Windows NT Architecture

Mark Russinovich

(Reprinted from WindowsItPro Magazine)

Windows NT's architecture influences everything from its API to its performance. In the late 1980s, Microsoft charged NT's developers with creating a new operating system, and the company mandated a hefty list of requirements to make NT the world's dominant desktop and enterprise-level operating system. NT's developers faced the constraints of supporting backward compatibility with DOS and Windows 3.x, as well as supporting a laundry list of capabilities intended to ensure NT's long-term success. What NT's developers produced was an operating system that made use of 1980s cutting-edge technologies but had roots in earlier operating systems. NT met Microsoft's broad requirement list, and that fact positioned NT to become widely adopted, no matter which of the popular operating system API sets, processor types, or network interfaces won market dominance.

This month I provide the first part of a two-part primer on NT architecture. I'll describe some of the design requirements that were goals for NT from the start. Then I'll outline in broad strokes the components that make up NT's base operating system and describe how they fit together. I conclude this month with a close look at NT operating system environments and system services. Next month I'll take an in-depth look at the NT Executive, Kernel, and hardware abstraction layer (HAL).

A Brief History of NT's Development

During NT's development period, from 1988 to 1993, the computing world was different from how it is today. DOS was the predominant PC operating system, and both Windows and OS/2 were gaining momentum. Servers and scientific and engineering workstations ran UNIX exclusively. Because all these operating systems were popular, Microsoft built support in NT for DOS, Windows 3.x, OS/2, and POSIX. This support created an upgrade and compatibility mode in NT for DOS and Windows users, and also enabled OS/2 and POSIX users to migrate to NT.

Microsoft realized that although NT's DOS and Windows 3.x support made its PC customers happy, enterprise-level (which at that time meant 32-bit) customers were more interested in the POSIX and OS/2 32-bit APIs. Microsoft saw that if it wanted to capture enterprise-level customers, it would have to develop its own 32-bit API. Thus, Win32, Microsoft's answer to OS/2 and POSIX, became NT's primary API.

Throughout NT's development period, most PCs used Intel's x86 processor. Several RISC processors competed for dominance of UNIX boxes: IBM and Motorola's PowerPC, MIPS processors, and Digital's Alpha. To keep its PC customer base, as well as to accommodate the desires of high-end users, Microsoft decided to make NT as portable as possible, and the company designed NT to run out of the box on any of the RISC processor chips. Microsoft reasoned that NT's portability across different processors would ensure its viability regardless of which chip came out on top in the market.

Windows 3.1 had no native networking support, and Windows 3.11's networking capabilities were cumbersome and relatively slow. As a result, Microsoft felt Novell NetWare's sting. Microsoft intended to avoid repeating these networking mistakes, and it outfitted NT with support for most of the APIs and networking protocols that were in widespread use in the 1980s. Those APIs included NetBIOS, remote procedure call (RPC), file server and redirector (Server Message BlockSMB), mail slots, named pipes, and Berkeley sockets. The protocols included TCP/IP, NetBEUI (Microsoft's LanManager protocol), IPX/SPX (Novell's NetWare protocol), AppleTalk Data Link Control (DLC), and SNA. By including protocols in NT that its competitors owned (i.e., AppleTalk and IPX/SPX), Microsoft opened the door to sites that were dominated by Macintosh or NetWare.

In addition to these high-level requirements, Microsoft included two important low-level operating system capabilities in NT. First, NT's developers designed its security subsystem as a centralized

module that can be easily and thoroughly validated. This security configuration earned NT a C2 security rating.

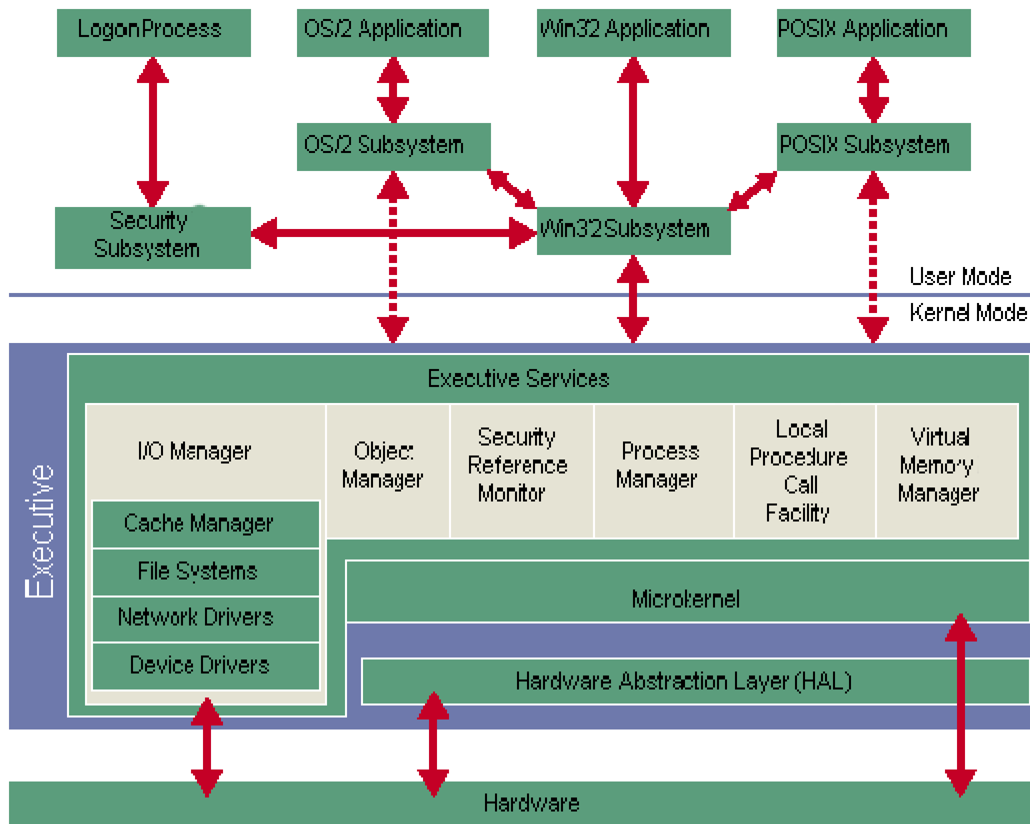
Second, its developers gave NT a multitasking preemptive-scheduling system. Neither DOS nor Windows 3.x is capable of true multitasking with preemptive scheduling. Without add-on software, DOS can execute only one program, or task, at a time. Windows 3.x can execute several programs concurrently, but they must be well-behaved; that is, each program must be aware that other programs may need to run, and it must therefore yield the machine at regular intervals. This design means that a buggy or malicious program can halt the computer simply by entering an infinite loop in which it never yields. In NT, a centralized scheduling authority doles out CPU time to programs that need it. Once a program's turn has ended, the scheduler has the power to preempt it and give another program a turn.

Microsoft wanted NT to incorporate one final major feature: It had to be truly 32-bit and provide protected address spaces, à la UNIX. DOS and Windows 3.x are 16-bit operating systems. Programs running on them cannot easily access large amounts of memory. Using 32-bit addressing enables programs on NT to access 4GB (232 bytes) of memory efficiently. (A 64-bit version of NT is due for release within 2 years, and it will let programs address even larger amounts of memory efficiently.)

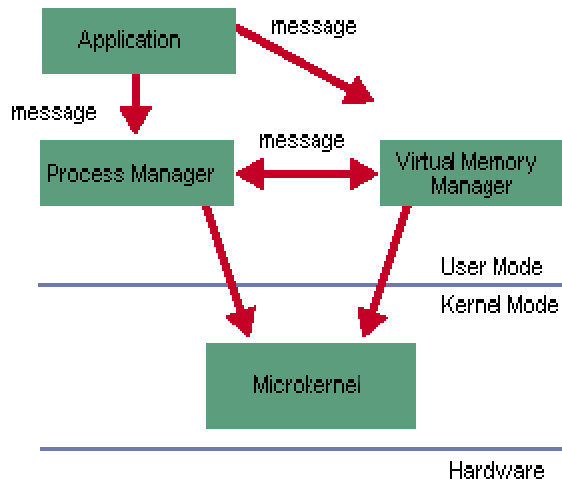
NT's developers ensured 32-bit system reliability by including protected address spaces in NT. Every program in Windows 3.x has a region of memory assigned to it. However, any program can scribble on the memory regions that belong to any other program--a program can even scribble on memory regions reserved for Windows, with disastrous effects. But with the protected address spaces in NT, all programs are confined to their memory regions and have no access (unless by permission) to the memory spaces of other applications. NT also prevents applications from accessing parts of memory owned by the Executive and by kernel-mode portions of the operating system, including device drivers.

An Overview of NT Architecture

Let's begin our look at NT's architecture by discussing the distinction between user mode and kernel mode. I discussed the user mode/kernel mode concept in "Inside the Blue Screen," December 1997, and I'll summarize it here (Figure 1 shows this architecture). User mode is the least-privileged mode NT supports; it has no direct access to hardware and only restricted access to memory. For example, when programs such as Word and Lotus Notes execute in user mode, they are confined to sandboxes with well-defined restrictions. They don't have direct access to hardware devices, and they can't touch parts of memory that are not specifically assigned to them. Kernel mode is a privileged mode. Those parts of NT that execute in kernel mode, such as device drivers and subsystems such as the Virtual Memory Manager, have direct access to all hardware and memory.

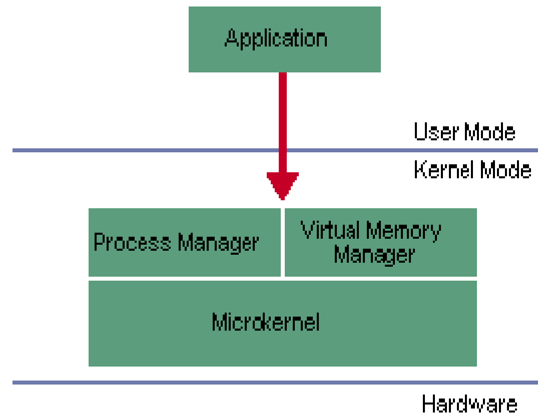


Other operating systems, including Windows 3.1 and UNIX, also use privileged and nonprivileged modes. What makes NT unique is where it draws the line between the two. NT is sometimes referred to as a microkernel-based operating system. Microkernel-based operating systems developed from university research in the mid-1980s. The idea behind the pure microkernel concept is that all operating system components except a small core (the microkernel) execute as user-mode processes, just as word processors and spreadsheets do. But the core components in the microkernel execute in privileged mode, so they access hardware directly. Figure 2, page 64, shows a pure microkernel operating system design.



Microkernel architecture gives a system configurability and fault tolerance. Because an operating system subsystem like the Virtual Memory Manager runs as a distinct program in microkernel design,

a different implementation that exports the same interface can replace it. If the Virtual Memory Manager fails, thanks to the fault-tolerance possible in a microkernel design, the operating system can restart it with minimal effect on the rest of the system. In monolithic operating system design (e.g., DOS and Windows 3.1), the entire operating system must be rebuilt to change any subsystem. Figure 3, page 64, shows a monolithic operating system design. If the Virtual Memory Manager has a bug in a monolithic system, the bug is likely to bring down the machine.



A disadvantage to pure microkernel design is slow performance. Every interaction between operating system components in microkernel design requires an interprocess message. For example, if the Process Manager requires the Virtual Memory Manager to create an address map for a new process, it must send a message to the Virtual Memory Manager. In addition to the overhead costs of creating and sending messages, the interprocess message requirement results in two context switches: the first from the Process Manager to the Virtual Memory Manager, and the second back to the Process Manager after the Virtual Memory Manager carries out the request.

NT takes a unique approach, known as modified microkernel, that falls between pure microkernel and monolithic design. In NT's modified microkernel design, operating system environments execute in user mode as discrete processes, including DOS, Win16, Win32, OS/2, and POSIX (DOS and Win16 are not shown in Figure 1). The basic operating system subsystems, including the Process Manager and the Virtual Memory Manager, execute in kernel mode, and they are compiled into one file image. These kernel-mode subsystems are not separate processes, and they can communicate with one another by using function calls for maximum performance.

NT's user-mode operating system environments implement separate operating system APIs. The degree of NT support for each environment varies, however. Support for DOS is limited to the DOS programs that do not attempt to access the computer's hardware directly. OS/2 and POSIX support stops short of user-interface functions and the advanced features of the APIs. Win32 is really the official language of NT, and it's the only API Microsoft has expanded since NT was first released.

NT's operating system environments rely on services that the kernel mode exports to carry out tasks that they can't carry out in user mode. The services invoked in kernel mode are known as NT's native API. This API is made up of about 250 functions that NT's operating systems access through software-exception system calls. A software-exception system call is a hardware-assisted way to change execution modes from user mode to kernel mode; it gives NT control over the data that passes between the two modes.

Native API requests are executed by functions in kernel mode, known as system services. To carry out work, system services call on functions in one or more components of NT's Executive. As shown in Figure 1, the Executive components include the I/O Manager, Object Manager, Security Reference Monitor, Process Manager, Local Procedure Call Facility, and Virtual Memory Manager. Each Executive component has a specific operating system responsibility. Device drivers are dynamically added NT components that work closely with the I/O Manager to connect NT to specific hardware devices, such as disks and input devices.

A handful of other components are not usually described in Microsoft's NT architecture literature (e.g., the Cache Manager and the Configuration Manager). I'll describe these components in Part 2 of this primer.

NT's Executive components use basic hardware functionality implemented in the microkernel. The microkernel, which is known in NT as the Kernel, contains the scheduler. The Kernel also manages the Executive's use of NT's hardware and software interrupt handlers and exports synchronization primitives.

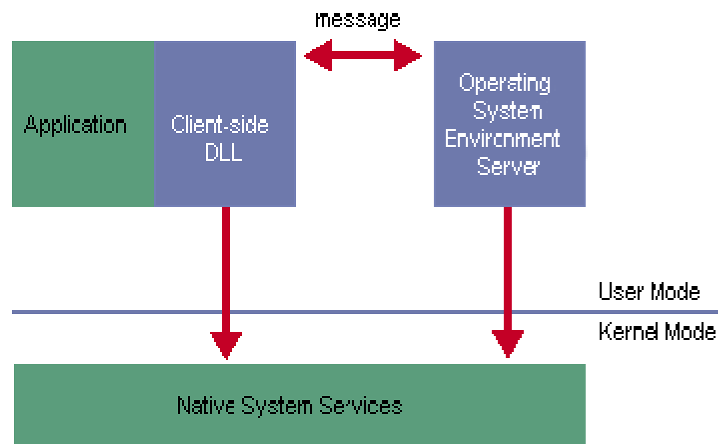
Device drivers and the Kernel use the HAL to interact with the computer's hardware. The HAL exports its own API, which translates abstract data into processor-specific commands. NT is portable across processor types because processor-specific code is restricted to the Kernel and the HAL. This situation means that when NT is ported to a new processor, only the Kernel and the HAL must be converted. The rest of NT's code is written in C and C++ and can simply be recompiled for the new processor.

Those are the basics of NT's architecture. Now let's delve into NT's operating system environments and system services more deeply.

Operating System Environments

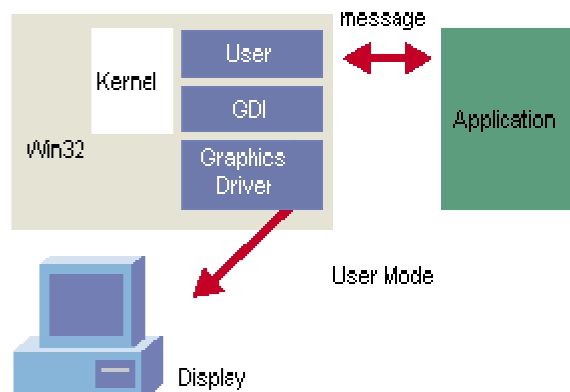
NT's operating system environments are implemented as client/server systems. As part of the compile process, applications are bound by a link-time binding to an operating system API that NT's operating system environments export. The link-time binding connects the application to the environment's client-side DLLs, which accomplish the exporting of the API. For example, a Win32 program is a client of the Win32 operating system environment server, so it is linked to Win32's client-side DLLs, including Kernel32.dll, gdi32.dll, and user32.dll. A POSIX program would be linked to the POSIX client-side DLL, psxdll.dll.

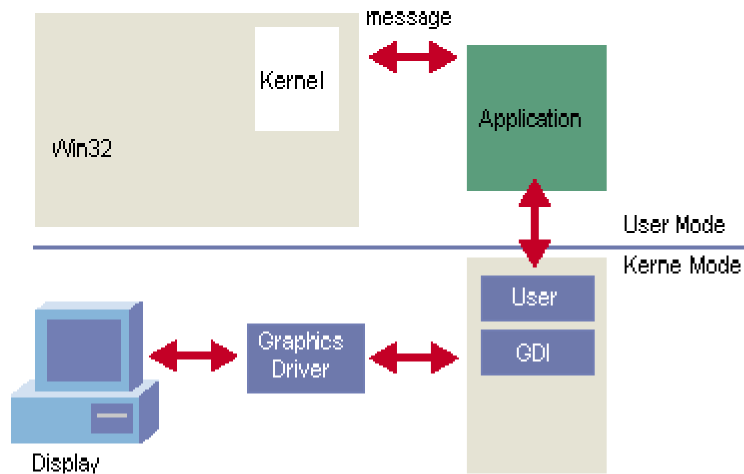
Client-side DLLs carry out tasks on behalf of their servers, but they execute as part of a client process. As Figure 4, page 66, shows, in some cases a client-side DLL can fully implement an API without having to call upon the help of the server; in other cases, the server must help out. The server's aid is usually necessary only when global information related to the environment must be updated. When the client-side DLL requires help from the server, the DLL sends a message known as a local procedure call (LPC) to the server. When the server completes the specified request and returns an answer, the DLL can complete the function and return control to the client. Both the client-side DLL and the server may use NT's native API when necessary. Operating system environment APIs augment the native API with additional functionality or semantics that are specific to themselves.



One example of an operating system environment API that the environment's server must service is a CreateProcess function, in which the server creates a relationship between the client process and a new process. To create such a relationship, the server's CreateProcess function must call NT's native API CreateProcess function. An example of an operating system environment API that does not require client-side DLL interaction with its server is the ReadFile function. The ReadFile function can be implemented entirely in the DLL with the aid of the native API's ReadFile function. Because the ReadFile function does not require the update of global information, the server 's help is not necessary.

Because some operating system environment APIs require messages between a client and its server, an assumption has developed that system calls in NT are expensive. However, NT's LPC facility is highly optimized and very efficient. Nevertheless, Microsoft removed the most LPC-intensive portion of NT 3.51's Win32 operating system environment. Figure 5 shows NT 3.51 Win32 architecture, and Figure 6 shows the change to this architecture in NT 4.0.





The Win32 environment includes graphics and user-interface functions, which are implemented in its graphics device interface (GDI) and User components. In NT 3.51, whenever a Win32 program makes a drawing or user-interface call, the GDI or User client-side DLLs make LPC calls to the Win32 server (CSRSS.EXE). Those LPC calls to the server cause Win32's sluggish performance--the bane of microkernel-based operating systems. In NT 4.0, the User and GDI components move from user mode into kernel mode as a new Executive subsystem, Win32K.SYS. When a drawing call is made, the client-side GDI's DLL makes a new native system call into kernel mode, where the request is carried out (Win32 native system calls didn't exist in NT 3.5x). There is no message passing and no context switches--just a switch from user mode to kernel mode and back. This optimization has a dramatic effect on the performance of Win32 applications.

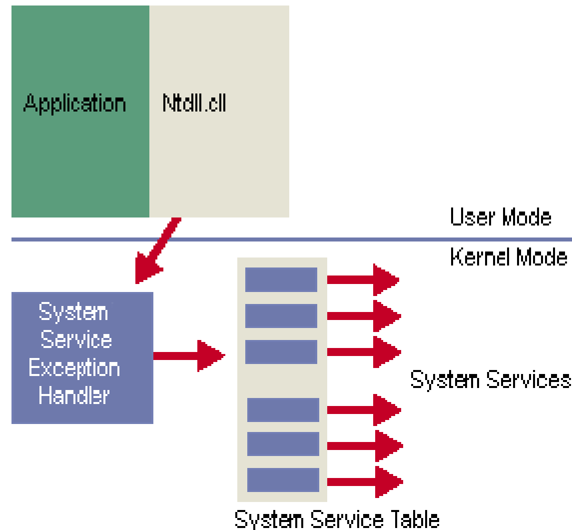
System Services

System services export the native API from kernel mode so that user-mode portions of NT can use it. The native API is intended for use by operating system environments, but nothing prevents an application from bypassing the operating system environment API and accessing the native API directly. However, the native API is usually undocumented, is very similar to (but more cumbersome than) the Win32 API, and would not give an application privileges or powers its operating system environment would not give it.

System services have names that begin with Nt. For example, Win32 has an API function called CreateProcess, which the Win32 server handles. CreateProcess calls the native API function, NtCreateProcess. The parameter lists for both functions are similar; however, CreateProcess performs significant amounts of work on behalf of the environment. For instance, it sets up the process's environment variables and command line and fills in the process address map with the program to be executed. System services validate parameters that are passed from user mode and then usually call functions within Executive subsystems. For example, NtCreateProcess calls the Process Manager Executive subsystem, invoking its PsCreateProcess function. Most system services are short because they serve primarily as thin interfaces between user mode and Executive subsystems. There can be a one-for-one correspondence between Win32 calls and native calls, but many Win32 functions make more than one native call to carry out a task.

Figure 7 shows the flow of control when an application calls a native API function. Applications and operating system environments that use the native API access it through a DLL named ntdll.dll. This DLL is linked to every process in an NT system and consists of entry points for every system service. These entry points don't do much other than preparing variables and causing a system service

software exception. The System Service Exception Handler in kernel mode is executed in response to system service exceptions, and it uses a number associated with the requested service to index the System Service Table and find the function that implements the service. Thus, adding a new system service requires updates of that table and ntdll.dll. Microsoft continues to add to the number of system services in NT, and almost two dozen new calls will appear in NT 5.0.



Stay Tuned

Next month I'll conclude our two-part primer on NT's architecture with a look at the Executive and a description of the responsibilities and capabilities of each of its subsystems. I'll also take you inside the Kernel and down into the HAL.

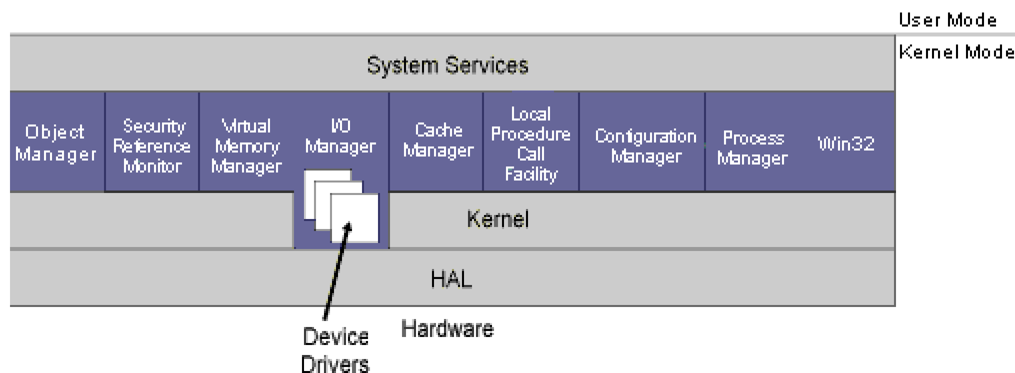
Last month I began a two-part primer on Windows NT architecture. This month I conclude with a description and discussion of the components that make up the NT Executive. I'll discuss the responsibilities of the Kernel and delve into one of the more mysterious elements of NT, the hardware abstraction layer (HAL).

The Executive

NT's Executive subsystems make up the meatiest layer in kernel mode, and they perform most of the functions traditionally associated with operating systems. Table 1 lists NT's Executive subsystems, and Figure 1, page 60, shows their position in NT's architecture. These subsystems have separate responsibilities and names, so you might think they are different processes. For example, when a program like Microsoft Word requests an operating-system service such as memory allocation, the flow of control proceeds from Word into kernel mode through NT's native system service interface. A system service handler for memory allocation then directly invokes the Virtual Memory Manager's allocation function. The requested allocation executes in the context of the process (Word) that requested it--there is no context switch to a different system process.

TABLE 1: NT's Executive Subsystems and Their Functions	
Executive Subsystem	Function
Object Manager	Manages resources and implements a global namespace
Security Reference Monitor	Implements NT's security model of security IDs (SIDs) and Discretionary

	Access Control Lists (DACLs)
Virtual Memory Manager	Defines process address spaces and assigns physical memory
I/O Manager	Serves as the interface between applications and device drivers
Cache Manager	Implements a file-based global file cache
Local Procedure Call (LPC) Facility	Provides efficient interprocess communication
Configuration Manager	Manages the Registry
Process Manager	Exports process and thread APIs
Win32	Implements Win32 messaging and drawing functions (new to NT 4.0)
Plug-and-Play Manager	Notifies device drivers of devices coming online and going offline (new to NT 5.0)
Power Manager	Controls the machine's power state (new to NT 5.0)



If you've seen the system process in NT's Performance Monitor (Perfmon), you might think that the Executive subsystems are different processes. However, the purpose of the system process in Perfmon is to own Executive threads (commonly called worker threads) that carry out work, usually of a background nature, for Executive subsystems. For example, the Cache Manager creates system process threads for lazy-write operations: Every few seconds the threads will flush dirty disk data from memory back to the disk. Because no user-mode application is associated with a system process, the user-mode portion of the system process' address map is not defined. And because the address map's user-mode portion does not change when a thread from the system process executes, the computer's address-mapping structures are not updated. This situation is different from a change from one application to another, in which case the user-mode portion of the address map would have to be changed from, say, Word's to Netscape's.

Just as NT doesn't assign Executive subsystems to different processes, NT doesn't place the Executive subsystems in different image files (an image file is an executable file). The ntoskrnl.exe file contains all NT Executive subsystems (except the Win32 subsystem, which is in win32k.sys) and the Kernel. NT loads the ntoskrnl.exe file during the system boot into the kernel-mode half of the virtual memory map.

Object Manager. The Object Manager, which I characterized in a previous column as probably the least known of NT's Executive subsystems (see "Inside NT's Object Manager," October 1997), is also

one of the most important. An operating system's primary role is to manage a computer's physical and logical resources. Other Executive subsystems use the Object Manager to define and manage objects that represent resources. For example, through the Object Manager, the Process Manager defines a process object to track active processes. Table 2 lists the NT 4.0-defined objects and the kernel-mode subsystems that manage them.

TABLE 2: NT 4.0 Object Types and the Kernel-Mode Subsystems That Manage Them

Object Type	Represents	Owning Subsystem
Object type	Object type object	Object Manager
Directory	Object namespace	Object Manager
SymbolicLink	Object namespace	Object Manager
Event	Synchronization primitive	Executive
EventPair	Synchronization primitive	Executive
Mutant	Synchronization primitive	Executive
Timer	Timer notifications	Executive
Semaphore	Synchronization primitive	Executive
Windows Station	Interactive logon	Win32
Desktop	Windows desktop	Win32
File	Tracks open files	I/O Manager
I/O Completion	Tracks I/O completion notifications	I/O Manager
Adapter	Direct memory access (DMA) resource	I/O Manager
Controller	DMA controller	I/O Manager
Device	Logical or physical device	I/O Manager
Driver	Device driver	I/O Manager
Key	Doorway to the Registry	Configuration Manager
Port	Communications channel	Local Procedure Call (LPC) Facility
Section	Memory mapping	Memory Manager
Process	Active process	Process Manager
Thread	Active thread	Process Manager
Token	Process security profile	Process Manager
Profile	Performance monitoring	Kernel

The Object Manager performs object-management duties that include identification and reference counting. When an application opens a resource, the Object Manager either locates the associated object or creates a new object. Instead of returning an object pointer to the application that opened the resource, the Object Manager returns an opaque identifier called a handle. The handle's value is unique to the application that opened the resource, but it is not unique to the system across different applications. The application uses the handle to identify the resource in subsequent operations. When

the application is finished with the object, the application closes the handle. The Object Manager uses reference counting to track how many system parts, including applications and Executive subsystems, are accessing an object that represents a resource. When the reference count goes to zero, the object is no longer in use representing the resource, and the Object Manager deletes the object (but not necessarily the resource).

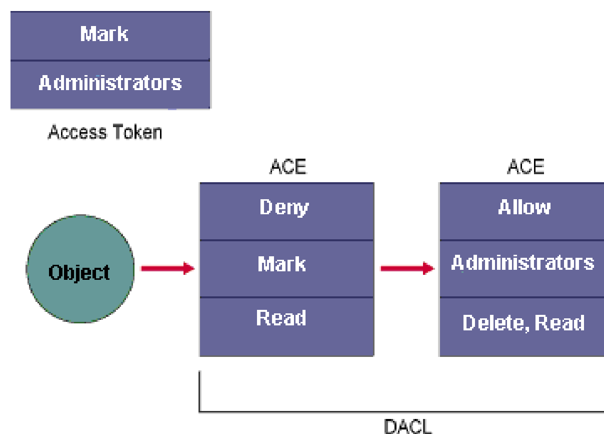
The Object Manager implements NT's namespace to provide object identification. All shareable resources in NT have names that are rooted in this namespace. For example, when a program opens a file, the Object Manager parses the file's name to locate the file-system driver for the disk that stores the file. Similarly, when an application opens a Registry key, the Object Manager determines from the Registry key's name that the Configuration Manager must be called.

Most native system services that NT implements are resource related; thus, almost every system service invokes Object Manager functions. For example, services that open an existing resource call on the Object Manager to look up the resource name in the Object Manager namespace, ensure the caller has sufficient rights to open the resource, and allocate and return a handle to identify the open instance. Services that require a handle to a previously opened resource call the Object Manager to translate the handle to the object it represents.

The Object Manager calls other Executive subsystems when necessary. Every object type has functions that execute when NT performs particular operations on objects of that type. Thus, when the Object Manager creates a file object to represent an open file, the Object Manager invokes the I/O Manager's function for opening files. Similarly, the Object Manager creates an associated process object for an open process and invokes the Process Manager's function for opening processes.

Security Reference Monitor. The Security Reference Monitor is closely associated with the Object Manager. The Object Manager calls the Security Reference Monitor for an access check before letting an application open an object. The Object Manager also calls the Security Reference Monitor before it lets applications perform other operations on objects, such as reading from the object or writing to it.

The Security Reference Monitor implements a security model based on security identifiers (SIDs) and Discretionary Access Control Lists (DACLs). Every process in NT has an associated access token object that contains the SID identifying the user that owns the process and the SIDs of the groups the user belongs to. When a security check takes place, the SIDs in the access token of the process describe the user trying to complete an action on an object. Figure 2 gives an example of a DACL that does not let the process owner, Mark, read the object, although it lets the group the owner belongs to (Administrators) read from and delete the object.



NT's security model has a powerful capability that lets a process impersonate any user other than the user associated with the process. Server applications such as NT's built-in file server (SRV) rely heavily on impersonation. When a client on a different machine opens a file on the server, the server impersonates the client by temporarily adopting an access token that identifies the server as the remote client. NT creates the token on the server, but the token contains the client's SIDs. When the server opens the file, it invokes the Object Manager, which then calls the Security Reference Monitor to make the appropriate access check. The client can have more or less privilege than the server (the server might not be allowed to open the file), but impersonation lets the server temporarily identify as the client and thus hides the discrepancy.

DACLs specify the actions that particular SIDs can perform on an object. A DACL can contain any number of access control entries (ACEs), including no entries, that contain the information about actions SIDs can perform. Each ACE contains a SID, a flag specifying whether the ACE is of the deny type or allow type, and an operations mask (i.e., read, write, delete). Every object can have an ACE connected to it, such as the example in Figure 2 shows. NT references the ACE when a user attempts to open the object.

Given the access token object and the DACL in Figure 2, the Security Reference Monitor would deny Mark read access to the object, even though it allows members of the Administrator group read access to the object. The Security Reference Monitor would deny Mark read access to the object because the deny ACE is in front of the allow ACE in the DACL.

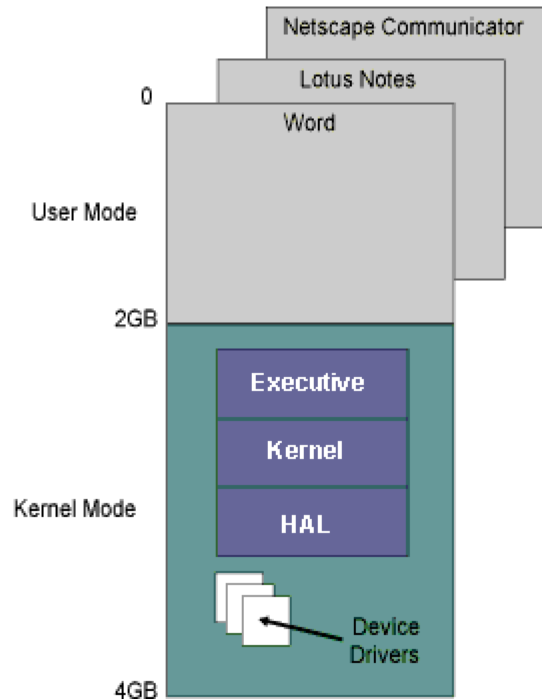
When a process wants to open an object, it must indicate the access it desires (e.g., read, write, delete). The Object Manager calls the Security Reference Monitor for an access check, and the Security Reference Monitor takes the desired access and the SIDs from the process' access token and goes through the object's DACL until it finds matching information. The Security Reference Monitor then looks at the DACL's ACE type: If the ACE is an allow type, the process can open the object. If the ACE is a deny type, the process cannot access the object. Two special cases exist in DACL security. First, users can fully access an object that does not have a DACL. Second, users cannot access an object with an existing but empty DACL.

When an object opens successfully, NT associates the access types granted to the calling process (and that match the access types specified during the open function) with the handle that NT returns to the calling process. When the calling process later performs an operation on the object, all the Security Reference Monitor must do is verify that the granted access types permit the operation--there is no need for the Security Reference Monitor to rescan the DACL.

The Security Reference Monitor also implements System Access Control Lists (SACLs), which are similar to DACLS. SACLs tell the system to log specific actions when particular users perform those actions. Systems administrators typically use SACLs to monitor and record attempted security violations.

Virtual Memory Manager. The Virtual Memory Manager has two main duties: to create and manage address maps for processes and to control physical memory allocation. NT 4.0 implements a 32-bit (4GB) address space; however, applications can directly access only the first 2GB, as Figure 3, page 62, shows. This portion of the address space is the user-mode half of the address map, and it changes to reflect the currently executing program's address map (e.g., Netscape, Notepad, Word). The 2GB to 4GB portion of the address space is for the kernel-mode portions of NT, and it doesn't change. NT 4.0 Service Pack 3 (SP3) and NT Server, Enterprise Edition 4.0, let administrators move

the boundary in the address space so that user-mode applications can access 3GB of the map and kernel-mode components can use only 1GB.



The Virtual Memory Manager implements demand-paged virtual memory, which means it manages memory in individual segments, or pages. In x86 systems, a page is 4096 bytes; in Alpha systems, a page is 8192 bytes. The total memory applications require can exceed the computer's physical memory space. The Virtual Memory Manager stores the data that exceeds a computer's physical memory on the hard disk in page files. The Virtual Memory Manager transfers data to physical memory from a paging file when an application requests the data.

The Virtual Memory Manager has advanced capabilities that implement file memory mapping, memory sharing, and copy-on-write page protection. NT uses file memory mapping to load executable images and DLLs efficiently. In memory mapping, the Virtual Memory Manager learns through the operating system that a portion of a process' address map is connected to a particular file. When the process touches these portions of its address map (e.g., when it tries to execute code), the Virtual Memory Manager automatically loads the data into physical memory.

NT uses memory sharing to enhance physical memory use and to communicate between processes. For example, multiple instances of a program share a memory-mapped file image, and this sharing increases memory efficiency.

Copy-on-write is an optimization related to memory sharing in which several programs share common data that each program can modify individually. When one program writes to a copy-on-write page that it shares with another program, the program that makes the modification gets its own version of the copy-on-write page to scribble on. The other program then becomes the original page's sole owner. NT uses copy-on-write optimization when several applications share the writable portions of system DLLs.

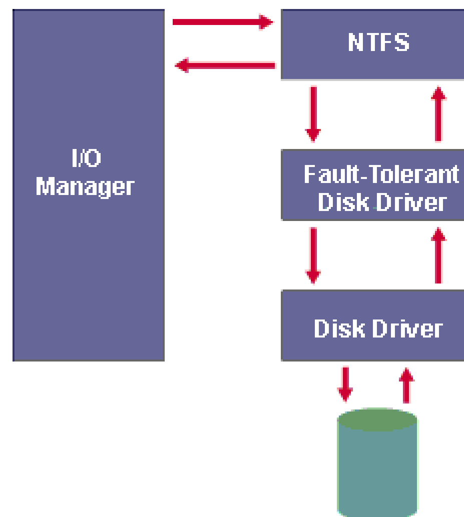
The Virtual Memory Manager divides physical memory among several executing programs. It uses a function called working-set tuning to allocate additional memory to programs that require it and ensure that other executing programs have enough memory to keep running.

I/O Manager. The I/O Manager is responsible for integrating add-on device drivers with NT. Microsoft did not build support for various hardware devices into NT. Device drivers, which are dynamically loaded kernel-mode components, provide hardware support. A device driver controls a specific type of hardware device by translating the commands that NT and applications direct to the device and manipulating the hardware to carry out the commands. Microsoft supplies several device drivers for common hardware. If you purchase a nonstandard hardware item, the hardware vendor will provide a device driver for it.

The I/O Manager supports asynchronous, packet-based I/O. For example, when a program such as Lotus Notes reads from a file, the read-file system service, NtReadFile, allocates an I/O request packet (IRP) that describes everything a device driver needs to know to complete the program's request. The IRP information includes the location of the buffer into which the program must read the requested data, a pointer to the file object that represents the open file, the offset into the file where the data resides, and the amount of data the program must read. The I/O Manager takes the IRP and passes it to the device driver--in this example, a file system driver responsible for the target file. A file object represents an open file in this example, but a file object can represent a keyboard, a mouse, or an open network connection.

NT's developers designed IRPs to contain everything pertinent to an application's request to make implementing asynchronous packet-based I/O easier. For example, after the I/O Manager gives an application's request (and an IRP) to a device driver, the I/O Manager returns control to the application. The application can continue performing useful work while the device driver is transferring data and can check the device driver after the data transfer. Because asynchrony (or the overlapping of control between the application and the device driver) is sometimes difficult to program to, standard Win32 APIs hide the asynchrony. For example, when an application calls the Win32 function to read from a file, the application does not resume execution until after the function reads the data. Most advanced applications use Win32 APIs that expose the I/O Manager's asynchrony, because doing so can improve performance.

The I/O Manager supports 64-bit file offsets and layered device drivers. Using 64-bit offsets lets NT's file systems address extremely large files and lets disk device drivers address extremely large disks. Layering lets device drivers divide their labor. As the example in Figure 4 shows, the NTFS driver is layered above the fault-tolerant disk driver, which sits above a standard disk driver. As the I/O Manager processes requests, IRPs move down the layers, and the results pass back up from the bottom as each driver finishes its work.



Cache Manager. The Cache Manager works closely with the Virtual Memory Manager and file system drivers. The Cache Manager maintains NT's global (shared by all file systems) file system cache. The working-set tuner assigns physical memory to the file system cache. The NT cache is file oriented rather than disk-block oriented, as Windows 95 is. When the working-set tuner takes memory containing modified file data away from the Cache Manager, the I/O Manager invokes file systems that manage the moved files to write their data back to the disk.

Local Procedure Call Facility. NT's Local Procedure Call (LPC) Facility optimizes communications for applications, including operating system environments. The LPC function is based on two types of port object: connection ports and communication ports. A server creates a connection port, which a client connects to. After the client establishes that connection, the server creates a communication port, which the server and client transmit data through.

Three kinds of LPC exist: data copying, shared memory, and shared memory with event pairs (Quick-LPC). NT uses data copying for small messages (less than 256 bytes). One end of the communications link (client or server) copies a message to a port, and the other end of the link copies the message out of the port.

NT uses shared memory for messages larger than 256 bytes. In shared-memory LPC, the connected client and server share a region of memory. When one end of the communications link wants to send a message larger than 256 bytes to the other, it sends through the communications port a short message whose only function is to point to the location of the primary message in the shared memory. Shared memory avoids a copy operation but requires dedicated shared memory.

Win32 under NT 3.51 uses Quick-LPC. Win32 doesn't send messages through ports. Instead, one end of the communications link uses an event-pair synchronization object to signal the other end of the communications link that it has placed a message in shared memory. Using event-pair synchronization objects avoids the overhead of communicating through a port but as a trade-off has even higher resource overhead than other LPC methods.

Configuration Manager. Microsoft rarely discusses the Configuration Manager in its NT architecture documentation, but Configuration Manager is an important Executive subsystem. The Configuration Manager manages the Registry, and Win32 Registry API functions rely on NT native APIs the Configuration Manager implements. The Configuration Manager also exports functions to the I/O

Manager, and the I/O Manager uses these functions to assign physical resources to device drivers. The Configuration Manager stores this assignment information in the Registry to detect and help prevent resource conflicts.

Process Manager. The Process Manager works with the Kernel to define process and thread objects. The Process Manager wraps the Kernel's process object and adds to it a process identifier (PID), the access token, an address map, and a handle table. The Process Manager performs a similar operation on the Kernel's thread object, adding to it a thread identifier (TID) and statistics. These statistics include process and thread start and exit times and various virtual-memory counters.

The Process Manager exports an interface that lets other Executive subsystems and user-mode applications manipulate processes and threads. For example, applications can access Process Manager functions to create processes, delete them, and modify their characteristics (such as their priority). You can access many Process Manager functions in user mode through system services.

Win32. Win32 consists of the messaging and drawing functions of the Win32 API. As I discussed in Part 1, in NT 3.51 these modules resided in user mode as part of the Win32 environment subsystem.
Plug-and-Play Manager and Power Manager. Two new Executive subsystems will debut in NT 5.0: the Plug-and-Play Manager and the Power Manager. The Plug-and-Play Manager notifies device drivers and the I/O Manager when hardware devices come online or are removed. The Power Manager maintains central control of the computer's power level, letting it shift into low power modes when possible. Both new subsystems work primarily with the I/O Manager and device drivers.

The Kernel

NT's Kernel operates more closely with hardware than the Executive does, and it contains CPU-specific code. NT's thread scheduler, called the dispatcher by NT's developers, resides in the Kernel. The dispatcher implements 32 priority levels, 0-31. The dispatcher reserves priority level 0 for a system thread that zeros memory pages as a background task. Priority levels 1 through 15 are variable (with some fixed priority levels) and are where programs execute; priority levels 16 through 31 are fixed priority levels that only administrators can access.

The NT dispatcher is a preemptive scheduler. The CPU's time is divided into slices called quantums. When a thread runs to the end of its quantum and doesn't yield the CPU, the dispatcher will step in and preempt it or schedule another thread of equal priority that is waiting to run (see "Inside the Windows NT Scheduler, Part 1," July 1997).

NT implements most synchronization primitives in the Kernel. NT has a rich set of synchronization types, including mutexes, semaphores, events, and spin locks. The Kernel implements and manages its own object types, and Kernel objects represent NT's synchronization primitives. In most cases, NT wraps Kernel objects with Executive objects so that applications can access them from user mode through the native API. As I described previously, the Process Manager wraps its process and thread objects around the Kernel's objects. NT stores all priority-related information and statistical information related to scheduling (e.g., context switches, user time) in the Kernel's objects.

The Kernel manages interrupt vectors (for more information on interrupt vectors, see my column "Inside NT's Interrupt Handling," November 1997). NT defines and implements IRQ levels in the Kernel.

The Hardware Abstraction Layer

The HAL is NT's interface to the raw CPU. Microsoft wanted to make NT portable across different processors. To make this portability feasible, NT's developers isolated as much CPU-specific code as possible into a separate, dynamically replaceable module, the HAL. The HAL exports a common processor model that masks the differences in various processor chips from NT. Device drivers use this common processor rather than a particular CPU type. Even different motherboards in the same processor family can differ significantly, but hardware vendors can ensure that NT will work with their boards by writing a custom HAL to work with NT.

A common difference between motherboards in the same processor family is that some are for multiprocessor systems and others are for uniprocessor systems. A multiprocessor HAL is different from a uniprocessor HAL. The multiprocessor-uniprocessor issue brings me to a little-known fact. Microsoft provides three versions of each NT release: the uniprocessor version, the multiprocessor version, and the debug version (or the checked-build version). In addition to having different HALs, uniprocessor and multiprocessor versions of NT have different ntoskrnl.exe images. The uniprocessor version of ntoskrnl.exe does not include code that is necessary for correct execution on multiprocessors. Microsoft provides the debug version of NT for device-driver developers. The debug version contains additional sanity checks and symbolic information not contained in the uniprocessor and multiprocessor versions of NT. Microsoft offers only one debug version of NT, which contains multiprocessor code and will work on multiprocessor and uniprocessor machines.

In a Nutshell

There you have it: NT's big picture in two articles. For more in-depth information, refer to previous columns in which I focused on subsystems or portions of subsystems (you can find these columns easily by searching the magazine archives on Windows NT Magazine's Web site, at http://www.winntmag.com). Another good source of information on NT subsystems is Helen Custer's Inside Windows NT (Microsoft Press, 1993).