I

Overview

Part I introduces the Windows NT Operating System and some of the issues of file system driver development.

- Chapter 1, *Windows NT System Components*
- Chapter 2, *File System Driver Development*
- Chapter 3, *Structured Driver Development*

In ibis chapter:
- *The Basics*
- *The Windows NT Kernel*
- *The Windows NT Executive*

1

Windows NT System Components

The focus of this book is the Windows NT *file system* and the interaction of the file system with the other core operating system components. If you are interested in providing value-added software for the Windows NT platform, the topics on *filter driver* design and development should provide you with a good understanding of some of the mechanics involved in designing such software.

File systems and filter drivers don't exist in a vacuum, but interact heavily with the rest of the operating system. This chapter provides an overview of the main components of the Windows NT operating system.

The Basics

Operating systems deal with issues that users prefer to forget, including initializing processor states, coordinating multiple CPUs, maintaining CPU cache coherency, managing the local bus, managing physical memory, providing virtual memory support, dealing with external devices, defining and scheduling user processes/threads, managing user data stored on external devices, and providing the foundation for an easily manageable and user-friendly computing system. Above all, the operating system must be perceived as reliable and efficient, since any perceived lack of these qualities will almost certainly result in the universal rejection and thereby in the quick death of the operating system.

Contrary to what you may have heard, Windows NT is not a state-of-the-art operating system by any means. It employs concepts and principles that have been known for years and have actually been implemented in many other commercial operating systems. You can envision the Windows NT platform as the result of a confluence of ideas, principles, and practices obtained from a wide variety of

sources, from both commercial products and research projects conducted by universities.

Design principles and methodologies from the venerable UNIX and OpenVMS operating system platforms, as well as the MACH operating system developed at CMU, are obvious in Windows NT. You can also see the influence of less sophisticated systems, such as MS-DOS and OS/2. However, do not be misled into thinking that Windows NT can be dismissed as just another conglomeration of rehashed design principles and ideas. The fact that the designers of Windows NT were willing to learn from their own experiences in designing other operating systems and the experiences of others has led to the development of a fairly stable and serious computing platform.

The Core Architecture

Certain philosophies derived from the MACH operating system are visible in the design of Windows NT. These include an effort to minimize the size of the kernel and to implement parts of the operating system using the client-server model, with a message-passing method to transfer information between modules. Furthermore, the designers have tried to implement a layered operating system, where each component interacts with other layers via a well-defined interface.

The operating system was designed specifically to run in both single-processor and symmetric multiprocessor environments.

Finally, one of the primary goals was to make the operating system easily portable across many different hardware architectures. The designers tried to achieve this goal by using an object-based model to design operating system components and by abstracting out those small pieces of the operating system that are hardware-dependent and therefore need to be reimplemented for each supported platform; the more portable components can, theoretically, simply be recompiled for the different architectures.

Figure 1-1 illustrates how the Windows NT operating system is structured. The figure shows that Windows NT can be broadly divided into two main components: user mode and kernel mode.

User mode

The operating system provides support for protected subsystems. Each protected subsystem resides in its own process with its memory protected from other subsystems. Memory protection support is provided by the Windows NT Virtual Memory Manager.

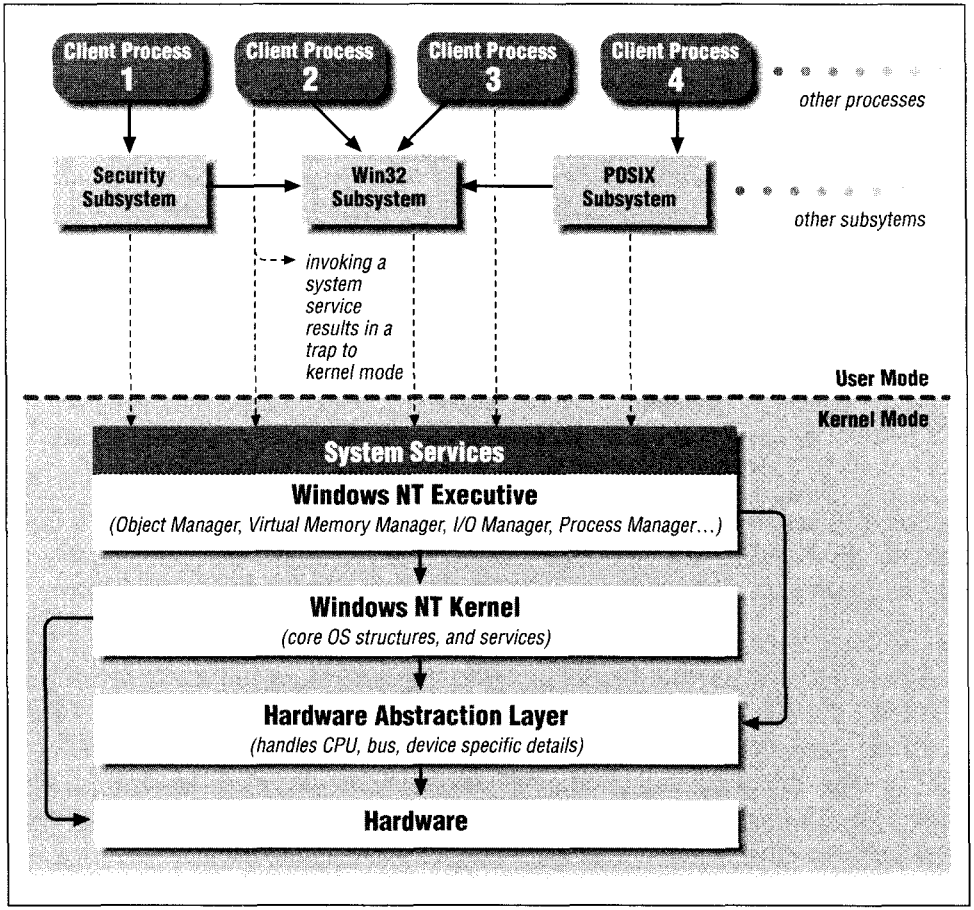

Figure 1-1. Overview of the Windows NT operating system environment

The subsystems provide well-defined Application Programming Interfaces (APIs) that can be used by user-mode processes to obtain desired functionality. The subsystems then communicate with the kernel-mode portion of the operating system using well-defined system service calls.

NOTE Microsoft has never really documented the operating-system-provided system-service calls. They instead encourage application developers for the Windows NT platform to use the services of one of the subsystems provided by the operating system environment.

By not documenting the native Windows NT system service APIs, the designers have tried to maintain an abstract view of the operating system. Therefore, applications only interact with their preferred native subsystem, leaving the subsystem to interact with the operating system. The benefit to Microsoft of using this philosophy is to tie most applications to the easily portable Win32 subsystem (the subsystem of choice and, sometimes, the subsystem of necessity), and also to allow the operating system to evolve more easily than would be possible if major applications depended on certain specific native Windows NT system services.

However, it is sometimes more efficient (or necessary) for Windows NT applications and kernel-mode driver developers to be able to access the system services directly. In Appendix A, *Windows NT System Services,* you'll find a list of the system services provided by the Windows NT I/O Manager to perform file I/O operations.

Environment subsystems provide an API and an execution environment to user processes that emulates some specific operating system (e.g., an OS/2 or UNIX or Windows 3.x operating system). Think of a subsystem as the *personality* of the operating system as viewed by a user process. The process can execute comfortably within the safe and nurturing environment provided by the specific subsystem without having to worry about the capabilities, programming interfaces, and requirements of the core Windows NT operating system.

The following environment subsystems are provided with Windows NT:

Win32

The native execution environment for Windows NT. Microsoft actively encourages application developers to use the Win32 API in their software to obtain operating system services.

This subsystem is also more privileged than the others.* It is solely responsible for managing the devices used to interact with users; the monitor, keyboard, and mouse are all controlled by the Win32 subsystem. It is also the

* In reality, this is the only subsystem that is actively encouraged by Microsoft for use by third-party application program designers. The other subsystems work (more often than not) but seem to exist only as checklist items. If, for example, you decided to develop an application using the POSIX subsystem instead, you will undoubtedly encounter limitations and frustrations due to the very visible lack of commitment on behalf of Microsoft in making the subsystem fully functional and full featured.

sole Window Manager for the system and defines the policies that control the appearance of graphical user interfaces.

POSIX

This exists to provide support to applications conforming to the POSIX 1003.1 source-code standard. If you have applications that were developed to use the APIs defined in that standard, you should theoretically be able to compile, link, and execute them on a Windows NT platform.

There are severe restrictions on functionality provided by the POSIX subsystem that your applications must be prepared to accept. For example, no networking support is provided by the POSIX subsystem.

OS/2

Provides API support for 16-bit OS/2 applications on the Intel x86 hardware platform.

WOW (Windows on Windows)

This provides support for 16-bit Windows 3.x applications. Note, however, that 16-bit applications that try to control or access hardware directly will not execute on Windows NT platforms.

VDM (Virtual DOS Machine)

Provided to support execution of 16-bit DOS applications. As in the case of 16-bit Windows 3.x applications, any process attempting to directly control or access system hardware will not execute on Windows NT.

Integral subsystems extend the operating system into user space and provide important system functionality. These include the user-space components of the Security subsystem (e.g., the Local Service Authority); the user-space components of the Windows NT LAN Manager Networking software; and the Service Control Manager responsible for loading, unloading, and managing kernel-mode drivers and system services, among others.

Kernel mode

The difference between executing code in kernel mode and in user mode is the hardware privilege level at which the CPU is executing the code.

Most CPU architectures provide at least two hardware privilege levels, and many provide multiple levels. The hardware privilege level of the CPU determines the possible set of instructions that code can execute. For example, when executing in user mode, processes cannot directly modify or access CPU registers or page-tables used for virtual memory management. Allowing all user processes access to such privileges would quickly result in chaos and would preclude any serious tasks from being performed on the CPU.

Windows NT uses a simplified hardware privilege model that has two privilege levels: *kernel mode,* which allows code to do anything necessary on the processor;* and *user mode,* where the process is tightly constrained in the range of allowed operations.

If you're familiar with the Intel x86 architecture set, kernel mode is equivalent to the *Ring 0* privilege level for processors in the set and user mode to *Ring 3-*

The terms kernel mode and user mode, although often used to describe code (functions), are actually privilege levels associated with the processor. Therefore, the term *kernel-mode code* simply means that the CPU will always be in kernel-mode privilege level when it executes that particular code, and the term *user-mode code* means that the CPU will execute the code at user-mode privilege level.

Typically, as a third-party developer, you cannot execute Windows NT programs while the CPU is at kernel-mode privilege level unless you design and develop Windows NT kernel-mode drivers.

The kernel-mode portion of Windows NT is composed of the following:

The Hardware Abstraction Layer (HAL)

The Windows NT operating system was designed to be portable across multiple architectures. In fact, you can run Windows NT on Intel x86 platforms, DEC Alpha platforms, and also the MIPS-based platforms (although support for this architecture has recently been discontinued by Microsoft). Furthermore, there are many kinds of external buses that you could use with Windows NT, including (but not limited to) ISA, EISA, VL-Bus, and PCI bus architectures. The Windows NT developers created the HAL to isolate hardware-specific code. The HAL is a relatively thin layer of software that interfaces directly with the CPU and other hardware system components and is responsible for providing appropriate abstractions to the rest of the system.

The rest of the Windows NT Kernel sees an idealized view of the hardware, as presented by the HAL. All differences across multiple hardware architectures are managed internally by the HAL. The set of functions exported by the HAL are invoked by both the core operating system components (e.g., the Windows NT Kernel component), and device drivers added to the operating system.

The HAL exports functions that allow access to system timers, I/O buses, DMA and Interrupt controllers, device registers, and so on.

---

* Code that executes in kernel mode can do virtually anything with the system. This includes crashing the system or corrupting user data. Therefore, with the flexibility of kernel-mode privileges comes a lot of responsibility that kernel-mode designers must be aware of.

The Windows NT Kernel

The Windows NT Kernel provides the fundamental operating system functionality that is used by the rest of the operating system. Think of the kernel as the module responsible for providing the building blocks that can subsequently be used by the Windows NT Executive to provide all of the powerful functionality offered by the operating system. The kernel is responsible for providing process and thread scheduling support, support for multiprocessor synchronization via spin lock structures, interrupt handling and dispatching, and other such functionality.

The Windows NT Kernel is described further in the next section.

The Windows NT Executive

The Executive comprises the largest portion of Windows NT. It uses the services of the kernel and the HAL, and is therefore highly portable across architectures and hardware platforms. It provides a rich set of system services to the various subsystems, allowing them to access the operating system functionality.

The major components of the Windows NT Executive include the Object Manager, the Virtual Memory Manager, the Process Manager, the I/O Manager, the Security Reference Monitor, the Local Procedure Call facility, the Configuration Manager, and the Cache Manager.

File systems, device drivers, and intermediate drivers form a part of the I/O subsystem that is managed by the I/O Manager and are part of the Windows NT Executive.

The Windows NT Kernel

The Windows NT Kernel has been described as the heart of the operating system, although it is quite small compared to the Windows NT Executive. The kernel is responsible for providing the following basic functionality:

- Support for kernel objects
- Thread dispatching
- Multiprocessor synchronization
- Hardware exception handling
- Interrupt handling and dispatching
- Trap handling
- Other hardware specific functionality

The Windows NT Kernel code executes at the highest privilege level on the processor.* It is designed to execute concurrently on multiple processors in a symmetric multiprocessing environment.

The kernel cannot take page faults; therefore, all of the code and data for the kernel is always resident in system memory. Furthermore, the kernel code cannot be preempted; therefore, context switches are not allowed when a processor executes code belonging to the kernel. However, all code executing on any processor can always be interrupted, provided the interrupt level is higher than the level at which the code is executing.

IRQ Levels

The Windows NT Kernel defines and uses Interrupt Request Levels (IRQLs) to prioritize execution of kernel-mode components. The particular IRQL at which a piece of kernel-mode code executes determines its *hardware priority.* All interrupts with an IRQL that is less than or equal to the IRQL of the currently executing kernel-mode code are masked off (i.e., disabled) by the Windows NT Kernel. However, the currently executing code on a processor can be interrupted by any software or hardware interrupt with an IRQL greater than that of the executing code. IRQLs are hierarchically ordered and are defined as follows (in increasing order of priority):

PASSIVE_LEVEL
: Normal thread execution interrupt levels. Most file system drivers are asked to provide functionality by a thread executing at IRQL PASSIVE_LEVEL, though this is not guaranteed. Most lower-level drivers, such as device drivers, are invoked at a higher IRQL than PASSIVE_LEVEL.

  This IRQL is also known as LOW_LEVEL.

APC_LEVEL
: Asynchronous Procedure Call (APC) interrupt level. Asynchronous Procedure Calls are invoked by a software interrupt, and affect the control flow for a target thread. The thread to which an APC is directed will be interrupted, and the procedure specified when creating the APC will be executed in the context of the interrupted thread at APC_LEVEL IRQL.

DISPATCH_LEVEL
: Thread dispatch (scheduling) and Deferred Procedure Call (DPC) interrupt level. DPCs are defined in Chapter 3, *Structured Driver Development.* Once a

* The highest privilege level is defined as the level at which the operating system software has complete and unrestricted access to all capabilities provided by the underlying CPU architecture.

thread IRQL has been raised to DPC level or greater, thread scheduling is automatically suspended.

Device Interrupt Levels (DIRQLs)
Platform-specific number and values of the device IRQ levels.

`PROFILE_LEVEL`
Timer used for profiling.

`CLOCK1_LEVEL`
Interval timer clock 1.

`CLOCK2_LEVEL`
Interval timer clock 2.

IPI_LEVEL
Interprocessor interrupt level used only on multiprocessor systems.

`POWER_LEVEL`
Power failure interrupt.

`HIGHEST_LEVEL`
Typically used for machine check and bus errors.

APC_LEVEL and DISPATCH_LEVEL interrupts are software interrupts. They are requested by the kernel-mode code and are lower in priority than any of the hardware interrupt levels. The interrupts in the range CLOCK1_LEVEL to HIGH_ LEVEL are the most time-critical interrupts, and they are therefore the highest priority levels for thread execution.

Supportfor Kernel Objects

The Windows NT Kernel also tries to maintain an object-based environment. It provides a core set of objects that can be used by the Windows NT Executive and also provides functions to access and manipulate such objects. Note that the Windows NT Kernel does not depend upon the Object Manager (which forms part of the Executive) to manage the kernel-defined object types.

The Windows NT Executive uses objects exported by the kernel to construct even more complex objects made available to users.

Kernel objects are of the following two types:

Dispatcher objects
These objects control the synchronization and dispatching of system threads. Dispatcher objects include thread, event, timer, mutex, and semaphore object types. You will find a description of most of these object types in Chapter 3.

Control objects

These objects affect the operation of kernel-mode code but do not affect dispatching or synchronization. Control objects include APC objects, DPC objects, interrupt objects, process objects, and device queue objects.

The Windows NT Kernel also maintains the following data structures:

Interrupt Dispatcher Table

This is a table maintained by the kernel to associate interrupt sources with appropriate Interrupt Service Routines.

Processor Control Blocks (PRCBs)

There is one PRCB for each processor on the system. This structure contains all sorts of processor-specific information, including pointers to the thread currently scheduled for execution, the next thread to be scheduled, and the idle thread.

NOTE Each processor has an idle thread that executes •whenever no other thread is available. The idle thread has a priority below that of all other threads on the system. The idle thread continuously loops looking for work such as processing the DPC queue and initiating a context switch whenever another thread becomes ready to execute on the processor.

Processor Control Region

This is a hardware architecture-specific kernel structure that contains pointers to the PRCB structure, the Global Descriptor Table (GOT), the Interrupt Descriptor Table (IDT), and other information.

DPC queue

This global queue contains a list of procedures to be invoked whenever the IRQL on a processor falls below IRQL DISPATCH_LEVEL.

Timer queue

A global timer queue is also maintained by the NT Kernel. This queue contains the list of timers that are scheduled to expire at some future time.

Dispatcher database

The thread dispatcher module maintains a database containing the execution state of all processors and threads on the system. This database is used by the dispatcher module to schedule thread execution.

In addition to the object types mentioned above, the Windows NT Kernel maintains device queues, power notification queues, processor requester queues, and other such data structures required for the correct functioning of the kernel itself.

Processes and Threads

A *process* is an object* that represents an instance of an executing program. In Windows NT, each process must have at least one *thread* of execution. The process abstraction is composed of the process-private virtual address space for the process, the code and data that is private to the process and contained within the virtual address space, and system resources that have been allocated to the process during the course of execution.

Note that process objects are not schedulable entities in themselves. Therefore you cannot actually schedule a process to execute. However, each process contains one or more schedulable threads of execution.

Each thread object executes program code for the process and is therefore scheduled for execution by the Windows NT Kernel. As noted above, more than one thread can be associated with any process, and each thread is scheduled for execution individually.

The context of a thread object consists of user- and kernel-stack pointers for the thread, a program counter for the thread indicating the current instruction being executed, system registers (including integer and floating-point registers) containing state information, and other processor status maintained while the thread is executing.

Each thread has a scheduling state associated with it. The possible states are *initialized, ready-to-run, standby, running, waiting,* and *terminated.* Only one thread can be in the *running* state on any processor at any given instant, though multiple threads can be in this state on multiprocessor systems (one per processor).

Threads have execution priority levels associated with them; higher priority threads are always given preference during scheduling decisions and always preempt the execution of lower priority threads. Priority levels are categorized into the *real-time* priority class and the *variable* priority class.

---

* The Windows NT Kernel defines the fundamental thread and process objects. The Windows NT Executive uses the core structures defined by the kernel to define Executive thread and process object abstractions.

NOTE It is possible to encounter situations of *priority-inversion* on Windows NT systems, where a lower-priority thread may be holding a critical resource required by a higher-priority thread (even a thread executing with real-time priority). Any thread that is of higher-priority than the one holding the critical resource would then get the opportunity to execute even if it has a priority lower than that of the thread waiting for the resource.*

The scenario described above violates the assumption that higher priority threads will always preempt and execute before any lower priority threads are allowed to execute. This could lead to incorrect behavior, especially in situations where thread priorities *must* be maintained (e.g., for real-time processes). Kernel-mode designers must anticipate and understand that these situations can occur unless they ensure that resource acquisition hierarchies are correctly defined and maintained.

Windows NT does not provide support for features such as *priority inheritance* that could automatically help avoid the priority inversion problem.

Most kernel-provided routines for programmatically manipulating or accessing thread or process structures are not exposed to third-party driver developers.

Thread Context and Traps

A *trap* is the processor-provided mechanism for capturing the context of an executing thread when certain events occur. Events that cause a trap include interrupts, exception conditions (described in Chapter 3), or a system service call causing a change in processor mode from user mode to kernel mode of execution.

When a trap condition occurs, the operating system *trap handler* is invoked.t The Windows NT trap handler code saves the information for an executing thread in the form of a *call frame* before invoking an appropriate routine to process the trap condition. Here are two components of a call frame:

A trap frame

This contains the volatile register state.

* Priority inversion requires three threads to be running concurrently: the *high-priority* thread that requires the critical resource, the *low-priority* thread that has the resource acquired, and the *intermediate-priority* thread that does not want or need the resource and therefore gets the opportunity to preempt the low-priority thread (because it has a higher relative priority) but also (in the process) prevents the high-priority thread from executing even though it has a relatively lower priority.

t The trap handler is written in assembly, is highly processor- and architecture-specific, and is a core piece of functionality provided by the Windows NT Kernel.

An exception frame
When exception conditions occur that cause the trap handler to be invoked, the nonvolatile register state is also saved.

In addition, the trap handler also saves the previous machine state and any information that will allow the thread to resume execution after the trap condition has been processed appropriately.

The Windows NT Executive

The Windows NT Executive is composed of distinct modules, or subsystems, each of which assumes responsibility for a primary piece of functionality. Typically, references to Windows NT kernel-mode code actually refer to modules in the Executive.

The Executive provides a rich set of system service calls (an API) for subsystems to access its services. In addition, the Executive also provides comprehensive support to developers who wish to extend the existing functionality. Development is usually in the form of third-party device drivers, installable file system drivers, and other intermediate and filter drivers used to provide value-added services.

The various components that comprise the Windows NT Executive maintain more or less strict boundaries around themselves. Once again, the object-based nature of the operating system manifests itself in the prolific use of abstract data types and methods. Modules do not directly access the internal data structures of other modules; note that, although the designers have managed to stick to well-defined interfaces internally, modules still make many assumptions when they invoke each other. The assumptions are often in the form of *expectations* of what processing the called module will perform and how error conditions will be handled and/or reported. Finally, as you will observe later in this book, the synchronization hierarchy employed by the Executive components when they recursively invoke each other is more than just a little complicated.

The Windows NT Object Manager

All components of the Windows NT Executive that export data types for use by other kernel-mode modules use the services of the Object Manager to define, create, and manage the data types, as well as instances of the data types.

The NT Object Manager manages objects. An object is defined as an opaque data structure implemented and manipulated by a specific kernel-mode (Executive) component. Each object may have a set of operations defined for it; these include

operations to create an instance of the object, delete an instance of the object, wait for the object to be signaled, and signal the object.

The Object Manager provides the capabilities to do the following:

- Add new object types to the system dynamically (note that the Object Manager does not concern itself with the internal data structure of the object).
- Allow modules to specify security and protection for instances of the object type.
- Provide methods to create and delete object instances.
- Allow the module defining an object type to provide its own methods (such as methods for create, close, and delete operations) to manipulate instances of object types.
- Provide a consistent methodology to maintain references of instances of the object type.
- Provide a global naming hierarchy based upon the more commonly used file system hierarchy *inverted-tree* format.

The Object Manager maintains a global name space for Windows NT. All named objects in the system can be accessed via this name space. The object name space is modeled on normal filenaming conventions. Therefore, there is a global root directory named "\" created by the Object Manager during system initialization. Executive components can create directories and subdirectories under the root directory and then create instances of defined object types under any such directory. Whenever an object is created or inserted (even for file-system-defined objects such as files and directories), parsing of the object name begins at the root of the Object-Manager-maintained name space. If an object type has a *parse method* defined for it (as for example, file objects representing open file system files and directories), the Object Manager invokes the parse method for the object. Chapter 2, *File System Driver Development,* provides additional information on how the Object Manager handles requests to open or create on-disk file or directory objects.

The object type structure maintained by the NT Object Manager contains information such as the type of memory pool from which instances of the object type should be allocated, the valid access types for the object, pointers to procedures associated with the object (these are optional and could include pointers to create, open, close, delete, and other such procedures), and some synchronization structure maintained by the Object Manager for all object instances of the particular type.

Each object instance has a standard object header and an object-type-specific object body associated with it. The standard object header contains information

such as pointers to the name of the object (if any), a security descriptor associated with the object (if any), the access mode for the object, reference counts for the object, a pointer to the object type (to which the object instance belongs), and other attributes associated with the object.

Whenever a thread successfully opens an instance of a particular object type, the NT Object Manager returns to the requesting thread an opaque *handle* to the object instance. Note that there can be more than one handle to any object instance at any given point in time. For example, object handles can potentially be inherited.

The Object Manager maintains information associated with each object handle, including a pointer to the object instance, the access information for the open operation, and other attributes for the handle. Note that there is no direct relationship between the handle and the pointer to the open instance of the object type. The handle is typically an index into an object table, which is composed of an array of object table entries.

WARNING Handles are specific to a process. Therefore, if a thread successfully performs a create and open operation and obtains a handle in return, all threads for the particular process can use that handle.

However, if the same handle *is* used in the context of a thread associated -with any other process, you will receive an error code indicating that the handle is invalid.*

Other Windows NT Executive Components

As mentioned earlier, the other major components of the Windows NT Executive are as follows:

The Process Manager

This component is responsible for the creation and deletion of processes and threads. It uses the services provided by the Windows NT Kernel to perform tasks such as suspending an executing process, resuming execution of a process, providing process information, and so on.

---

* Although this may not make sense to you yet, this error is a leading cause of frustration to driver developers who open a resource in their DriverEntry () routine and then try to use the returned handle in some other dispatch routine, which is typically executed in the context of another thread (and process).

The Local Procedure Call (LPC) facility

This facility is the mechanism by which messages can be passed between two processes on the same node.* The client process typically passes parameters to a server process and requests some services. In return, it may receive some processed data back from the server process.

The client's call to the server is intercepted by a stub in the client process that packages the parameters being sent to the server procedure. Then the LPC facility provides the mechanism for the client process to transmit the data to the server and then wait for a response back from the server. This is done using a *Port* object, defined and created by the LPC facility.

The LPC facility is modeled on the Remote Procedure Call mechanism used to implement the client-server model across machines connected by a local or wide area network. The LPC facility is better optimized for communication within a node where all processes have access to the same physical memory.

Security Reference Monitor

This module is responsible for enforcing security policy on the local node. It also provides object auditing facilities.

Virtual Memory Manager

The Windows NT Virtual Memory Manager (VMM) manages all available physical memory on the local node. It is also responsible for providing virtual memory management functionality to the rest of the operating system, as well as to all applications that execute on the node.

Almost all kernel-mode and user-mode modules must interact with the Virtual Memory Manager component. Most modules are clients of the Virtual Memory Manager and therefore depend on the VMM to provide memory management services. File systems, however, are special, because they must often provide services to the VMM (e.g., for reading or writing page files). File system designers must understand thoroughly the interactions of file system drivers with the VMM module. The VMM is discussed in greater detail in Chapter 5, *The NT Virtual Memory Manager.*

Cache Manager

The Windows NT Executive contains a dedicated caching module to provide virtual block caching functionality (in system memory) for file data stored on secondary storage media. The Cache Manager uses the services of the Windows NT Virtual Memory Manager to provide caching functionality. All of the native NT file system driver implementations use the services of the Cache

* A single node can be defined as a computer containing either a single processor or multiple processors. Multiple nodes can potentially be networked together to create a Windows NT cluster.

Manager. The Windows NT Cache Manager is discussed in detail in Chapters 6-8.

I/O Manager

The Windows NT I/O Manager defines and manages the framework within which all kernel-mode drivers (including file system, network, disk, intermediate, and filter drivers) can reside. The I/O Manager is described in detail in Chapter 4, *The NTI/O Manager*.