The Windows Operating System

William Stallings

This document is an extract from
Operating Systems: Internals and Design Principles, Sixth Edition
William Stallings
Prentice Hall 2008
ISBN-10: 0-13-600632-9 ISBN-13: 978-0-13-600632-9
http://williamstallings.com/OS/OS6e.html

2.5 MICROSOFT WINDOWS OVERVIEW

History

The story of Windows begins with a very different OS, developed by Microsoft for the first IBM personal computer and referred to as MS-DOS or PC-DOS. The initial version, DOS 1.0, was released in August 1981. It consisted of 4000 lines of assembly language source code and ran in 8 Kbytes of memory using the Intel 8086 microprocessor.

When IBM developed a hard disk-based personal computer, the PC XT, Microsoft developed DOS 2.0, released in 1983. It contained support for the hard disk and provided for hierarchical directories. Heretofore, a disk could contain only one directory of files, supporting a maximum of 64 files. While this was adequate in the era of floppy disks, it was too limited for a hard disk, and the single-directory restriction was too clumsy. This new release allowed directories to contain subdirectories as well as files. The new release also contained a richer set of commands embedded in the OS to provide functions that had to be performed by external programs provided as utilities with Release 1. Among the capabilities added were several UNIX-like features, such as I/O redirection, which is the ability to change the input or output identity for a given application, and background printing. The memory-resident portion grew to 24 Kbytes.

When IBM announced the PC AT in 1984, Microsoft introduced DOS 3.0. The AT contained the Intel 80286 processor, which provided extended addressing and memory protection features. These were not used by DOS. To remain compatible with previous releases, the OS simply used the 80286 as a "fast 8086." The OS did provide support for new keyboard and hard disk peripherals. Even so, the memory requirement grew to 36 Kbytes. There were several notable upgrades to the 3.0 release. DOS 3.1, released in 1984, contained support for networking of PCs. The size of the resident portion did not change; this was achieved by increasing the amount of the OS that could be swapped. DOS 3.3, released in 1987, provided support for the new line of IBM computers, the PS/2. Again, this release did not take advantage of the processor capabilities of the PS/2, provided by the 80286 and the 32-bit 80386 chips. The resident portion at this stage had grown to a minimum of 46 Kbytes, with more required if certain optional extensions were selected.

By this time, DOS was being used in an environment far beyond its capabilities. The introduction of the 80486 and then the Intel Pentium chip provided power and features that could not be exploited by the simple-minded DOS. Meanwhile, beginning in the early 1980s, Microsoft began development of a graphical user interface (GUI) that would be interposed between the user and DOS. Microsoft's intent was to compete with Macintosh, whose OS was unsurpassed for ease of use. By 1990, Microsoft had a version of the GUI, known as Windows 3.0, which incorporated some of the user friendly features of Macintosh. However, it was still hamstrung by the need to run on top of DOS.

After an abortive attempt by Microsoft to develop with IBM a next-generation OS, which would exploit the power of the new microprocessors and which would incorporate the ease-of-use features of Windows, Microsoft struck out on its own and developed a new OS from the ground up, Windows NT. Windows NT exploits the capabilities of contemporary microprocessors and provides multitasking in a single-user or multiple-user environment.

The first version of Windows NT (3.1) was released in 1993, with the same GUI as Windows 3.1, another Microsoft OS (the follow-on to Windows 3.0). However, NT 3.1 was a new 32-bit OS with the ability to support older DOS and Windows applications as well as provide OS/2 support.

After several versions of NT 3.x, Microsoft released NT 4.0. NT 4.0 has essentially the same internal architecture as 3.x. The most notable external change is that NT 4.0 provides the same user interface as Windows 95 (an enhanced upgrade to Windows 3.1). The major architectural change is that several graphics components that ran in user mode as part of the Win32 subsystem in 3.x have been moved into the Windows NT Executive, which runs in kernel mode. The benefit of this change is to speed up the operation of these important functions. The potential drawback is that these graphics functions now have direct access to low-level system services, which could impact the reliability of the OS.

In 2000, Microsoft introduced the next major upgrade: Windows 2000. Again, the underlying Executive and Kernel architecture is fundamentally the same as in NT 4.0, but new features have been added. The emphasis in Windows 2000 is the addition of services and functions to support distributed processing. The central element of Windows 2000's new features is Active Directory, which is a distributed directory service able to map names of arbitrary objects to any kind of information about those objects. Windows 2000 also added the plug-and-play and power-management facilities that were already in Windows 98, the successor to Windows 95. These features are particularly important for laptop computers, which frequently use docking stations and run on batteries.

One final general point to make about Windows 2000 is the distinction between Windows 2000 Server and Windows 2000 desktop. In essence, the kernel and executive architecture and services remain the same, but Server includes some services required to use as a network server.

In 2001, a new desktop version of Windows was released, known as Windows XP. Both home PC and business workstation versions of XP were offered. In 2003, Microsoft introduced a new server version, known as Windows Server 2003, supporting both 32-bit and 64-bit processors. The 64-bit versions of Server 2003 was designed specifically for the 64-bit Intel Itanium hardware. With the first service pack

update for Server 2003, Microsoft introduced support for the AMD64 processor architecture for both desktops and servers.

In 2007, the latest desktop version of Windows was released, known as Windows Vista. Vista supports both the Intel x86 and AMD x64 architectures. The main features of the release were changes to the GUI and many security improvements. The corresponding server release is Windows Server 2008.

Single-User Multitasking

Windows (from Windows 2000 onward) is a significant example of what has become the new wave in microcomputer operating systems (other examples are Linux and MacOS). Windows was driven by a need to exploit the processing capabilities of today's 32-bit and 64-bit microprocessors, which rival mainframes of just a few years ago in speed, hardware sophistication, and memory capacity.

One of the most significant features of these new operating systems is that, although they are still intended for support of a single interactive user, they are multitasking operating systems. Two main developments have triggered the need for multitasking on personal computers, workstations, and servers. First, with the increased speed and memory capacity of microprocessors, together with the support for virtual memory, applications have become more complex and interrelated. For example, a user may wish to employ a word processor, a drawing program, and a spreadsheet application simultaneously to produce a document. Without multitasking, if a user wishes to create a drawing and paste it into a word processing document, the following steps are required:

1. Open the drawing program.
2. Create the drawing and save it in a file or on a temporary clipboard.
3. Close the drawing program.
4. Open the word processing program.
5. Insert the drawing in the correct location.

If any changes are desired, the user must close the word processing program, open the drawing program, edit the graphic image, save it, close the drawing program, open the word processing program, and insert the updated image. This becomes tedious very quickly. As the services and capabilities available to users become more powerful and varied, the single-task environment becomes more clumsy and user unfriendly. In a multitasking environment, the user opens each application as needed, and leaves it open. Information can be moved around among a number of applications easily. Each application has one or more open windows, and a graphical interface with a pointing device such as a mouse allows the user to navigate quickly in this environment.

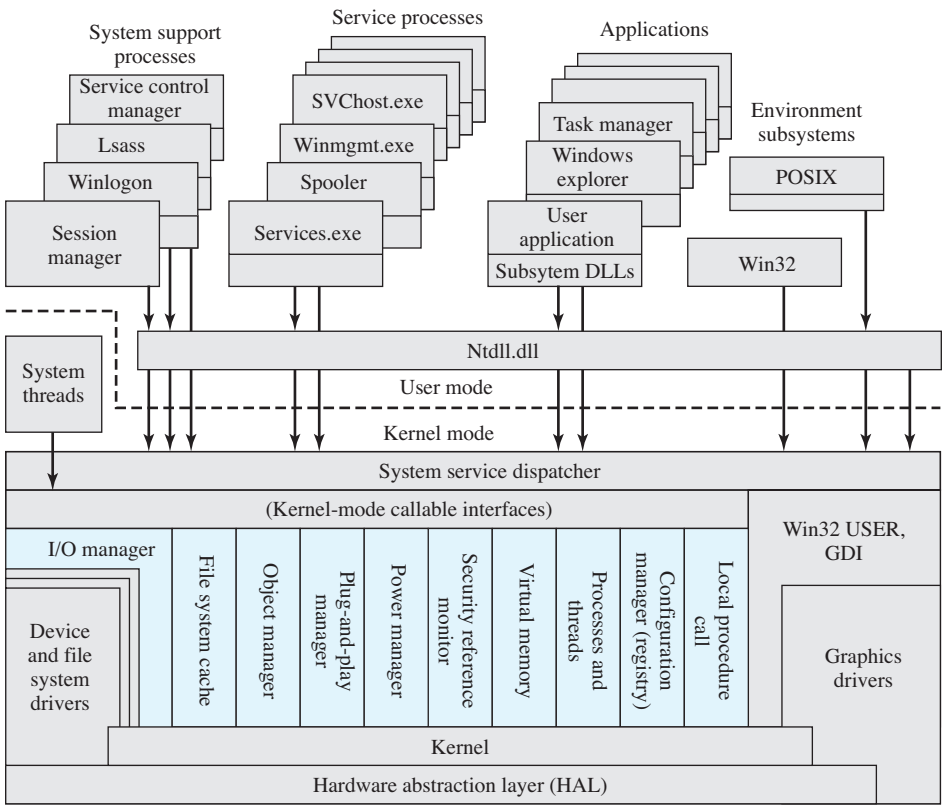
A second motivation for multitasking is the growth of client/server computing. With client/server computing, a personal computer or workstation (client) and a host system (server) are used jointly to accomplish a particular application. The two are linked, and each is assigned that part of the job that suits its capabilities. Client/server can be achieved in a local area network of personal computers and servers or by means of a link between a user system and a large host such as a mainframe. An

application may involve one or more personal computers and one or more server devices. To provide the required responsiveness, the OS needs to support high-speed networking interfaces and the associated communications protocols and data transfer architectures while at the same time supporting ongoing user interaction.

The foregoing remarks apply to the desktop versions of Windows. The Server versions are also multitasking but may support multiple users. They support multiple local server connections as well as providing shared services used by multiple users on the network. As an Internet server, Windows may support thousands of simultaneous Web connections.

Architecture

Figure 2.13 illustrates the overall structure of Windows 2000; later releases of Windows, including Vista, have essentially the same structure at this level of detail. Its modular structure gives Windows considerable flexibility. It is designed to execute



Lsass = local security authentication server
POSIX = portable operating system interface
GDI = graphics device interface
DLL = dynamic link libraries

Colored area indicates Executive

Figure 2.13 Windows and Windows Vista Architecture [RUSS05]

on a variety of hardware platforms and supports applications written for a variety of other operating systems. As of this writing, desktop Windows is only implemented on the Intel x86 and AMD64 hardware platforms. Windows server also supports the Intel IA64 (Itanium).

As with virtually all operating systems, Windows separates application-oriented software from the core OS software. The latter, which includes the Executive, the Kernel, device drivers, and the hardware abstraction layer, runs in kernel mode. Kernel mode software has access to system data and to the hardware. The remaining software, running in user mode, has limited access to system data.

Operating System Organization Windows has a highly modular architecture. Each system function is managed by just one component of the OS. The rest of the OS and all applications access that function through the responsible component using standard interfaces. Key system data can only be accessed through the appropriate function. In principle, any module can be removed, upgraded, or replaced without rewriting the entire system or its standard application program interface (APIs).

The kernel-mode components of Windows are the following:

- **Executive:** Contains the base OS services, such as memory management, process and thread management, security, I/O, and interprocess communication.
- **Kernel:** Controls execution of the processor(s). The Kernel manages thread scheduling, process switching, exception and interrupt handling, and multiprocessor synchronization. Unlike the rest of the Executive and the user level, the Kernel's own code does not run in threads.
- **Hardware abstraction layer (HAL):** Maps between generic hardware commands and responses and those unique to a specific platform. It isolates the OS from platform-specific hardware differences. The HAL makes each computer's system bus, direct memory access (DMA) controller, interrupt controller, system timers, and memory module look the same to the Executive and Kernel components. It also delivers the support needed for symmetric multiprocessing (SMP), explained subsequently.
- **Device drivers:** Dynamic libraries that extend the functionality of the Executive. These include hardware device drivers that translate user I/O function calls into specific hardware device I/O requests and software components for implementing file systems, network protocols, and any other system extensions that need to run in kernel mode.
- **Windowing and graphics system:** Implements the graphical user interface (GUI) functions, such as dealing with windows, user interface controls, and drawing.

The Windows Executive includes components for specific system functions and provides an API for user-mode software. Following is a brief description of each of the Executive modules:

- **I/O manager:** Provides a framework through which I/O devices are accessible to applications, and is responsible for dispatching to the appropriate device drivers for further processing. The I/O manager implements all the Windows I/O APIs and enforces security and naming for devices, network protocols, and file systems (using the object manager). Windows I/O is discussed in Chapter 11.

- **Cache manager:** Improves the performance of file-based I/O by causing recently referenced file data to reside in main memory for quick access, and by deferring disk writes by holding the updates in memory for a short time before sending them to the disk.
- **Object manager:** Creates, manages, and deletes Windows Executive objects and abstract data types that are used to represent resources such as processes, threads, and synchronization objects. It enforces uniform rules for retaining, naming, and setting the security of objects. The object manager also creates object handles, which consist of access control information and a pointer to the object. Windows objects are discussed later in this section.
- **Plug-and-play manager:** Determines which drivers are required to support a particular device and loads those drivers.
- **Power manager:** Coordinates power management among various devices and can be configured to reduce power consumptionby shutting down idle devices, putting the processor to sleep, and even writing all of memory to disk and shutting off power to the entire system.
- **Security reference monitor:** Enforces access-validation and audit-generation rules. The Windows object-oriented model allows for a consistent and uniform view of security, right down to the fundamental entities that make up the Executive. Thus, Windows uses the same routines for access validation and for audit checks for all protected objects, including files, processes, address spaces, and I/O devices. Windows security is discussed in Chapter 15.
- **Virtual memory manager:** Manages virtual addresses, physical memory, and the paging files on disk. Controls the memory management hardware and data structures which map virtual addresses in the process's address space to physical pages in the computer's memory. Windows virtual memory management is described in Chapter 8.
- **Process/thread manager:** Creates, manages, and deletes process and thread objects. Windows process and thread management are described in Chapter 4.
- **Configuration manager:** Responsible for implementing and managing the system registry, which is the repository for both system wide and per-user settings of various parameters.
- **Local procedure call (LPC) facility:** Implements an efficient cross-process procedure call mechanism for communication between local processes implementing services and subsystems. Similar to the remote procedure call (RPC) facility used for distributed processing.

User-Mode Processes Four basic types of user-mode processes are supported by Windows:

- **Special system processes:** User mode services needed to manage the system, such as the session manager, the authentication subsystem, the service manager, and the logon process
- **Service processes:** The printer spooler, the event logger, user mode components that cooperate with device drivers, various network services, and many, many others. Services are used by both Microsoft and external software developers to

extend system functionality as they are the only way to run background user mode activity on a Windows system.

- **Environment subsystems:** Provide different OS personalities (environments). The supported subsystems are Win32/WinFX and POSIX. Each environment subsystem includes a subsystem process shared among all applications using the subsystem and dynamic link libraries (DLLs) that convert the user application calls to LPC calls on the subsystem process, and/or native Windows calls.
- **User applications:** Executables (EXEs) and DLLs that provide the functionality users run to make use of the system. EXEs and DLLs are generally targeted at a specific environment subsystems; although some of the programs that are provided as part of the OS use the native system interfaces (NTAPI). There is also support for running 16-bit programs written for Windows 3.1 or MS-DOS.

Windows is structured to support applications written for multiple OS personalities. Windows provides this support using a common set of kernel mode components that underlie the protected environment subsystems. The implementation of each subsystem includes a separate process, which contains the shared data structures, privileges, and Executive object handles needed to implement a particular personality. The process is started by the Windows Session Manager when the first application of that type is started. The subsystem process runs as a system user, so the Executive will protect its address space from processes run by ordinary users.

A protected subsystem provides a graphical or command-line user interface that defines the look and feel of the OS for a user. In addition, each protected subsystem provides the API for that particular operating environment. This means that applications created for a particular operating environment may run unchanged on Windows, because the OS interface that they see is the same as that for which they were written.

The most important subsystem is Win32. Win32 is the API implemented on both Windows NT and Windows 95 and later releases of Windows 9x. Many Win32 applications written for the Windows 9x line of operating systems run on NT systems unchanged. At the release of Windows XP, Microsoft focused on improving compatibility with Windows 9x so that enough applications (and device drivers) would run that they could cease any further support for 9x and focus on NT.

The most recent programming API for Windows is WinFX, which is based on Microsoft's .NET programming model. WinFX is implemented in Windows as a layer on top of Win32 and not as a distinct subsystem type

Client/Server Model

The Windows operating system services, the protected subsystems, and the applications are structured using the client/server computing model, which is a common model for distributed computing and which is discussed in Part Six. This same architecture can be adopted for use internal to a single system, as is the case with Windows.

The native NT API is a set of kernel-based services which provide the core abstractions used by the system, such as processes, threads, virtual memory, I/O, and communication. Windows provides a far richer set of services by using the client/server model to implement functionality in user-mode processes. Both the environment

subsystems and the Windows user-mode services are implemented as processes that communicate with clients via RPC. Each server process waits for a request from a client for one of its services (for example, memory services, process creation services, or networking services). A client, which can be an application program or another server program, requests a service by sending a message. The message is routed through the Executive to the appropriate server. The server performs the requested operation and returns the results or status information by means of another message, which is routed through the Executive back to the client.

Advantages of a client/server architecture include the following:

- It simplifies the Executive. It is possible to construct a variety of APIs implemented in user-mode servers without any conflicts or duplications in the Executive. New APIs can be added easily.
- It improves reliability. Each new server runs outside of the kernel, with its own partition of memory, protected from other servers. A single server can fail without crashing or corrupting the rest of the OS.
- It provides a uniform means for applications to communicate with services via RPCs without restricting flexibility. The message-passing process is hidden from the client applications by function stubs, which are small pieces of code which wrap the RPC call. When an application makes an API call to an environment subsystem or service, the stub in the client application packages the parameters for the call and sends them as a message to a server subsystem that implements the call.
- It provides a suitable base for distributed computing. Typically, distributed computing makes use of a client/server model, with remote procedure calls implemented using distributed client and server modules and the exchange of messages between clients and servers. With Windows, a local server can pass a message on to a remote server for processing on behalf of local client applications. Clients need not know whether a request is serviced locally or remotely. Indeed, whether a request is serviced locally or remotely can change dynamically based on current load conditions and on dynamic configuration changes.

Threads and SMP

Two important characteristics of Windows are its support for threads and for symmetric multiprocessing (SMP), both of which were introduced in Section 2.4. [RUSS05] lists the following features of Windows that support threads and SMP:

- OS routines can run on any available processor, and different routines can execute simultaneously on different processors.
- Windows supports the use of multiple threads of execution within a single process. Multiple threads within the same process may execute on different processors simultaneously.
- Server processes may use multiple threads to process requests from more than one client simultaneously.
- Windows provides mechanisms for sharing data and resources between processes and flexible interprocess communication capabilities.

Windows Objects

Windows draws heavily on the concepts of object-oriented design. This approach facilitates the sharing of resources and data among processes and the protection of resources from unauthorized access. Among the key object-oriented concepts used by Windows are the following:

- **Encapsulation:** An object consists of one or more items of data, called attributes, and one or more procedures that may be performed on those data, called services. The only way to access the data in an object is by invoking one of the object's services. Thus, the data in the object can easily be protected from unauthorized use and from incorrect use (e.g., trying to execute a nonexecutable piece of data).
- **Object class and instance:** An object class is a template that lists the attributes and services of an object and defines certain object characteristics. The OS can create specific instances of an object class as needed. For example, there is a single process object class and one process object for every currently active process. This approach simplifies object creation and management.
- **Inheritance:** Although the implementation is hand coded, the Executive uses inheritance to extend object classes by adding new features. Every Executive class is based on a base class which specifies virtual methods that support creating, naming, securing, and deleting objects. Dispatcher objects are Executive objects that inherit the properties of an event object, so they can use common synchronization methods. Other specific object types, such as the device class, allow classes for specific devices to inherit from the base class, and add additional data and methods.
- **Polymorphism:** Internally, Windows uses a common set of API functions to manipulate objects of any type; this is a feature of polymorphism, as defined in Appendix B. However, Windows is not completely polymorphic because there are many APIs that are specific to specific object types.

The reader unfamiliar with object-oriented concepts should review Appendix B at the end of this book.

Not all entities in Windows are objects. Objects are used in cases where data are intended for user mode access or when data access is shared or restricted. Among the entities represented by objects are files, processes, threads, semaphores, timers, and windows. Windows creates and manages all types of objects in a uniform way, via the object manager. The object manager is responsible for creating and destroying objects on behalf of applications and for granting access to an object's services and data.

Each object within the Executive, sometimes referred to as a kernel object (to distinguish from user-level objects not of concern to the Executive), exists as a memory block allocated by the kernel and is directly accessible only by kernel mode components. Some elements of the data structure (e.g., object name, security parameters, usage count) are common to all object types, while other elements are specific to a particular object type (e.g., a thread object's priority). Because these object data structures are in the part of each process's address space accessible only by the kernel, it is impossible for an application to reference these data structures and read or write them directly. Instead, applications manipulate objects indirectly through the set of object manipulation functions supported by the Executive. When an object is

created, the application that requested the creation receives back a handle for the object. In essence a handle is an index into a Executive table containing a pointer to the referenced object. This handle can then be used by any thread within the same process to invoke Win32 functions that work with objects, or can be duplicated into other processes.

Objects may have security information associated with them, in the form of a Security Descriptor (SD). This security information can be used to restrict access to the object based on contents of a token object which describes a particular user. For example, a process may create a named semaphore object with the intent that only certain users should be able to open and use that semaphore. The SD for the semaphore object can list those users that are allowed (or denied) access to the semaphore object along with the sort of access permitted (read, write, change, etc.).

In Windows, objects may be either named or unnamed. When a process creates an unnamed object, the object manager returns a handle to that object, and the handle is the only way to refer to it. Named objects are also given a name that other processes can use to obtain a handle to the object. For example, if process A wishes to synchronize with process B, it could create a named event object and pass the name of the event to B. Process B could then open and use that event object. However, if A simply wished to use the event to synchronize two threads within itself, it would create an unnamed event object, because there is no need for other processes to be able to use that event.

There are two categories of objects used by Windows for synchronizing the use of the processor:

- **Dispatcher objects:** The subset of Executive objects which threads can wait on to control the dispatching and synchronization of thread-based system operations. These are described in Chapter 6.
- **Control objects:** Used by the Kernel component to manage the operation of the processor in areas not managed by normal thread scheduling. Table 2.5 lists the Kernel control objects.

Table 2.5 Windows Kernel Control Objects

Asynchronous Procedure Call	Used to break into the execution of a specified thread and to cause a procedure to be called in a specified processor mode.
Deferred Procedure Call	Used to postpone interrupt processing to avoid delaying hardware interrupts. Also used to implement timers and inter-processor communication
Interrupt	Used to connect an interrupt source to an interrupt service routine by means of an entry in an Interrupt Dispatch Table (IDT). Each processor has an IDT that is used to dispatch interrupts that occur on that processor.
Process	Represents the virtual address space and control information necessary for the execution of a set of thread objects. A process contains a pointer to an address map, a list of ready threads containing thread objects, a list of threads belonging to the process, the total accumulated time for all threads executing within the process, and a base priority.
Thread	Represents thread objects, including scheduling priority and quantum, and which processors the thread may run on.
Profile	Used to measure the distribution of run time within a block of code. Both user and system code can be profiled.

Windows is not a full-blown object-oriented OS. It is not implemented in an object-oriented language. Data structures that reside completely within one Executive component are not represented as objects. Nevertheless, Windows illustrates the power of object-oriented technology and represents the increasing trend toward the use of this technology in OS design.

4.4 WINDOWS THREAD AND SMP MANAGEMENT

Windows process design is driven by the need to provide support for a variety of OS environments. Processes supported by different OS environments differ in a number of ways, including the following:

- How processes are named
- Whether threads are provided within processes
- How processes are represented
- How process resources are protected
- What mechanisms are used for interprocess communication and synchronization
- How processes are related to each other

Accordingly, the native process structures and services provided by the Windows Kernel are relatively simple and general purpose, allowing each OS subsystem to emulate a particular process structure and functionality. Important characteristics of Windows processes are the following:

- Windows processes are implemented as objects.
- An executable process may contain one or more threads.
- Both process and thread objects have built-in synchronization capabilities.

Figure 4.12, based on one in [RUSS05], illustrates the way in which a process relates to the resources it controls or uses. Each process is assigned a security

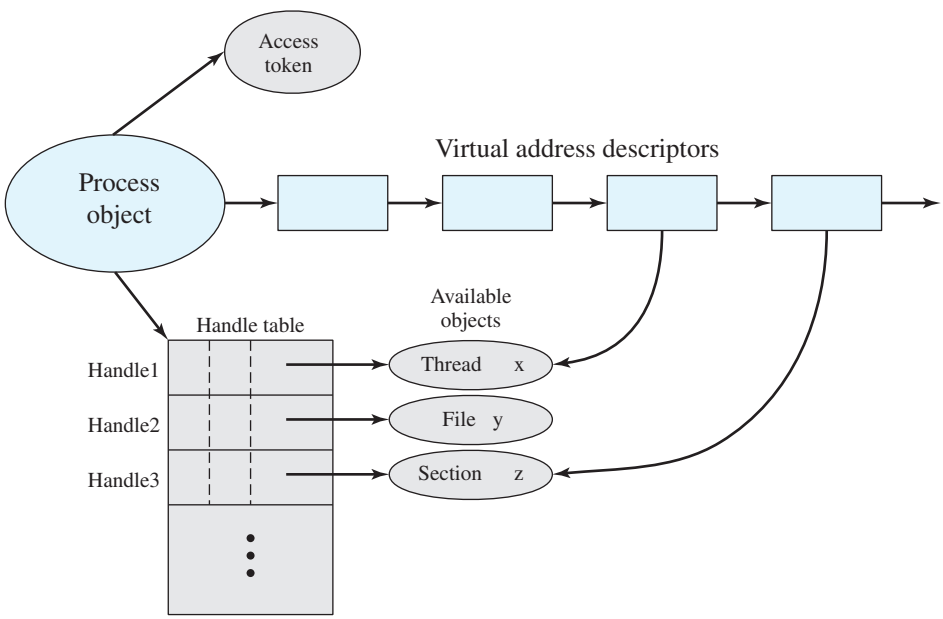


Figure 4.12 A Windows Process and Its Resources

access token, called the primary token of the process. When a user first logs on, Windows creates an access token that includes the security ID for the user. Every process that is created by or runs on behalf of this user has a copy of this access token. Windows uses the token to validate the user's ability to access secured objects or to perform restricted functions on the system and on secured objects. The access token controls whether the process can change its own attributes. In this case, the process does not have a handle opened to its access token. If the process attempts to open such a handle, the security system determines whether this is permitted and therefore whether the process may change its own attributes.

Also related to the process is a series of blocks that define the virtual address space currently assigned to this process. The process cannot directly modify these structures but must rely on the virtual memory manager, which provides a memory-allocation service for the process.

Finally, the process includes an object table, with handles to other objects known to this process. One handle exists for each thread contained in this object. Figure 4.12 shows a single thread. In addition, the process has access to a file object and to a section object that defines a section of shared memory.

Process and Thread Objects

The object-oriented structure of Windows facilitates the development of a general-purpose process facility. Windows makes use of two types of process-related objects: processes and threads. A process is an entity corresponding to a user job or application that owns resources, such as memory, and opens files. A thread is a dispatchable unit of work that executes sequentially and is interruptible, so that the processor can turn to another thread.

Each Windows process is represented by an object whose general structure is shown in Figure 4.13a. Each process is defined by a number of attributes and encapsulates a number of actions, or services, that it may perform. A process will perform a service when called upon through a set of published interface methods. When Windows creates a new process, it uses the object class, or type, defined for the Windows process as a template to generate a new object instance. At the time of creation, attribute values are assigned. Table 4.3 gives a brief definition of each of the object attributes for a process object.

A Windows process must contain at least one thread to execute. That thread may then create other threads. In a multiprocessor system, multiple threads from the same process may execute in parallel. Figure 4.13b depicts the object structure for a thread object, and Table 4.4 defines the thread object attributes. Note that some of the attributes of a thread resemble those of a process. In those cases, the thread attribute value is derived from the process attribute value. For example, the *thread processor affinity* is the set of processors in a multiprocessor system that may execute this thread; this set is equal to or a subset of the *process processor affinity*.

Note that one of the attributes of a thread object is context. This information enables threads to be suspended and resumed. Furthermore, it is possible to alter the behavior of a thread by altering its context when it is suspended.

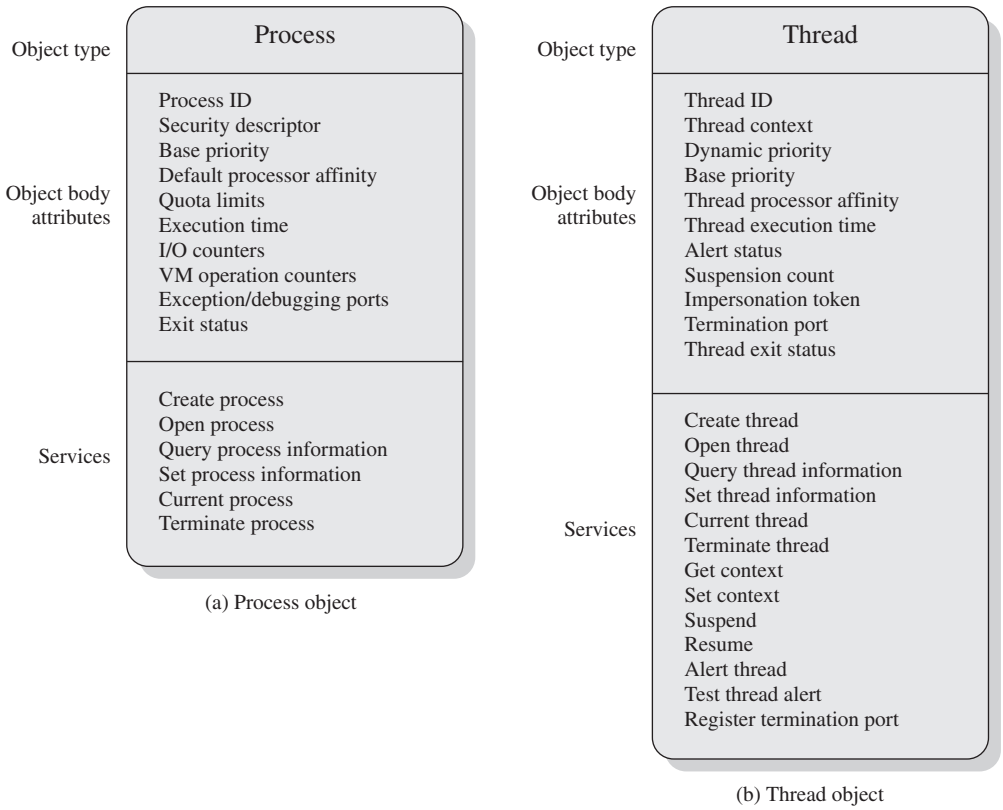


Figure 4.13 Windows Process and Thread Objects

Multithreading

Windows supports concurrency among processes because threads in different processes may execute concurrently. Moreover, multiple threads within the same process may be allocated to separate processors and execute simultaneously. A multithreaded process achieves concurrency without the overhead of using multiple processes. Threads within the same process can exchange information through their common address space and have access to the shared resources of the process. Threads in different processes can exchange information through shared memory that has been set up between the two processes.

An object-oriented multithreaded process is an efficient means of implementing a server application. For example, one server process can service a number of clients.

Thread States

An existing Windows thread is in one of six states (Figure 4.14):

- **Ready:** May be scheduled for execution. The Kernel dispatcher keeps track of all ready threads and schedules them in priority order.

Table 4.3 Windows Process Object Attributes

Process ID	A unique value that identifies the process to the operating system.
Security Descriptor	Describes who created an object, who can gain access to or use the object, and who is denied access to the object.
Base priority	A baseline execution priority for the process's threads.
Default processor affinity	The default set of processors on which the process's threads can run.
Quota limits	The maximum amount of paged and nonpaged system memory, paging file space, and processor time a user's processes can use.
Execution time	The total amount of time all threads in the process have executed.
I/O counters	Variables that record the number and type of I/O operations that the process's threads have performed.
VM operation counters	Variables that record the number and types of virtual memory operations that the process's threads have performed.
Exception/debugging ports	Interprocess communication channels to which the process manager sends a message when one of the process's threads causes an exception. Normally these are connected to environment subsystem and debugger processes, respectively.
Exit status	The reason for a process's termination.

- **Standby:** A standby thread has been selected to run next on a particular processor. The thread waits in this state until that processor is made available. If the standby thread's priority is high enough, the running thread on that processor may be preempted in favor of the standby thread. Otherwise, the standby thread waits until the running thread blocks or exhausts its time slice.

Table 4.4 Windows Thread Object Attributes

Thread ID	A unique value that identifies a thread when it calls a server.
Thread context	The set of register values and other volatile data that defines the execution state of a thread.
Dynamic priority	The thread's execution priority at any given moment.
Base priority	The lower limit of the thread's dynamic priority.
Thread processor affinity	The set of processors on which the thread can run, which is a subset or all of the processor affinity of the thread's process.
Thread execution time	The cumulative amount of time a thread has executed in user mode and in kernel mode.
Alert status	A flag that indicates whether a waiting thread may execute an asynchronous procedure call.
Suspension count	The number of times the thread's execution has been suspended without being resumed.
Impersonation token	A temporary access token allowing a thread to perform operations on behalf of another process (used by subsystems).
Termination port	An interprocess communication channel to which the process manager sends a message when the thread terminates (used by subsystems).
Thread exit status	The reason for a thread's termination.

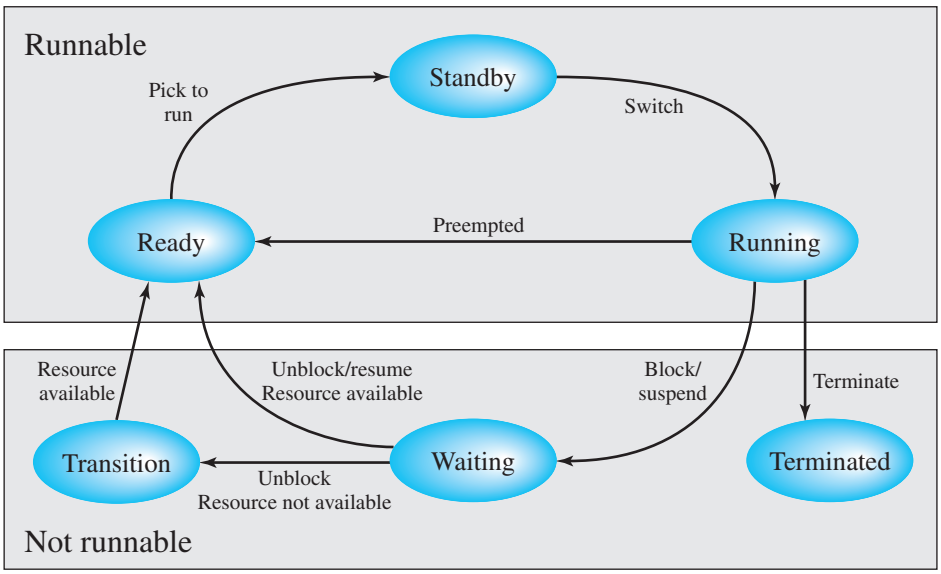


Figure 4.14 **Windows Thread States**

- **Running:** Once the Kernel dispatcher performs a thread switch, the standby thread enters the Running state and begins execution and continues execution until it is preempted by a higher priority thread, exhausts its time slice, blocks, or terminates. In the first two cases, it goes back to the ready state.
- **Waiting:** A thread enters the Waiting state when (1) it is blocked on an event (e.g., I/O), (2) it voluntarily waits for synchronization purposes, or (3) an environment subsystem directs the thread to suspend itself. When the waiting condition is satisfied, the thread moves to the Ready state if all of its resources are available.
- **Transition:** A thread enters this state after waiting if it is ready to run but the resources are not available. For example, the thread's stack may be paged out of memory. When the resources are available, the thread goes to the Ready state.
- **Terminated:** A thread can be terminated by itself, by another thread, or when its parent process terminates. Once housekeeping chores are completed, the thread is removed from the system, or it may be retained by the executive[9] for future reinitialization.

Support for OS Subsystems

The general-purpose process and thread facility must support the particular process and thread structures of the various OS clients. It is the responsibility of each OS

[9]The Windows executive is described in Chapter 2. It contains the base operating system services, such as memory management, process and thread management, security, I/O, and interprocess communication.

subsystem to exploit the Windows process and thread features to emulate the process and thread facilities of its corresponding OS. This area of process/thread management is complicated, and we give only a brief overview here.

Process creation begins with a request for a new process from an application. The application issues a create-process request to the corresponding protected subsystem, which passes the request to the Windows executive. The executive creates a process object and returns a handle to that object to the subsystem. When Windows creates a process, it does not automatically create a thread. In the case of Win32, a new process is always created with a thread. Therefore, for these operating systems, the subsystem calls the Windows process manager again to create a thread for the new process, receiving a thread handle back from Windows. The appropriate thread and process information are then returned to the application. In the case of 16-bit Windows and POSIX, threads are not supported. Therefore, for these operating systems, the subsystem obtains a thread for the new process from Windows so that the process may be activated but returns only process information to the application. The fact that the application process is implemented using a thread is not visible to the application.

When a new process is created in Win32, the new process inherits many of its attributes from the creating process. However, in the Windows environment, this process creation is done indirectly. An application client process issues its process creation request to the OS subsystem; then a process in the subsystem in turn issues a process request to the Windows executive. Because the desired effect is that the new process inherits characteristics of the client process and not of the server process, Windows enables the subsystem to specify the parent of the new process. The new process then inherits the parent's access token, quota limits, base priority, and default processor affinity.

Symmetric Multiprocessing Support

Windows supports an SMP hardware configuration. The threads of any process, including those of the executive, can run on any processor. In the absence of affinity restrictions, explained in the next paragraph, the microkernel assigns a ready thread to the next available processor. This assures that no processor is idle or is executing a lower-priority thread when a higher-priority thread is ready. Multiple threads from the same process can be executing simultaneously on multiple processors.

As a default, the microkernel uses the policy of **soft affinity** in assigning threads to processors: The dispatcher tries to assign a ready thread to the same processor it last ran on. This helps reuse data still in that processor's memory caches from the previous execution of the thread. It is possible for an application to restrict its thread execution to certain processors (**hard affinity**).

6.10 WINDOWS CONCURRENCY MECHANISMS

Windows provides synchronization among threads as part of the object architecture. The most important methods of synchronization are Executive dispatcher objects, user mode critical sections, slim reader-writer locks, and condition variables. Dispatcher objects make use of wait functions. We first describe wait functions and then look at the synchronization methods.

Wait Functions

The wait functions allow a thread to block its own execution. The wait functions do not return until the specified criteria have been met. The type of wait function determines the set of criteria used. When a wait function is called, it checks whether the wait criteria have been met. If the criteria have not been met, the calling thread enters the wait state. It uses no processor time while waiting for the criteria to be met.

The most straightforward type of wait function is one that waits on a single object. The `WaitForSingleObject` function requires a handle to one synchronization object. The function returns when one of the following occurs:

- The specified object is in the signaled state.
- The time-out interval elapses. The time-out interval can be set to `INFINITE` to specify that the wait will not time out.

Dispatcher Objects

The mechanism used by the Windows Executive to implement synchronization facilities is the family of dispatcher objects, which are listed with brief descriptions in Table 6.7.

The first five object types in the table are specifically designed to support synchronization. The remaining object types have other uses but also may be used for synchronization.

Each dispatcher object instance can be in either a signaled or unsignaled state. A thread can be blocked on an object in an unsignaled state; the thread is released when the object enters the signaled state. The mechanism is straightforward: A thread issues

WINDOWS/LINUX COMPARISON	
Windows	**Linux**
Common synchronization primitives, such as semaphores, mutexes, spinlocks, timers, based on an underlying wait/signal mechanism	Common synchronization primitives, such as semaphores, mutexes, spinlocks, timers, based on an underlying sleep/wakeup mechanism
Many kernel objects are also *dispatcher objects*, meaning that threads can synchronize with them using a common event mechanism, available at user-mode. Process and thread termination are events, I/O completion is an event	
Threads can wait on multiple dispatcher objects at the same time	Processes can use the select() system call to wait on I/O from up to 64 file descriptors
User-mode reader/writer locks and condition variables are supported	User-mode reader/writer locks and condition variables are supported
Many hardware atomic operations, such as atomic increment/decrement, and compare-and-swap, are supported	Many hardware atomic operations, such as atomic increment/decrement, and compare-and-swap, are supported
A non-locking atomic LIFO queue, called an SLIST, is supported using compare-and-swap; widely used in the OS and also available to user programs	
A large variety of synchronization mechanisms exist within the kernel to improve scalability. Many are based on simple compare-and-swap mechanisms, such as push-locks and fast references of objects	
Named pipes, and sockets support remote procedure calls (RPCs), as does an efficient Local Procedure Call mechanism (ALPC), used within a local system. ALPC is used heavily for communicating between clients and local services	Named pipes, and sockets support remote procedure calls (RPCs)
Asynchronous Procedure Calls (APCs) are used heavily within the kernel to get threads to act upon themselves (e.g. termination and I/O completion use APCs since these operations are easier to implement in the context of a thread rather than cross-thread). APCs are also available for user-mode, but user-mode APCs are only delivered when a user-mode thread blocks in the kernel	Unix supports a general *signal* mechanism for communication between processes. Signals are modeled on hardware interrupts and can be delivered at any time that they are not blocked by the receiving process; like with hardware interrupts, signal semantics are complicated by multi-threading
Hardware support for deferring interrupt processing until the interrupt level has dropped is provided by the Deferred Procedure Call (DPC) control object	Uses *tasklets* to defer interrupt processing until the interrupt level has dropped

Table 6.7 Windows Synchronization Objects

Object Type	Definition	Set to Signaled State When	Effect on Waiting Threads
Notification Event	An announcement that a system event has occurred	Thread sets the event	All released
Synchronization event	An announcement that a system event has occurred.	Thread sets the event	One thread released
Mutex	A mechanism that provides mutual exclusion capabilities; equivalent to a binary semaphore	Owning thread or other thread releases the mutex	One thread released
Semaphore	A counter that regulates the number of threads that can use a resource	Semaphore count drops to zero	All released
Waitable timer	A counter that records the passage of time	Set time arrives or time interval expires	All released
File	An instance of an opened file or I/O device	I/O operation completes	All released
Process	A program invocation, including the address space and resources required to run the program	Last thread terminates	All released
Thread	An executable entity within a process	Thread terminates	All released

Note: Shaded rows correspond to objects that exist for the sole purpose of synchronization.

a wait request to the Windows Executive, using the handle of the synchronization object. When an object enters the signaled state, the Windows Executive releases one or all of the thread objects that are waiting on that dispatcher object.

The **event object** is useful in sending a signal to a thread indicating that a particular event has occurred. For example, in overlapped input and output, the system sets a specified event object to the signaled state when the overlapped operation has been completed. The **mutex object** is used to enforce mutually exclusive access to a resource, allowing only one thread object at a time to gain access. It therefore functions as a binary semaphore. When the mutex object enters the signaled state, only one of the threads waiting on the mutex is released. Mutexes can be used to synchronize threads running in different processes. Like mutexes, **semaphore objects** may be shared by threads in multiple processes. The Windows semaphore is a counting semaphore. In essence, the **waitable timer object** signals at a certain time and/or at regular intervals.

Critical Sections

Critical sections provide a synchronization mechanism similar to that provided by mutex objects, except that critical sections can be used only by the threads of a single process. Event, mutex, and semaphore objects can also be used in a single-process application, but critical sections provide a much faster, more efficient mechanism for mutual-exclusion synchronization.

The process is responsible for allocating the memory used by a critical section. Typically, this is done by simply declaring a variable of type `CRITICAL_SECTION`. Before the threads of the process can use it, initialize the critical section by using the `InitializeCriticalSection or InitializeCriticalSectionAndSpinCount` function.

A thread uses the `EnterCriticalSection` or `TryEnterCriticalSection` function to request ownership of a critical section. It uses the `LeaveCriticalSection` function to release ownership of a critical section. If the critical section is currently owned by another thread, `EnterCriticalSection` waits indefinitely for ownership. In contrast, when a mutex object is used for mutual exclusion, the wait functions accept a specified time-out interval. The `TryEnterCriticalSection` function attempts to enter a critical section without blocking the calling thread.

Critical sections use a sophisticated algorithm when trying to acquire the mutex. If the system is a multiprocessor, the code will attempt to acquire a spin-lock. This works well in situations where the critical section is acquired for only a short time. Effectively the spinlock optimizes for the case where the thread that currently owns the critical section is executing on another processor. If the spinlock cannot be acquired within a reasonable number of iterations, a dispatcher object is used to block the thread so that the Kernel can dispatch another thread onto the processor. The dispatcher object is only allocated as a last resort. Most critical sections are needed for correctness, but in practice are rarely contended. By lazily allocating the dispatcher object the system saves significant amounts of kernel virtual memory.

Slim Read-Writer Locks and Condition Variables

Windows Vista added a user mode reader-writer. Like critical sections, the reader-writer lock enters the kernel to block only after attempting to use a spin-lock. It is *slim* in the sense that it normally only requires allocation of a single pointer-sized piece of memory.

To use an SRW a process declares a variable of type SRWLOCK and a calls `InitializeSRWLock` to initialize it. Threads call `AcquireSRWLockExclusive` or `AcquireSRWLockShared` to acquire the lock and `ReleaseSRWLockExclusive` or `ReleaseSRWLockShared` to release it.

Windows Vista also added condition variables. The process must declare a CONDITION_VARIABLE and initialize it in some thread by calling `InitializeConditionVariable`. Condition variables can be used with either critical sections or SRW locks, so there are two methods, `SleepConditionVariableCS` and `SleepConditionVariableSRW`, which sleep on the specified condition and releases the specified lock as an atomic operation.

There are two wake methods, `WakeConditionVariable` and `WakeAllConditionVariable`, which wake one or all of the sleeping threads, respectively. Condition variables are used as follows:

1. Acquire exclusive lock
2. While (predicate() == FALSE) SleepConditionVariable()
3. Perform the protected operation
4. Release the lock

8.5 WINDOWS MEMORY MANAGEMENT

The Windows virtual memory manager controls how memory is allocated and how paging is performed. The memory manager is designed to operate over a variety of platforms and use page sizes ranging from 4 Kbytes to 64 Kbytes. Intel and AMD64 platforms have 4096 bytes per page and Intel Itanium platforms have 8192 bytes per page.

Windows Virtual Address Map

On 32-bit platforms, each Windows user process sees a separate 32-bit address space, allowing 4 Gbytes of virtual memory per process. By default, a portion of this memory is reserved for the operating system, so each user actually has 2 Gbytes of available virtual address space and all processes share the same 2 Gbytes of system space. There an option that allows user space to be increased to 3 Gbytes, leaving 1 Gbyte for system space. This feature is intended to support large memory-intensive applications on servers with multiple gigabytes of RAM, and that the use of the larger address space can dramatically improve performance for applications such as decision support or data mining.

---

[7]The page cache has properties similar to a disk buffer, described in this chapter, as well as a disk cache, described in Chapter 11. We defer a discussion of the Linux page cache to Chapter 11.

WINDOWS/LINUX COMPARISON	
Windows	**Linux**
Physical Memory dynamically mapped into kernel address space as needed	Up to 896MB physical memory statically mapped into kernel address space (32-bit), with rest dynamically mapped into a fixed 128MB of kernel addresses, which can include non-contiguous use
Kernel and applications can use x86 *large pages* for TLB efficiency	
Much of code and data for kernel and drivers is pageable; initialization code deleted after boot; page tables are fully pageable	Kernel is non-paged; modules are non-paged, but can be unloaded
User-mode allocation of virtual addresses separated from mapping addresses as a view of a physical object (files, devices, physical memory)	User-mode addresses directly mapped to physical objects
Physical memory can be allocated to large applications, and directly managed by efficiently mapping/unmapping into the address space using Address Windowing Extensions (AWE) – which is much like old-fashion overlays [not needed with 64-bit]	
Copy-on-write support	Copy-on-write support
Normal user/kernel split is 2GB/2GB; Windows can be booted to give 3GB/1GB	Normal user/kernel split is 3GB/1GB; Linux can run kernel and user in separate address spaces, giving user up to 4GB
Cache manager manages memory mapping of files into kernel address space, using virtual memory manager to do actual paging and caching of pages in the standby and modified lists of pages	Page cache implements caching of pages and used as lookaside cache for paging system
Threads can do direct I/O to bypass cache manager views	Processes can do direct I/O to bypass page cache
Page Frame Number (PFN) database is central data structure. Pages in PFN are either in a process page table or linked into one of several lists: standby, modified, free, bad	Pages removed from process address spaces kept in page cache
Section Objects describe map-able memory objects, like files, and include pageable, create-on-demand prototype page table which can be used to uniquely locate pages, including when faulted pages are already in transition	Swap Cache used to manage multiple instances of faulting the same page
Page replacement is based on working sets, for both processes and the kernel-mode (the system process)	Page replacement uses a global clock algorithm
Security features for encrypting page files, and clearing pages when freed	
Allocate space in paging file as needed, so writes can be localized for a group of freed pages; shared pages use indirection through prototype page tables associated with section object, so pagefile space can be freed immediately	Allocate space in swap disk as needed, so writes can be localized for a group of freed pages; shared pages keep swap slot until all processes the slot have faulted the page back in

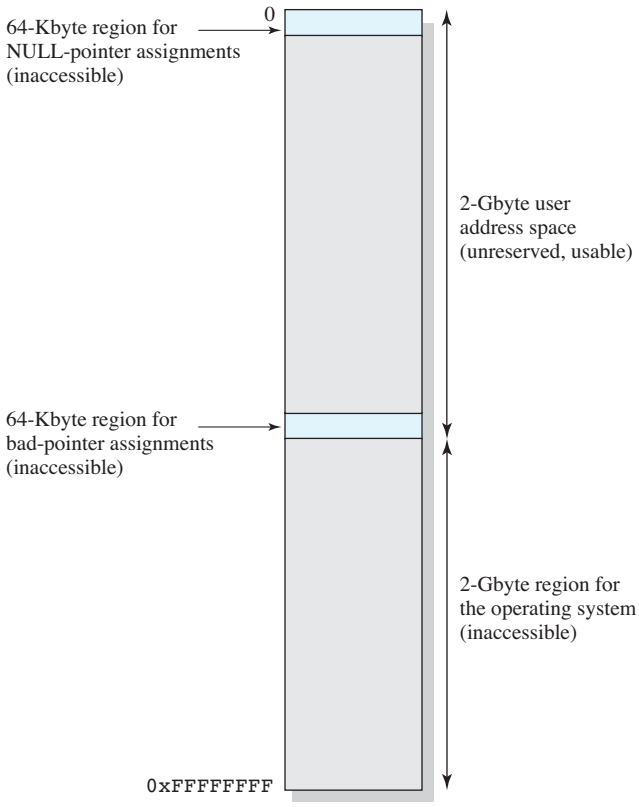


Figure 8.26 Windows Default 32-Bit Virtual Address Space

Figure 8.26 shows the default virtual address space seen by a normal 32-bit user process. It consists of four regions:

- **0x00000000 to 0x0000FFFF:** Set aside to help programmers catch NULL-pointer assignments.
- **0x00010000 to 0x7FFEFFFF:** Available user address space. This space is divided into pages that may be loaded into main memory.
- **0x7FFF0000 to 0x7FFFFFFF:** A guard page inaccessible to the user. This page makes it easier for the operating system to check on out-of-bounds pointer references.
- **0x80000000 to 0xFFFFFFFF:** System address space. This 2-Gbyte process is used for the Windows Executive, Kernel, and device drivers.

On 64-bit platforms, 8TB of user address space is available in Windows Vista.

Windows Paging

When a process is created, it can in principle make use of the entire user space of almost 2 Gbytes. This space is divided into fixed-size pages, any of which can be

8.6 SUMMARY

To use the processor and the I/O facilities efficiently, it is desirable to maintain as many processes in main memory as possible. In addition, it is desirable to free programmers from size restrictions in program development.

The way to address both of these concerns is virtual memory. With virtual memory, all address references are logical references that are translated at run time to real addresses. This allows a process to be located anywhere in main memory and for that location to change over time. Virtual memory also allows a process to be broken up into pieces. These pieces need not be contiguously located in main memory during execution and, indeed, it is not even necessary for all of the pieces of the process to be in main memory during execution.

WINDOWS/LINUX COMPARISON—SCHEDULING	
Windows	**Linux**
O(1) scheduling, using per-CPU priority lists	Linux added O(1) scheduling, using per-CPU priority lists in version 2.6
Lower priority numbers represent lower priority	As with other flavors of UNIX, lower priority numbers represent higher priority
16 non-real-time priorities (0-15)	40 non-real-time priorities (100-139)
Highest priority runnable threads (almost) always scheduled on the available processors	Linux runs highest priority non-real-time processes unless they hit their quantum (i.e. are CPU bound), in which case lower-priority processes are allowed to run to the end of their quantum
Applications can specify CPU affinities, and subject to that constraint, scheduler picks an ideal processor which it always tries to use for better cache performance. But threads are moved to other idle CPUs, or CPUs running lower-priority threads	
Priority inversion is managed by a crude mechanism that gives a huge priority boost to threads that have been starved for seconds	Letting lower-priority processes runs ahead of high-priority threads that hit their quantum avoids starvation and thus fixes cases of priority inversion
Priorities for non-real-time threads dynamically adjusted to give better performance for foreground applications and I/O. Priorities boost from the base and then decay at quantum end	Periodically the scheduler rebalances the assignment of processes to CPUs by moving processes from the ready queues for busy CPUs to underutilized CPUs Rebalancing is based on system defined scheduling domains rather than process affinities (i.e. to correspond to NUMA nodes)
NUMA aware	NUMA aware
16 real-time priority levels (16-31) with priority over non-real-time threads	99 real-time priority levels (1-99) with priority over non-real-time processes
Real-time threads are scheduled round-robin	Real-time processes can be scheduled either round or FIFO, meaning they will not preempted except by a higher priority real-time process

10.5 WINDOWS SCHEDULING

Windows is designed to be as responsive as possible to the needs of a single user in a highly interactive environment or in the role of a server. Windows implements a preemptive scheduler with a flexible system of priority levels that includes round-robin scheduling within each level and, for some levels, dynamic priority variation on the basis of their current thread activity. Threads are the unit of scheduling in Windows rather than processes.

Process and Thread Priorities

Priorities in Windows are organized into two bands, or classes: real time and variable. Each of these bands consists of 16 priority levels. Threads requiring immediate attention are in the real-time class, which includes functions such as communications and real-time tasks.

Overall, because Windows makes use of a priority-driven preemptive scheduler, threads with real-time priorities have precedence over other threads. On a uniprocessor, when a thread becomes ready whose priority is higher than the currently executing thread, the lower-priority thread is preempted and the processor given to the higher-priority thread.

Priorities are handled somewhat differently in the two classes (Figure 10.15). In the real-time priority class, all threads have a fixed priority that never changes. All of the active threads at a given priority level are in a round-robin queue. In the

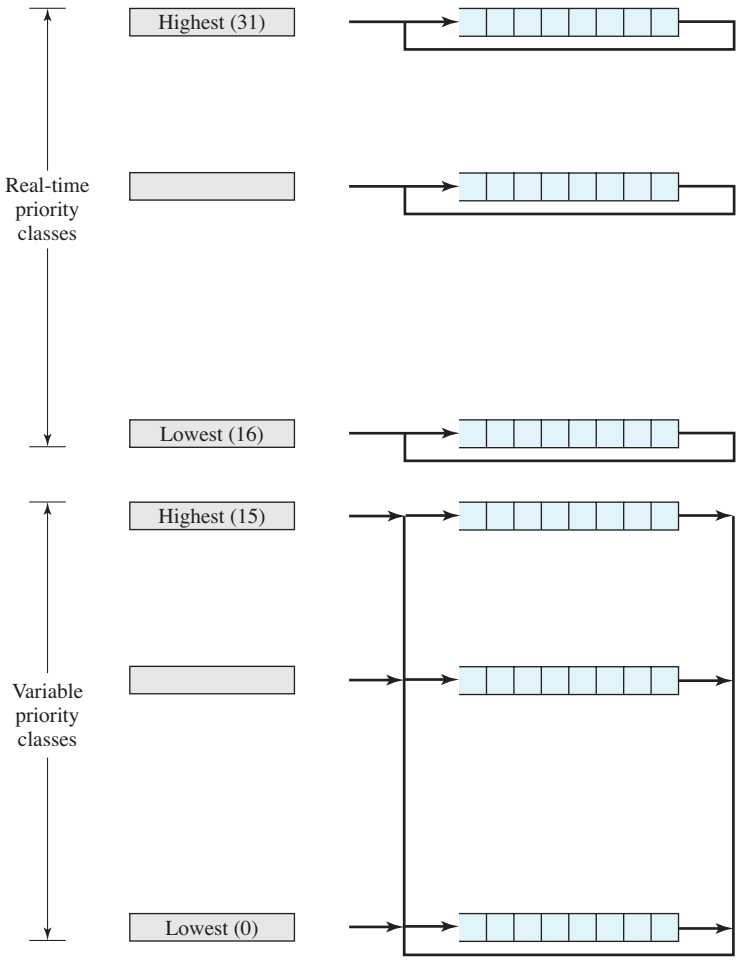


Figure 10.15 **Windows Thread Dispatching Priorities**

variable priority class, a thread's priority begins at some initial assigned value and then may be temporarily boosted (raised) during the thread's lifetime. There is a FIFO queue at each priority level; a thread will change queues among the variable priority classes as its own priority changes. However, a thread at priority level 15 or below is never boosted to level 16 or any other level in the real-time class.

The initial priority of a thread in the variable priority class is determined by two quantities: process base priority and thread base priority. The process base priority is an attribute of the process object, and can take on any value from 0 through 15. Each thread object associated with a process object has a thread base priority attribute that indicates the thread's base priority relative to that of the process. The thread's base priority can be equal to that of its process or within two levels above or below that of the process. So, for example, if a process has a base priority of 4 and one of its threads has a base priority of −1, then the initial priority of that thread is 3.

Once a thread in the variable priority class has been activated, its actual priority, referred to as the thread's current priority, may fluctuate within given boundaries. The current priority may never fall below the thread's base priority and it may never exceed 15. Figure 10.16 gives an example. The process object has a base priority attribute of 4. Each thread object associated with this process object must have an initial priority of between 2 and 6. Suppose the base priority for thread is 4. Then the current priority for that thread may fluctuate in the range from 4 through 15 depending on what boosts it has been given. If a thread is interrupted to wait on an I/O event, the Windows Kernel boosts its priority. If a boosted thread is interrupted because it has used up its current time quantum, the Kernel lowers its priority. Thus, processor-bound threads tend toward lower priorities and I/O-bound threads tend toward higher priorities. In the case of I/O-bound threads, the Kernel boosts the priority more for interactive waits (e.g., wait on keyboard or display) than for other

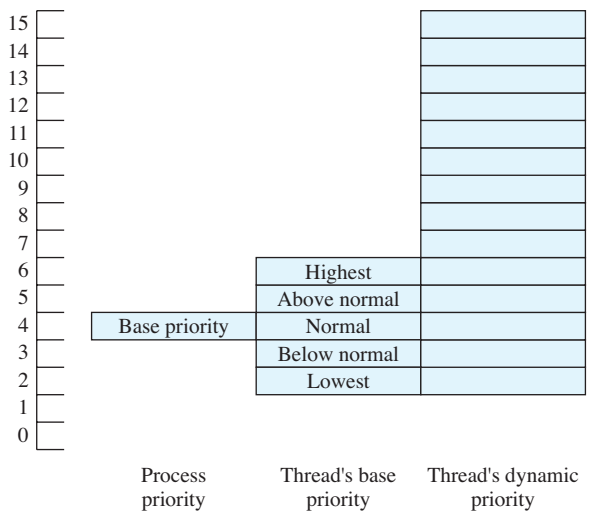


Figure 10.16 Example of Windows Priority Relationship

types of I/O (e.g., disk I/O). Thus, interactive threads tend to have the highest priorities within the variable priority class.

Multiprocessor Scheduling

When Windows is run on a single processor, the highest-priority thread is always active unless it is waiting on an event. If there is more than one thread that has the same highest priority, then the processor is shared, round robin, among all the threads at that priority level. In a multiprocessor system with *N* processors, the Kernel tries to give the *N* processors to the *N* highest priority threads that are ready to run. The remaining, lower-priority, threads must wait until the other threads block or have their priority decay. Lower-priority threads may also have their priority boosted to 15 for a very short time if they are being starved, solely to correct instances of priority inversion.

The foregoing scheduling discipline is affected by the processor affinity attribute of a thread. If a thread is ready to execute but the only available processors are not in its processor affinity set, then that thread is forced to wait, and the Kernel schedules the next available thread.

Figure 11.15 shows the key kernel mode components related to the Windows I/O manager. The I/O manager is responsible for all I/O for the operating system and provides a uniform interface that all types of drivers can call.

Basic I/O Facilities

The I/O manager works closely with four types of kernel components:

- **Cache manager:** The cache manager handles file caching for all file systems. It can dynamically increase and decrease the size of the cache devoted to a particular file as the amount of available physical memory varies. The system

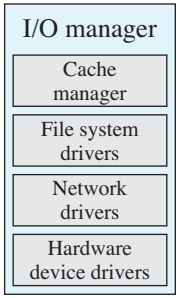


Figure 11.15 Windows I/O Manager

records updates in the cache only and not on disk. A kernel thread, the lazy writer, periodically batches the updates together to write to disk. Writing the updates in batches allows the I/O to be more efficient. The cache manager works by mapping regions of files into kernel virtual memory and then relying on the virtual memory manager to do most of the work to copy pages to and from the files on disk.

- **File system drivers:** The I/O manager treats a file system driver as just another device driver and routes I/O requests for file system volumes to the appropriate software driver for that volume. The file system, in turn, sends I/O requests to the software drivers that manage the hardware device adapter.
- **Network drivers:** Windows includes integrated networking capabilities and support for remote file systems. The facilities are implemented as software drivers rather than part of the Windows Executive.
- **Hardware device drivers:** These software drivers access the hardware registers of the peripheral devices using entry points in the kernel's Hardware Abstraction Layer. A set of these routines exists for every platform that Windows supports; because the routine names are the same for all platforms, the source code of Windows device drivers is portable across different processor types.

Asynchronous and Synchronous I/O

Windows offers two modes of I/O operation: asynchronous and synchronous. The asynchronous mode is used whenever possible to optimize application performance. With asynchronous I/O, an application initiates an I/O operation and then can continue processing while the I/O request is fulfilled. With synchronous I/O, the application is blocked until the I/O operation completes.

Asynchronous I/O is more efficient, from the point of view of the calling thread, because it allows the thread to continue execution while the I/O operation is queued by the I/O manager and subsequently performed. However, the application that invoked the asynchronous I/O operation needs some way to determine when the operation is complete. Windows provides five different techniques for signaling I/O completion:

- **Signaling the file object:** With this approach, the event associated with a file object is set when an operation on that object is complete. The thread that invoked the I/O operation can continue to execute until it reaches a point where it must stop until the I/O operation is complete. At that point, the thread can wait until the operation is complete and then continue. This technique is simple and easy to use but is not appropriate for handling multiple I/O requests. For example, if a thread needs to perform multiple simultaneous actions on a single file, such as reading from one portion and writing to another portion of the file, with this technique, the thread could not distinguish between the completion of the read and the completion of the write. It would simply know that some requested I/O operation on this file was complete.
- **Signaling an event object:** This technique allows multiple simultaneous I/O requests against a single device or file. The thread creates an event for each

request. Later, the thread can wait on a single one of these requests or on an entire collection of requests.

- **Asynchronous procedure call:** This technique makes use of a queue associated with a thread, known as the asynchronous procedure call (APC) queue. In this case, the thread makes I/O requests, specifying a user mode routine to call when the I/O completes. The I/O manager places the results of each request in the calling thread's APC queue. The next time the thread blocks in the kernel, the APCs will be delivered; each causing the thread to return to user mode and execute the specified routine.
- **I/O completion ports:** This technique is used on a Windows server to optimize the use of threads. The application creates a pool of threads for handling the completion of I/O requests. Each thread waits on the completion port, and the Kernel wakes threads to handle each I/O completion. One of the advantages of this approach is that the application can specify a limit for how many of these threads will run at a time.
- **Polling:** Asynchronous I/O requests write a status and transfer count into the process' user virtual memory when the operation completes. A thread can just check these values to see if the operation has completed.

Software RAID

Windows supports two sorts of RAID configurations, defined in [MS96] as follows:

- **Hardware RAID:** Separate physical disks combined into one or more logical disks by the disk controller or disk storage cabinet hardware.
- **Software RAID:** Noncontiguous disk space combined into one or more logical partitions by the fault-tolerant software disk driver, FTDISK.

In hardware RAID, the controller interface handles the creation and regeneration of redundant information. The software RAID, available on Windows Server, implements the RAID functionality as part of the operating system and can be used with any set of multiple disks. The software RAID facility implements RAID 1 and RAID 5. In the case of RAID 1 (disk mirroring), the two disks containing the primary and mirrored partitions may be on the same disk controller or different disk controllers. The latter configuration is referred to as *disk duplexing*.

Volume Shadow Copies

Shadow copies are an efficient way of making consistent snapshots of volumes to that they can be backed up. They are also useful for archiving files on a per-volume basis. If a user deletes a file, he or she can retrieve an earlier copy from any available shadow copy made by the system administrator. Shadow copies are implemented by a software driver that makes copies of data on the volume before it is overwritten.

Volume Encryption

Starting with Windows Vista, the operating system supports the encryption of entire volumes. This is more secure than encrypting individual files, as the entire system

works to be sure that the data is safe. Up to three different methods of supplying the cryptographic key can be provided; allowing multiple interlocking layers of security.

12.10 WINDOWS FILE SYSTEM

The developers of Windows designed a new file system, the New Technology File System (NTFS), that is intended to meet high-end requirements for workstations and servers. Examples of high-end applications include the following:

- Client/server applications such as file servers, compute servers, and database servers
- Resource-intensive engineering and scientific applications
- Network applications for large corporate systems

This section provides an overview of NTFS.

Key Features of NTFS

NTFS is a flexible and powerful file system built, as we shall see, on an elegantly simple file system model. The most noteworthy features of NTFS include the following:

- **Recoverability:** High on the list of requirements for the new Windows file system was the ability to recover from system crashes and disk failures. In the

WINDOWS/LINUX COMPARISON	
Windows	**Linux**
Windows supports a variety of file systems, including the legacy FAT/FAT32 file systems from DOS/Windows and formats common to CDs and DVDs	Linux supports a variety of file systems, including Microsoft file systems, for compatibility and interoperation
The most common file system used in Windows is NTFS, which has many advanced features related to security, encryption, compression, journaling, change notifications, and indexing built in	The most common file systems are Ext2, Ext3, and IBM's JFS journaling file system
NTFS uses logging of metadata to avoid having to perform file system checks after crashes	With Ext3, journaling of changes allows file system checks to be avoided after crashes
Windows file systems are implemented as device drivers, and can be stacked in layers, as with other device drivers, due to the object-oriented implementation of Windows I/O. Typically NTFS is sandwiched between 3rd party filter drivers, which implement functions like anti-virus, and the volume management drivers, which implement RAID	Linux file systems are implemented using the Virtual File System (VFS) technique developed by Sun Microsystems. File systems are plug-ins in the VFS model, which is similar to the general object-oriented model used for block and character devices
The file systems depend heavily on the I/O system and the CACHE manager. The cache manager is a virtual file cache that maps regions of files into kernel virtual-address space	Linux uses a page cache which keeps copies of recently used pages in memory. Pages are organized per 'owner': most commonly an inode for files and directories, or the inode of the block device for file system metadata
The CACHE manager is implemented on top of the virtual memory system, providing a unified caching mechanism for both pages and file blocks	The Linux virtual memory system builds memory-mapping of files on top of the page cache facility
Directories, bitmaps, file and file system metadata, are all represented as files by NTFS, and thus all rely on unified caching by the CACHE manager	The Common File System model of VFS treats directory entries and file inodes and other file system metadata, such as the superblock, separately from file data with special caching for each category. File data can be stored in the cache twice, once for the 'file' owner and once for the 'block device' owner
Pre-fetching of disk data uses sophisticated algorithms which remember past access patterns of code and data for applications, and initiate asynchronous pagefault operations when applications launch, move to the foreground, or resume from the system hibernate power-off state	Pre-fetching of disk data uses read-ahead of files that are being accessed sequentially

event of such failures, NTFS is able to reconstruct disk volumes and return them to a consistent state. It does this by using a transaction processing model for changes to the file system; each significant change is treated as an atomic action that is either entirely performed or not performed at all. Each transaction that was in process at the time of a failure is subsequently backed out or

brought to completion. In addition, NTFS uses redundant storage for critical file system data, so that failure of a disk sector does not cause the loss of data describing the structure and status of the file system.

- **Security:** NTFS uses the Windows object model to enforce security. An open file is implemented as a file object with a security descriptor that defines its security attributes. The security descriptor is persisted as an attribute of each file on disk.
- **Large disks and large files:** NTFS supports very large disks and very large files more efficiently than most other file systems, including FAT.
- **Multiple data streams:** The actual contents of a file are treated as a stream of bytes. In NTFS it is possible to define multiple data streams for a single file. An example of the utility of this feature is that it allows Windows to be used by remote Macintosh systems to store and retrieve files. On Macintosh, each file has two components: the file data and a resource fork that contains information about the file. NTFS treats these two components as two data streams.
- **Journaling:** NTFS keeps a log of all changes made to files on the volumes. Programs, such as desktop search, can read the journal to identify what files have changed.
- **Compression and Encryption:** Entire directories and individual files can be transparently compressed and/or encrypted.

NTFS Volume and File Structure

NTFS makes use of the following disk storage concepts:

- **Sector:** The smallest physical storage unit on the disk. The data size in bytes is a power of 2 and is almost always 512 bytes.
- **Cluster:** One or more contiguous (next to each other on the same track) sectors. The cluster size in sectors is a power of 2.
- **Volume:** A logical partition on a disk, consisting of one or more clusters and used by a file system to allocate space. At any time, a volume consists of a file system information, a collection of files, and any additional unallocated space remaining on the volume that can be allocated to files. A volume can be all or a portion of a single disk or it can extend across multiple disks. If hardware or software RAID 5 is employed, a volume consists of stripes spanning multiple disks. The maximum volume size for NTFS is 2^{64} bytes.

The cluster is the fundamental unit of allocation in NTFS, which does not recognize sectors. For example, suppose each sector is 512 bytes and the system is configured with two sectors per cluster (one cluster = 1K bytes). If a user creates a file of 1600 bytes, two clusters are allocated to the file. Later, if the user updates the file to 3200 bytes, another two clusters are allocated. The clusters allocated to a file need not be contiguous; it is permissible to fragment a file on the disk. Currently, the maximum file size supported by NTFS is 2^{32} clusters, which is equivalent to a maximum of 2^{48} bytes. A cluster can have at most 2^{16} bytes.

Table 12.5 Windows NTFS Partition and Cluster Sizes

Volume Size	Sectors per Cluster	Cluster Size
512 Mbyte	1	512 bytes
512 Mbyte–1 Gbyte	2	1K
1 Gbyte–2 Gbyte	4	2K
2 Gbyte–4 Gbyte	8	4K
4 Gbyte–8 Gbyte	16	8K
8 Gbyte–16 Gbyte	32	16K
16 Gbyte–32 Gbyte	64	32K
> 32 Gbyte	128	64K

The use of clusters for allocation makes NTFS independent of physical sector size. This enables NTFS to support easily nonstandard disks that do not have a 512-byte sector size and to support efficiently very large disks and very large files by using a larger cluster size. The efficiency comes from the fact that the file system must keep track of each cluster allocated to each file; with larger clusters, there are fewer items to manage.

Table 12.5 shows the default cluster sizes for NTFS. The defaults depend on the size of the volume. The cluster size that is used for a particular volume is established by NTFS when the user requests that a volume be formatted.

NTFS Volume Layout NTFS uses a remarkably simple but powerful approach to organizing information on a disk volume. Every element on a volume is a file, and every file consists of a collection of attributes. Even the data contents of a file is treated as an attribute. With this simple structure, a few general-purpose functions suffice to organize and manage a file system.

Figure 12.19 shows the layout of an NTFS volume, which consists of four regions. The first few sectors on any volume are occupied by the **partition boot sector** (although it is called a sector, it can be up to 16 sectors long), which contains information about the volume layout and the file system structures as well as boot startup information and code. This is followed by the **master file table (MFT)**, which contains information about all of the files and folders (directories) on this NTFS volume. In essence, the MFT is a list of all files and their attributes on this NTFS volume, organized as a set of rows in a relational database structure.

Following the MFT is a region, typically about 1 Mbyte in length, containing **system files**. Among the files in this region are the following:

- **MFT2:** A mirror of the first three rows of the MFT, used to guarantee access to the MFT in the case of a single-sector failure

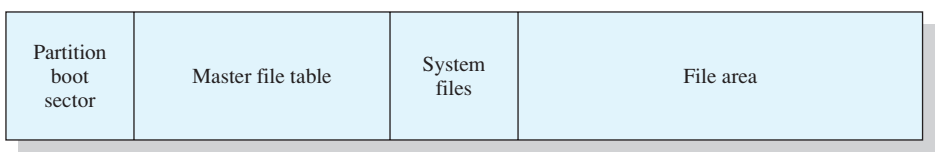


Figure 12.19 **NTFS Volume Layout**

- **Log file:** A list of transaction steps used for NTFS recoverability
- **Cluster bit map:** A representation of the volume, showing which clusters are in use
- **Attribute definition table:** Defines the attribute types supported on this volume and indicates whether they can be indexed and whether they can be recovered during a system recovery operation

Master File Table The heart of the Windows file system is the MFT. The MFT is organized as a table of 1024-byte rows, called records. Each row describes a file on this volume, including the MFT itself, which is treated as a file. If the contents of a file are small enough, then the entire file is located in a row of the MFT. Otherwise, the row for that file contains partial information and the remainder of the file spills over into other available clusters on the volume, with pointers to those clusters in the MFT row of that file.

Each record in the MFT consists of a set of attributes that serve to define the file (or folder) characteristics and the file contents. Table 12.6 lists the attributes that may be found in a row, with the required attributes indicated by shading.

Recoverability

NTFS makes it possible to recover the file system to a consistent state following a system crash or disk failure. The key elements that support recoverability are as follows (Figure 12.20):

- **I/O manager:** Includes the NTFS driver, which handles the basic open, close, read, write functions of NTFS. In addition, the software RAID module FTDISK can be configured for use.

Table 12.6 Windows NTFS File and Directory Attribute Types

Attribute Type	Description
Standard information	Includes access attributes (read-only, read/write, etc.); time stamps, including when the file was created or last modified; and how many directories point to the file (link count).
Attribute list	A list of attributes that make up the file and the file reference of the MFT file record in which each attribute is located. Used when all attributes do not fit into a single MFT file record.
File name	A file or directory must have one or more names.
Security descriptor	Specifies who owns the file and who can access it.
Data	The contents of the file. A file has one default unnamed data attribute and may have one or more named data attributes.
Index root	Used to implement folders.
Index allocation	Used to implement folders.
Volume information	Includes volume-related information, such as the version and name of the volume.
Bitmap	Provides a map representing records in use on the MFT or folder.

Note: Colored rows refer to required file attributes; the other attributes are optional.

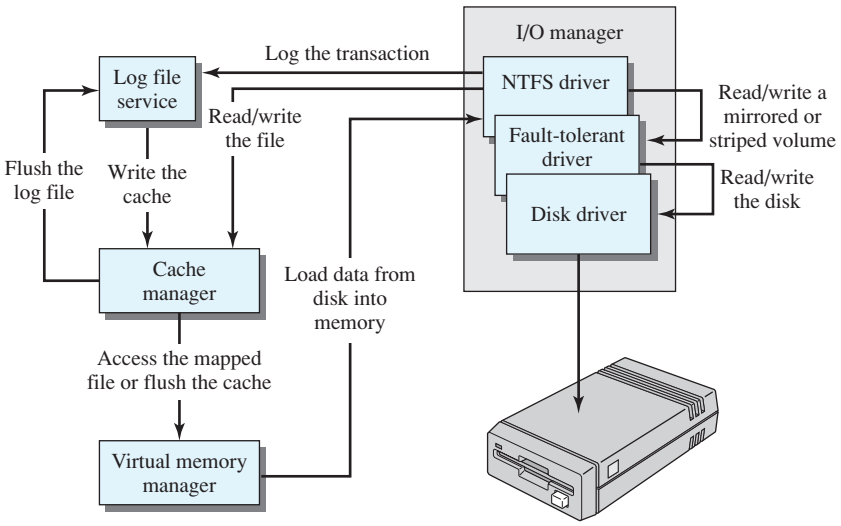


Figure 12.20 Windows NTFS Components

- **Log file service:** Maintains a log of file system metadata changes on disk. The log file is used to recover an NTFS-formatted volume in the case of a system failure (i.e., without having to run the file system check utility).
- **Cache manager:** Responsible for caching file reads and writes to enhance performance. The cache manager optimizes disk I/O.
- **Virtual memory manager:** The NTFS accesses cached files by mapping file references to virtual memory references and reading and writing virtual memory.

It is important to note that the recovery procedures used by NTFS are designed to recover file system metadata, not file contents. Thus, the user should never lose a volume or the directory/file structure of an application because of a crash. However, user data are not guaranteed by the file system. Providing full recoverability, including user data, would make for a much more elaborate and resource-consuming recovery facility.

The essence of the NTFS recovery capability is logging. Each operation that alters a file system is treated as a transaction. Each suboperation of a transaction that alters important file system data structures is recorded in a log file before being recorded on the disk volume. Using the log, a partially completed transaction at the time of a crash can later be redone or undone when the system recovers.

In general terms, these are the steps taken to ensure recoverability, as described in [RUSS05]:

1. NTFS first calls the log file system to record in the log file in the cache any transactions that will modify the volume structure.
2. NTFS modifies the volume (in the cache).

3. The cache manager calls the log file system to prompt it to flush the log file to disk.
4. Once the log file updates are safely on disk, the cache manager flushes the volume changes to disk.

15.6 WINDOWS VISTA SECURITY

A good example of the access control concepts we have been discussing is the Windows access control facility, which exploits object-oriented concepts to provide a powerful and flexible access control capability.

Windows provides a uniform access control facility that applies to processes, threads, files, semaphores, windows, and other objects. Access control is governed by two entities: an access token associated with each process and a security descriptor associated with each object for which interprocess access is possible.

Access Control Scheme

When a user logs on to an Windows system, Windows uses a name/password scheme to authenticate the user. If the logon is accepted, a process is created for the user and an access token is associated with that process object. The access token, whose details are described later, include a security ID (SID), which is the identifier by which this user is known to the system for purposes of security. If the initial user process spawns a new process, the new process object inherits the same access token.

The access token serves two purposes:

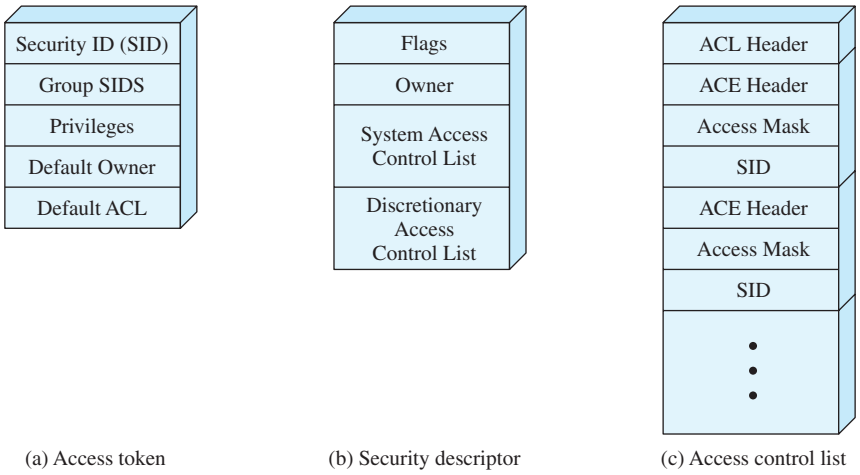
1. It keeps all necessary security information together to speed access validation. When any process associated with a user attempts access, the security subsystem can make use of the token associated with that process to determine the user's access privileges.
2. It allows each process to modify its security characteristics in limited ways without affecting other processes running on behalf of the user.

The chief significance of the second point has to do with privileges that may be associated with a user. The access token indicates which privileges a user may have. Generally, the token is initialized with each of these privileges in a disabled state. Subsequently, if one of the user's processes needs to perform a privileged operation, the process may enable the appropriate privilege and attempt access. It would be undesirable to share the same token among all a user's processes, because in that case enabling a privilege for one process enables it for all of them.

Associated with each object for which interprocess access is possible is a security descriptor. The chief component of the security descriptor is an access control list that specifies access rights for various users and user groups for this object. When a process attempts to access this object , the SID of the process is matched against the access control list of the object to determine if access will be allowed or denied.

When an application opens a reference to a securable object, Windows verifies that the object's security descriptor grants the application's user access. If the check succeeds, Windows caches the resulting granted access rights.

An important aspect of Windows security is the concept of impersonation, which simplifies the use of security in a client/server environment. If client and server talk through a RPC connection, the server can temporarily assume the



(a) Access token (b) Security descriptor (c) Access control list

Figure 15.11 Windows Security Structures

identity of the client so that it can evaluate a request for access relative to that client's rights. After the access, the server reverts to its own identity.

Access Token

Figure 15.11a shows the general structure of an access token, which includes the following parameters:

- **Security ID:** Identifies a user uniquely across all of the machines on the network. This generally corresponds to a user's logon name.
- **Group SIDs:** A list of the groups to which this user belongs. A group is simply a set of user IDs that are identified as a group for purposes of access control. Each group has a unique group SID. Access to an object can be defined on the basis of group SIDs, individual SIDs, or a combination. There is also a SID which reflects the process integrity level (low, medium, high, or system).
- **Privileges:** A list of security-sensitive system services that this user may call. An example is create token. Another example is the set backup privilege; users with this privilege are allowed to use a backup tool to back up files that they normally would not be able to read.
- **Default Owner:** If this process creates another object, this field specifies who is the owner of the new object. Generally, the owner of the new process is the same as the owner of the spawning process. However, a user may specify that the default owner of any processes spawned by this process is a group SID to which this user belongs.
- **Default ACL:** This is an initial list of protections applied to the objects that the user creates. The user may subsequently alter the ACL for any object that it owns or that one of its groups owns.

Security Descriptors

Figure 15.11b shows the general structure of a security descriptor, which includes the following parameters:

- **Flags:** Defines the type and contents of a security descriptor. The flags indicate whether or not the SACL and DACL are present, whether or not they were placed on the object by a defaulting mechanism, and whether the pointers in the descriptor use absolute or relative addressing. Relative descriptors are required for objects that are transmitted over a network, such as information transmitted in a RPC.
- **Owner:** The owner of the object can generally perform any action on the security descriptor. The owner can be an individual or a group SID. The owner has the authority to change the contents of the DACL.
- **System Access Control List (SACL):** Specifies what kinds of operations on the object should generate audit messages. An application must have the corresponding privilege in its access token to read or write the SACL of any object. This is to prevent unauthorized applications from reading SACLs (thereby learning what not to do to avoid generating audits) or writing them (to generate many audits to cause an illicit operation to go unnoticed). The SACL also specifies the object integrity level. Processes cannot modify an object unless the process integrity level meets or exceeds the level on the object.
- **Discretionary Access Control List (DACL):** Determines which users and groups can access this object for which operations. It consists of a list of access control entries (ACEs).

When an object is created, the creating process can assign as owner its own SID or any group SID in its access token. The creating process cannot assign an owner that is not in the current access token. Subsequently, any process that has been granted the right to change the owner of an object may do so, but again with the same restriction. The reason for the restriction is to prevent a user from covering his tracks after attempting some unauthorized action.

Let us look in more detail at the structure of access control lists, because these are at the heart of the Windows access control facility (Figure 15.11c). Each list consists of an overall header and a variable number of access control entries. Each entry specifies an individual or group SID and an access mask that defines the rights to be granted to this SID. When a process attempts to access an object, the object manager in the Windows Executive reads the SID and group SIDs from the access token and including the integrity level SID. If the access requested includes modifying the object, the integrity level is checked against the object integrity level in the SACL. If that test passes, the object manager then scans down the object's DACL. If a match is found (that is, if an ACE is found with a SID that matches one of the SIDs from the access token), then the process can have the access rights specified by the access mask in that ACE. This also may including denying access, in which case the access request fails.

Figure 15.12 shows the contents of the access mask. The least significant 16 bits specify access rights that apply to a particular type of object. For example, bit 0 for a file object is `File_Read_Data` access and bit 0 for an event object is `Event_Query_Status` access.

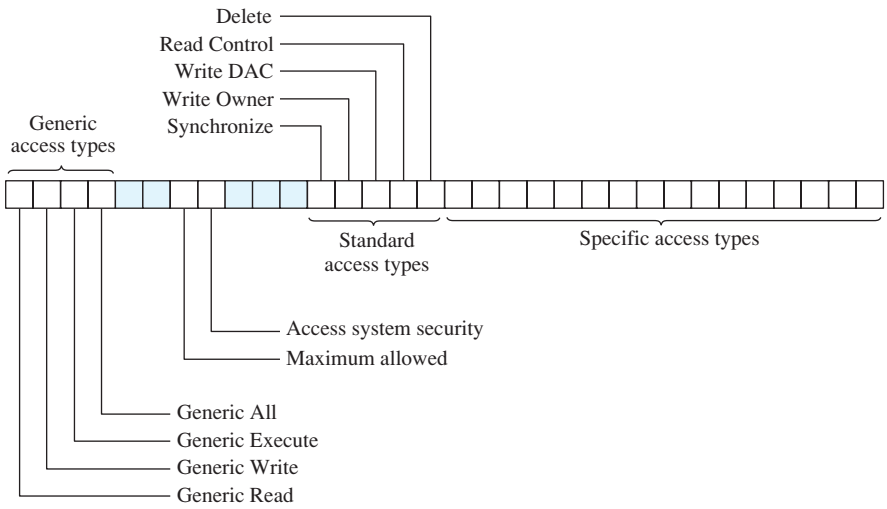


Figure 15.12 **Access Mask**

The most significant 16 bits of the mask contains bits that apply to all types of objects. Five of these are referred to as standard access types:

- **Synchronize:** Gives permission to synchronize execution with some event associated with this object. In particular, this object can be used in a wait function.
- **Write_owner:** Allows a program to modify the owner of the object. This is useful because the owner of an object can always change the protection on the object (the owner may not be denied Write DAC access).
- **Write_DAC:** Allows the application to modify the DACL and hence the protection on this object.
- **Read_control:** Allows the application to query the owner and DACL fields of the security descriptor of this object.
- **Delete:** Allows the application to delete this object.

The high-order half of the access mask also contains the four generic access types. These bits provide a convenient way to set specific access types in a number of different object types. For example, suppose an application wishes to create several types of objects and ensure that users have read access to the objects, even though read has a somewhat different meaning for each object type. To protect each object of each type without the generic access bits, the application would have to construct a different ACE for each type of object and be careful to pass the correct ACE when creating each object. It is more convenient to create a single ACE that expresses the generic concept allow read, simply apply this ACE to each object that is created, and have the right thing happen. That is the purpose of the generic access bits, which are

- Generic_all: Allow all access
- Generic_execute: Allow execution if executable
- Generic_write: Allow write access
- Generic_read: Allow read only access

The generic bits also affect the standard access types. For example, for a file object, the Generic_Read bit maps to the standard bits Read_Control and Synchronize and to the object-specific bits File_Read_Data, File_Read_Attributes, and File_Read_EA. Placing an ACE on a file object that grants some SID Generic_Read grants those five access rights as if they had been specified individually in the access mask.

The remaining two bits in the access mask have special meanings. The Access_System_Security bit allows modifying audit and alarm control for this object. However, not only must this bit be set in the ACE for a SID, but the access token for the process with that SID must have the corresponding privilege enabled.

Finally, the Maximum_Allowed bit is not really an access bit, but a bit that modifies Windows 's algorithm for scanning the DACL for this SID. Normally, Windows will scan through the DACL until it reaches an ACE that specifically grants (bit set) or denies (bit not set) the access requested by the requesting process or until it reaches the end of the DACL, in which latter case access is denied. The Maximum_Allowed bit allows the object's owner to define a set of access rights that is the maximum that will be allowed to a given user. With this in mind, suppose that an application does not know all of the operations that it is going to be asked to perform on an object during a session. There are three options for requesting access:

1. Attempt to open the object for all possible accesses. The disadvantage of this approach is that the access may be denied even though the application may have all of the access rights actually required for this session.
2. Only open the object when a specific access is requested, and open a new handle to the object for each different type of request. This is generally the preferred method because it will not unnecessarily deny access, nor will it allow more access than necessary. However, it imposes additional overhead.
3. Attempt to open the object for as much access as the object will allow this SID. The advantage is that the user will not be artificially denied access, but the application may have more access than it needs. This latter situation may mask bugs in the application.

An important feature of Windows security is that applications can make use of the Windows security framework for user-defined objects. For example, a database server might create it own security descriptors and attach them to portions of a database. In addition to normal read/write access constraints, the server could secure database-specific operations, such as scrolling within a result set or performing a join. It would be the server's responsibility to define the meaning of special rights and perform access checks. But the checks would occur in a standard context, using systemwide user/group accounts and audit logs. The extensible security model should prove useful to implementers of foreign files systems.