Part I

Essentials

Chapter 1

Windows NT: An Inside Look

IN THIS CHAPTER

+ Evaluating Windows NT
+ Delving into the Windows NT architecture

THIS BOOK IS AN EXPLORATION of the internals of the Windows NT operating system. Before entering the jungle of Windows NT internals, an overview of the topic is necessary. In this chapter, we explain the overall structure of the Windows NT operating system.

Evaluating Windows NT

The qualities of an operating system are the result of the way in which the operating system is designed and implemented. For an operating system to be portable, extensible, and compatible with previous releases, the basic architecture has to be well designed. In the following sections, we evaluate Windows NT in light of these issues.

Portability

As you know, Windows NT is available on several platforms, namely, Intel, MIPS, Power PC, and DEC Alpha. Many factors contribute to Windows NT's portability. Probably the most important factor of all is the language used for implementation. Windows NT is mostly coded in C, with some parts coded in C++. Assembly language, which is platform specific, is used only where necessary. The Windows NT team also isolated the hardware-dependent sections of the operating system in HAL.DLL. As a result, the hardware-independent portions of Windows NT can be coded in a high-level language, such as C, and easily ported across platforms.

Extensibility

Windows NT is highly extensible, but because of a lack of documentation, its extensibility features are rarely explored. The list of undocumented features starts with the subsystems. The subsystems provide multiple operating system interfaces in one operating system. You can extend Windows NT to have a new operating system interface simply by adding a new subsystem program. Windows NT provides Win32, OS/2, POSIX, Winl6, and DOS interfaces using the subsystems concept, but Microsoft keeps mum when it comes to documenting the procedure to add a new subsystem.

The Windows NT kernel is highly extensible because of dynamically loadable kernel modules that are loaded as device drivers. In Windows NT, Microsoft provides enough documentation for you to write hardware device drivers-that is, hard disk device drivers, network card device drivers, tape drive device drivers, and so on. In Windows NT, you can write device drivers that do not control any hardware device. Even file systems are loaded as device drivers under Windows NT.

Another example of Windows NT's extensibility is its implementation of the system call interface. Developers commonly modify operating system behavior by hooking or adding system calls. The Windows NT development team designed the system call interface to facilitate easy hooking and adding of system calls, but again Microsoft has not documented these mechanisms.

Compatibility

Downward compatibility has been a long-standing characteristic of Intel's microprocessors and Microsoft's operating systems, and a key to the success of these two giants. Windows NT had to allow programs for DOS, Win 16, and OS/2 to run unaltered. Compatibility is another reason the NT development team went for the subsystem concept. Apart from binary compatibility, where the executable has to be allowed to run unaltered, Windows NT also provides source compatibility for POSIX-compliant applications. In another attempt to increase compatibility, Windows NT supports other file systems, such as the file allocation table (FAT) file system from DOS and the High Performance File System (HPFS) from OS/2, in addition to the native NT file system (NTFS).

Maintainability

Windows NT is a big piece of code, and maintaining it is a big job. The NT development team has achieved maintainability through an object-oriented design. Also, the breakup of the operating system functionality into various layers improves maintainability. The topmost layer, which is the one that is seen by the users of the operating system, is the subsystems layer. The subsystems use the system call interface to provide the application programming interface (API) to the outside world. Below the system call interface layer lies the NT executive, which in turn rests on

the kernel, which ultimately relies on the hardware abstraction layer (HAL) that talks directly with the hardware.

The NT development team's choice of programming language also contributes to Windows NT's maintainability. As we stated previously, the entire operating system has been coded in C and C++, except for a few portions where the use of assembly language was inevitable.

Plus Points over Windows 95/98

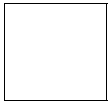
Microsoft has come up with two 32-bit operating systems: Windows 95/98 and Windows NT. Windows NT is a high-end operating system that offers additional features separate from those provided by conventional PC or desktop operating systems, such as process management, memory management, and storage management.

Security

Windows NT is a secure operating system based on the following characteristic: A user needs to log in to the system before he or she can access it. The resources in the system are treated as objects, and every object has a security descriptor associated with it. A security descriptor has access control lists attached to it that dictate which users can access the object.

All this being said, a secure operating system cannot be complete without a secure file system, and the FAT file system from the days of DOS does not have any provision for security. DOS, being a single-user operating system, did not care about security.

In response to this shortcoming, the Windows NT team came up with a new file system based on the HPFS, which is the native file system for OS/2. This new native file system for Windows NT, known as NTFS, has support for access control. A user can specify the access rights for a file or directory being created under NTFS, and NTFS allows only the processes with proper access rights to access that file or directory.



Keep in mind that no system is 100 percent secure. Windows NT, although remarkably secure, is not DoD compliant. (For the latest news on DoD compliance, check out **http://www.fcw.com/pubs/fcw/1998/0727/fcw-newsdodsec-7-27-98.htm.**)

Multiprocessing

Windows NT supports symmetric multiprocessing, the workstation version of Windows NT can support two processors, and the server version of Windows NT can support up to four processors. The operating system needs special synchronization

constructs for supporting multiprocessing. On a single-processor system, critical portions of code can be executed without interruption by disabling all the hardware interrupts. This is required to maintain the integrity of the kernel data structures. In a multiprocessor environment, it is not possible to disable the interrupts on all processors. Windows NT uses spin locks to protect kernel data structures in a multiprocessor environment.

Multiprocessing can be classified as *asymmetric and symmetric.* In asymmetric multiprocessing, a single processor acts as the master processor and the other processors act as slaves. Only the master processor runs the kernel code, while the slaves can run only the user threads. Whenever a thread running on a slave processor invokes a system service, the master processor takes over the thread and executes the requested kernel service. The scheduler, being a kernel code, runs only on the master processor. Thus, the master processor acts as the scheduler, dispatching user mode threads to the slave processors. Naturally, the master processor is heavily loaded and the system is not scalable. Compare this with symmetric multiprocessing, where any processor can run the kernel code as well as the user code.

International Language Support

A significant portion of PC users today use languages other than English. The key to reaching these users is to have the operating system support their languages. Windows NT achieves this by adopting the Unicode standard for character sets. The Unicode standard has 16-bit character set, while ASCII uses an 8-bit character set. The first 256 characters in Unicode match the ASCII character set. This leaves enough space for representing characters from non-Latin scripts and languages. The Win32 API allows Unicode as well as ASCII character sets, but the Windows NT kernel uses and understands only Unicode. Although the application programmer can get away without knowing Unicode, device driver developers need to be familiar with Unicode because the kernel interface functions accept only Unicode strings and the driver entry points are supplied with Unicode strings.

Multiprogramming

Windows NT 3.51 and Windows NT 4.0 lack an important feature, namely, the support for remote login or Telnet of a server operating system. Both these versions of Windows NT can operate as file servers because they support the common Internet file system (CIFS) protocol. But they cannot act as CPU servers because logging into a Windows NT machine over the network is not possible. Consequently, only one user can access a Windows NT machine at a time. Windows

2000 plans to overcome this deficiency by providing a Telnet server along with the operating system. This will enable multiple programmers to log in on the machine at the same time, making Windows 2000 a true server operating system.

Third-party Telnet servers are available for Windows NT 3.51 and Windows NT 4.0. However, Microsoft's own Telnet server comes only with Windows 2000.

Delving into the Windows NT Architecture

Windows NT borrows its core architecture from the MACH operating system, which was developed at Carnegie Mellon University. The basic approach of the MACH operating system is to reduce the kernel size to the minimum by pushing complex operating system functionality outside the kernel onto user-level server processes. This client-server architecture of the operating system serves yet another purpose: It allows multiple APIs for the same operating system. This is achieved by implementing the APIs through the server processes.

The MACH operating system kernel provides a very simple set of interface functions. A server process implementing a particular API uses these interface functions to provide a more complex set of interface functions. Windows NT borrows this idea from the MACH operating system. The server processes in Windows NT are called as the subsystems. NT's choice of the client-server architecture shows its commitment to good software management principles such as modularity and structured programming. Windows NT had the option to implement the required APIs in the kernel. Also, the NT team could have added different layers on top of the Windows NT kernel to implement different APIs. The NT team voted in favor of the subsystem approach for purposes of maintainability and extensibility.

The Subsystems

There are two types of subsystems in Windows NT: integral subsystems and environment subsystems. The *integral subsystems*, such as the security manager subsystem, perform some essential operating system task. The *environment subsystems* enable different types of APIs to be used on a Windows NT machine. Windows NT comes with subsystems to support the following APIs:

+ Win32 Subsystem. The Win32 subsystem provides the Win32 API. The applications conforming to the Win32 API are supposed to run unaltered on all the 32-bit platforms provided by Microsoft - that is, Windows NT, Windows 95, and Win32s. Unfortunately, as you will see later in this book, this is not always the case.

+ WOW Subsystem. The Windows on Windows (WOW) subsystem provides backward compatibility to 16-bit Windows applications, enabling Win16 applications to run on Windows NT. These applications can run on Windows NT unless they use some of the undocumented API functions from Windows 3.1 that are not defined in Windows NT.
+ NTVDM Subsystem. The NT Virtual DOS Machine (NTVDM) provides a text-based environment where DOS applications can run.
+ OS/2 Subsystem. The OS/2 subsystem enables OS/2 applications to run. WOW, NTVDM, and OS/2 are available only on Intel platforms because they provide binary compatibility to applications. One cannot run the executable files or binary files created for one type of processor on another type of processor because of the differences in machine code format.
+ POSIX Subsystem. The POSIX subsystem provides API compliance to the POSIX 1003.1 standard.

The applications are unaware of the fact that the API calls invoked are processed by the corresponding subsystem. This is hidden from the applications by the respective client-side DLLs for each subsystem. This DLL translates the API call into a local procedure call (LPC). LPC is similar to the remote procedure call (RFC) facility available on networked Unix machines. Using RFC, a client application can invoke a function residing in a server process running on another machine over the network. LPC is optimized for the client and the server running on the same machine.

THE WIN32 SUBSYSTEM

The Win32 subsystem is the most important subsystem. Other subsystems such as WOW and OS/2 are provided mainly for backward compatibility, while the POSIX subsystem is very restrictive in functionality. (For example, POSIX applications do not have access to any network that exists.) The Win32 subsystem is important because it controls access to the graphics device. In addition, the other subsystems are actually Win32 applications that use the Win32 API to provide their own different APIs. In essence, all the subsystems are based on the core Win32 subsystem.

The Win32 subsystem in Windows NT 3.51 contains the following components:

+ CSRSS.EXE. This is the user mode server process that serves the USER and GDI calls.

Traditionally, Windows API calls are classified as user/gdi calls and kernel calls. The majority of user/gdi functions are related to the graphical user interface (GUI) and reside in USER.DLL under Windows 3.x. The kernel functions are related to non-GUI O/S services — such as file system management and process management — and reside in KERNEL.EXE under Windows 3.x.

+ KERNEL32.DLL. The KERNEL.EXE in Windows 3.1 has changed to KERNEL32.DLL in Windows NT. This is more than a change in name. The KERNEL.EXE contained all the kernel code for Windows 3.1, while KERNEL32.DLL contains just the stub functions. These stub functions call the corresponding NTDLL.DLL functions, which in turn invoke system call code in the kernel.
+ USER32.DLL. This is another client-side DLL for the Win32 subsystem. The majority of the functions in USER32.DLL are stub functions that convert the function call to an LPC for the server process.
+ GDI32.DLL. The functions calls related to the graphical device interface are handled by another client-side DLL for the Win32 subsystem. The functions in GDI32.DLL are similar to those in USER32.DLL in that they are just stubs invoking LPCs for the server process.

Under Windows NT 4.0 and Windows 2000, the functionality of CSRSS is moved into a kernel mode driver (WIN32K.SYS) and USER32 and GDI32 use the system calls interface to call the services in WIN32K.SYS.

The Core

We have to resort to new terminology for explaining the kernel component of the Windows NT operating system. Generally, the part of an operating system that runs in privileged mode is called as the kernel. The Windows NT design team strove to achieve a structured design for the operating system. The privileged-mode component of Windows NT is also designed in a layered fashion. A layer uses only the functions provided by the layer below itself. The main layers in the Windows NT core are the HAL, the kernel, and the NT executive. Because one of the layers running in privileged mode is itself called as the kernel, we had to come up with a new term that refers to all these layers together. We'll refer to it as the core of Windows NT.

Most modern microprocessors run in at least two modes: *normal* and *privileged.* Some machine instructions can be executed only when the processor is in privileged mode. Also, some memory area can be marked as "to be accessed in privileged mode only."The operating systems use this feature of the processors to implement a secure operating environment for multitasking.The user processes run in normal (nonprivileged) mode, and the operating system kernel runs in privileged mode. Thus, the operating system ensures that user processes cannot harm the operating system.

This division of the Windows NT core into layers is logical. Physically, only the HAL comes as a separate module. The kernel, NT executive, and the system call layer are all packed in a single NTOSKRNL.EXE (or NTKRNLMP.EXE, for multiprocessor systems). Though they are considered part of the NT executive in this chapter, the device drivers (including the file system drivers) are separate driver modules and are loaded dynamically.

THE HAL

The lowest of the aforementioned layers is the hardware abstraction layer, which deals directly with the hardware of the machine. The HAL, as its name suggests, hides hardware idiosyncrasies from the layers above it. As we mentioned previously, Windows NT is a highly portable operating system that runs on DEC Alpha, MIPS, and Power-PC, in addition to Intel machines. Along with the processor, the other aspects of a machine, such as the bus architecture, interrupt handling, and DMA management also change. The HAL.DLL file contains the code that hides the processor- and machine-specific details from other parts of the core. The kernel component of the core and the device drivers use the HAL interface functions. Thus, only the HAL code changes from platform to platform; the rest of the core code that uses the HAL interface is highly portable.

THE KERNEL

The kernel of Windows NT offers very primitive but essential services such as multiprocessor synchronization, thread scheduling, interrupt dispatching, and so on. The kernel is the only core component that cannot be preempted or paged out. All the other components of the Windows NT core are preemptive. Hence, under Windows NT, one can find more than one thread running in privileged mode. Windows NT is one of the few operating systems in which the core is also multithreaded.

A very natural question to ask is "Why is the kernel nonpreemptive and nonpageable?" Actually, you can page out the kernel, but a problem arises when you page in. The kernel is responsible for handling page faults and bringing in the required pages in memory from secondary storage. Hence, the kernel itself cannot be paged out, or rather, it cannot be paged in if it is paged out. The same problem prevents the disk drivers supporting the swap space from being pageable. As the kernel and the device drivers use the HAL services, naturally, the HAL is also nonpreemptive.

THE NT EXECUTIVE

The NT executive constitutes the majority of the Windows NT core. It sits on top of the kernel and provides a complex interface to the outside world. The executive is designed in an object-oriented manner. The NT executive forms the part of the Windows NT core that is fully preemptive. Generally, the core components added by developers form a part of the NT executive or rather the I/O Manager. Hence, driver developers should always keep in mind that their code has to be fully preemptive.

The NT executive can further be subdivided into separate components that implement different operating system functionality. The various components of the executive are described in the following sections.

THE OBJECT MANAGER Windows NT is designed in an object-oriented fashion. Windows, devices, drivers, files, mutexes, processes, and threads have one thing in common: All of them are treated as objects. In simpler terms, an *object* is the data bundled with the set of methods that operate on this data. The Object Manager makes the task of handling objects much easier by implementing the common functionality required to manage any type of object. The main tasks of the Object Manager are as follows:

+ Memory allocation/deallocation for objects.

+ Object name space maintenance. The Windows NT object name space is structured as a tree, just like a file system directory structure. An object
- name is composed of the entire directory path, starting from the root directory. The Object Manager is responsible for maintaining this object
'. name space. Unrelated processes can access an object by getting a handle to it using the object's name.

+ Handle maintenance. To use an object, a process opens the object and gets back a handle. The process can use this handle to perform further operations on the object. Each process has a handle table that is maintained by the Object Manager. A handle table is nothing more than an array of pointers to objects; a handle is just an index in this array.
- When a process refers to a handle, the Object Manager gets hold of the actual object by indexing the handle in the handle table.

+ Reference count maintenance. The Object Manager maintains a reference count for objects, and automatically deletes an object when the corresponding reference count drops to zero. The user mode code accesses objects via handles, while the kernel mode code uses pointers to directly access objects. The Object Manager increments the object reference count for every handle pointing to the particular object. The reference count is decremented whenever a handle to the object is closed. Whenever the kernel mode code references an object, the reference count for that object is incremented. The reference count is decremented as soon as the kernel mode code is finished accessing the object.

+ Object security. The Object Manager also checks whether a process is allowed to perform a certain operation on an object. When a process creates an object, it specifies the security descriptor for that object. When another process tries to open the object, the Object Manager verifies whether the process is allowed to open the object in the specified mode. The Object Manager returns a handle to the object if the open request succeeds. As described earlier, a handle is simply an index in a per-process

table that has pointers to actual objects. The mode in which the open request on an object is granted is stored in the handle table along with the object pointers. Later, when the process tries to access the object using the handle, the Object Manager ensures that proper access rights are associated with the handle.

THE 1/0 MANAGER The I/O Manager controls everything related to input and output. It provides a framework that all the I/0-related modules (device drivers, file systems, Cache Manager, and network drivers) must adhere to.

+ Device Drivers. Windows NT supports a layered device driver model. The I/O Manager defines a common interface that all the device drivers need to provide. This ensures that the I/O Manager can treat all the devices in the same manner. Also, device drivers can be layered, and a device driver can expect the same interface from the driver sitting below it. A typical example of layering is the device driver stack to access a hard disk. The lowest-level driver can talk in terms of sectors, tracks, and sides. There may be a second layer that can deal with hard disk partitions and provide an interface for dealing with logical block numbers. The third layer can be a volume manager driver that can club several partitions into volumes. Finally, a file system driver that provides an interface to the outside world can sit on top of the volume manager.

+ File Systems. File systems are also coded as loadable device drivers under Windows NT. Consequently, a file system can be stacked on top of a disk device driver. Also, multiple file systems can be layered in such a manner that each layer adds to the functionality. For example, a replication file system can be layered on top of a normal disk file system. The replication file system need not implement the code for on-disk structure modifications.

+ Cache Manager. In her book *Inside Windows NT*, Helen Custer considers the Cache Manager part of the I/O Manager, though the Cache Manager does not adhere to the device driver interface. The Cache Manager is responsible for ensuring faster file read/write response. Though hard disk speeds are increasing, reading/writing to a hard disk is much slower than reading/writing to RAM. Hence, most operating systems cache the file data in RAM to satisfy the read requests without needing to read the actual disk block.

  Also, a write request can be satisfied without actually writing to the disk. The actual block write happens when system activity is low. This technique is called as *delayed write.*

  Another technique called as *read ahead* improves response time. In this

technique, the operating system guesses the disk blocks that will be read in the future, depending on the access patterns. These blocks are read even before they are requested. The Cache Manager uses the memory mapping features of the Virtual Memory Manager to implement caching.

+ Network Drivers. The network drivers have an interface standard different from regular device drivers. The network card drivers stick to the network driver interface specification (NDIS) standard. The drivers providing transport level interface are layered above the network card drivers and provide transport driver interface (TDI).

THE SECURITY REFERENCE MONITOR The Security Reference Monitor is responsible for validating a process's access permissions against the security descriptor of an object. The Object Manager uses the services of the Security Reference Monitor while validating a process's request to access any object.

THE VIRTUAL MEMORY MANAGER An operating system performs two essential tasks:

1. It provides a virtual machine, which is easy to program, on top of raw hardware, which is cumbersome to program. For example, an operating system provides services to access and manipulate files. Maintaining data in files is much easier than maintaining data on a raw hard disk.
2. It allows the applications to share the hardware in a transparent way. For example, an operating system provides applications with a virtual view of the CPU, where the CPU is exclusively allotted to the application. In reality, the CPU is shared by various applications, and the operating system acts as an arbitrator.

These two tasks are performed by the Virtual Memory Manager component of the operating system when it comes to the hardware memory. Modern microprocessors need an intricate data structure setup (for example, the segment table setup or the page table setup) for accessing the memory. The Virtual Memory Manager performs this task for you, which makes life easier. Furthermore, the Virtual Memory Manager enables the applications to share the physical memory transparently. It presents each application with a virtual address space where the entire address space is owned by the application.

The virtual memory concept is one of the key concepts in modern operating systems. The idea behind it is as follows. In case the operating system loads the entire program in memory while executing it, the size of the program is severely constrained by the size of physical memory. A very straightforward solution to the problem is not to load the entire program in memory at one time, but to load portions of it as and when required. A fact that supports this solution is the locality of reference phenomenon.

A process accesses only a small number of adjacent memory locations, if one considers a small time frame. This is even more pronounced because of the presence of looping constructs. In other words, the access is localized to a small number of memory pages, which is the reason it is called as locality of reference.

The operating system needs to keep only the *working set* of a process in memory. The rest of the address space of the process is supported by the swap space on the secondary storage. The Virtual Memory Manager is responsible for bringing in the pages from the secondary storage to the main memory in case the process accesses a paged-out memory location. The Virtual Memory Manager is also responsible for providing a separate address space for every process so that no process can hamper the behavior of any other process. The Virtual Memory Manager is also responsible for providing shared memory support and memory-mapped files. The Cache Manager uses the memory-mapping interface of the Virtual Memory Manager.

A *working set* is the set of memory pages that needs to be in memory for a process to execute without incurring too many page faults. A *page fault* is the hardware exception received by the operating system when an attempt is made to access a paged-out memory location.

THE PROCESS MANAGER The Process Manager is responsible for creating processes and threads. Windows NT makes a very clear distinction between processes and threads. A *process* is composed of the memory space along with various objects (such as files, mutexes, and others) opened by the process and the threads running in the process. A *thread* is simply an execution context - that is, the CPU state (especially the register contents). A process has one or more threads running in it.

THE LOCAL PROCEDURE CALL FACILITY The local procedure call (LPC) facility is specially designed for the subsystem communication. LPC is based on remote procedure call (RFC), which is the de facto Unix standard for communication between processes running on two different machines. LPC has been optimized for communication between processes running on the same machine. As discussed earlier, the LPC facility is used as the communication mechanism between the subsystems and their client processes. A client thread invokes LPC when it needs some service from the subsystem. The LPC mechanism passes on the parameters for the service invocation to the server thread. The server thread executes the service and passes the results back to the client thread using the LPC facility.

WIN32K.SYS: A Core Architecture Modification

In Windows NT 3.51, the KERNEL32.DLL calls are translated to system calls via NTDLLDLL, while the GDI and user calls are passed on to the Win32 subsystem process. Windows NT 4.0 has maintained more or less the same architecture as Version 3.51. However, there is a major modification in the core architecture (apart from the completely revamped GUI).

In Windows NT 4.0, Microsoft moved the entire Win32 subsystem to the kernel space in an attempt to improve performance. A new device driver, WIN32K.SYS, implements the Win32 API, and API calls are translated as system calls instead of IPCs. These system calls invoke the functions in the new WIN32K.SYS driver. Moving the services out of the subsystem process avoids the context switches required to process a service request. In Windows NT 3.51, each call to the Win32 subsystem involves two context switches: one from the client thread to the subsystem thread, and the second from the subsystem thread back to the client thread. Windows 2000 also continues with the kernel implementation of the Win32 subsystem.

As you will see in Chapter 8, in Windows NT 3.51 the Win32 subsystem uses quick LPC, which is supposed to be much faster than regular LPC. Still, two context switches per GDI/user call is quite a bit of overhead. In Windows NT 4.0 and Windows 2000, the GDI/user calls are processed by the kernel mode driver in the context of the calling thread, thus avoiding the context switching overheads.

THE SYSTEM CALL INTERFACE The system call interface is a very thin layer whose only job is to direct the system call requests from the user mode processes to appropriate functions in the Windows NT core. Though the layer is quite thin, it is a very important because it is the face of the core (kernel mode) component of Windows NT that the outside user-mode world sees. The system call interface defines the services offered by the core.

The key portion of the system call interface is to change the processor mode from user mode to privileged mode. On Intel platforms, this can be achieved through software interrupts. Windows NT uses the software interrupt 2Eh to implement the system call interface. The handing routine for interrupt 2Eh passes on the control to the appropriate routine in the core component, depending on the requested system service ID. NTDLL.DLL is the user mode component of the system call interface. The user mode programs call NTDLL.DLL functions (through KERNEL32.DLL functions). The NTDLL.DLL functions are stub routines that set up appropriate parameters and trigger interrupt 2Eh.. The stub functions in NTDLL.DLL also pass the system service ID to the interrupt 2Eh handler. The interrupt handler indexes the service ID in the system call table to get to the core function that fulfills the requested system service. The interrupt handler calls this

core function after copying the required parameters from the user mode stack to the kernel mode stack.

Summary

In this chapter, we discussed the overall architecture of Windows NT. Windows NT architecture is robust in the areas of portability, extensibility, compatibility, and maintainability. Features such as security, symmetric multiprocessor support, and international language support position the Windows NT operating system on the high end of the scale compared to Windows 95.

The subsystems that run in user mode and the Windows NT core that runs in kernel mode make up the operating system environment. The Win32 subsystem is the most important of the environment subsystems. The Win32 subsystem comprises the client-side DLLs and the CSRSS process. The Win32 subsystem implements the Win32 API atop the native services provided by the Windows NT core.

The Windows NT core comprises the hardware abstraction layer (HAL), the kernel, the Windows NT executive, and the system call interface. The NT executive, which forms a major portion of the NT core, consists of the Object Manager, the I/O Manager, the Security Reference Monitor, the Virtual Memory Manager, the Process Manager, and the local procedure call (LPC) facility.

The chapters that follow cover the main components of the Windows NT operating system in detail.