— The Reference Monitor and Session Manager subsystem are invoked next to complete their initialization.

5. At this point, Phases 3-5 of the system boot process have been completed. Remember that NT Executive components were initialized in the context of the system worker thread belonging to the system process, which was created by the NT kernel. This thread now assumes the role of the Memory Manager *zero page thread;* this is a very low-priority thread used to asynchronously zero out pages that are placed on the free list by the VMM. As you will read in the next chapter, all pages need to be zeroed out before they can be reused to make the system conform to *C2* security defined by the US Department Of Defense (DOD).

6. The system has been initialized at this point. During Phases 6-8 of the system boot process, the various subsystems are initialized and other services are loaded by the Service Controller Manager. This includes the loading of kernel-mode drivers with a *Start* value of 2 in the Registry.

We have finished our bird's-eye view of the NT system boot process. This should have given you a reasonable understanding of the steps executed to bring the NT system to a stable state so that it can begin responding to user requests.

This chapter has introduced you to the Windows NT I/O Manager and the Windows NT I/O subsystem. Another important component of the NT Executive is the NT Virtual Memory Manager, which is the topic of the next chapter.