THE WINDOWS OPERATING SYSTEM

William Stallings

This document is an extract from
Operating Systems: Internals and Design Principles, Fifth Edition
Prentice Hall, 2005, ISBN 0-13-147954-7

TABLE OF CONTENTS

2.5 MICROSOFT WINDOWS OVERVIEW

In this section, we provide an overview of Microsoft Windows.

History

The story of Windows begins with a very different operating system, 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 operating system 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 operating system simply used the 80286 as a "fast 8086." The operating system 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 operating system that could be swapped. DOS 3.3, released in 1987, provided support for the new line of IBM machines, 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 simply 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 operating system 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 operating system, 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 operating system 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 operating system (the follow-on to Windows 3.0). However, NT 3.1 was a new 32-bit operating system 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. 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 access to low-level system services, which could impact the reliability of the operating system.

In 2000, Microsoft introduced the next major upgrade, now called 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.

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, the latest desktop version of Windows was released, known as Windows XP. Both home PC and business workstation versions of XP are offered. Also in 2001, a 64-bit version of XP was introduced. In 2003, Microsoft introduced a new server version, known as Windows Server 2003; both 32-bit and 64 bit versions are available. The 64-bit versions of XP and Server 2003 are designed specifically for the 64-bit Intel Itanium hardware.

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 OS/2 and MacOS). Windows was driven by a need to exploit the processing capabilities of today's 32-bit microprocessors, which rival mainframes and minicomputers 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 operating system needs to support sophisticated real-time communication hardware and the associated communications protocols and data transfer architectures while at the same time supporting ongoing user interaction.

The foregoing remarks apply to the Professional version of Windows. The Server version is also multitasking but may support multiple users. It supports 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.

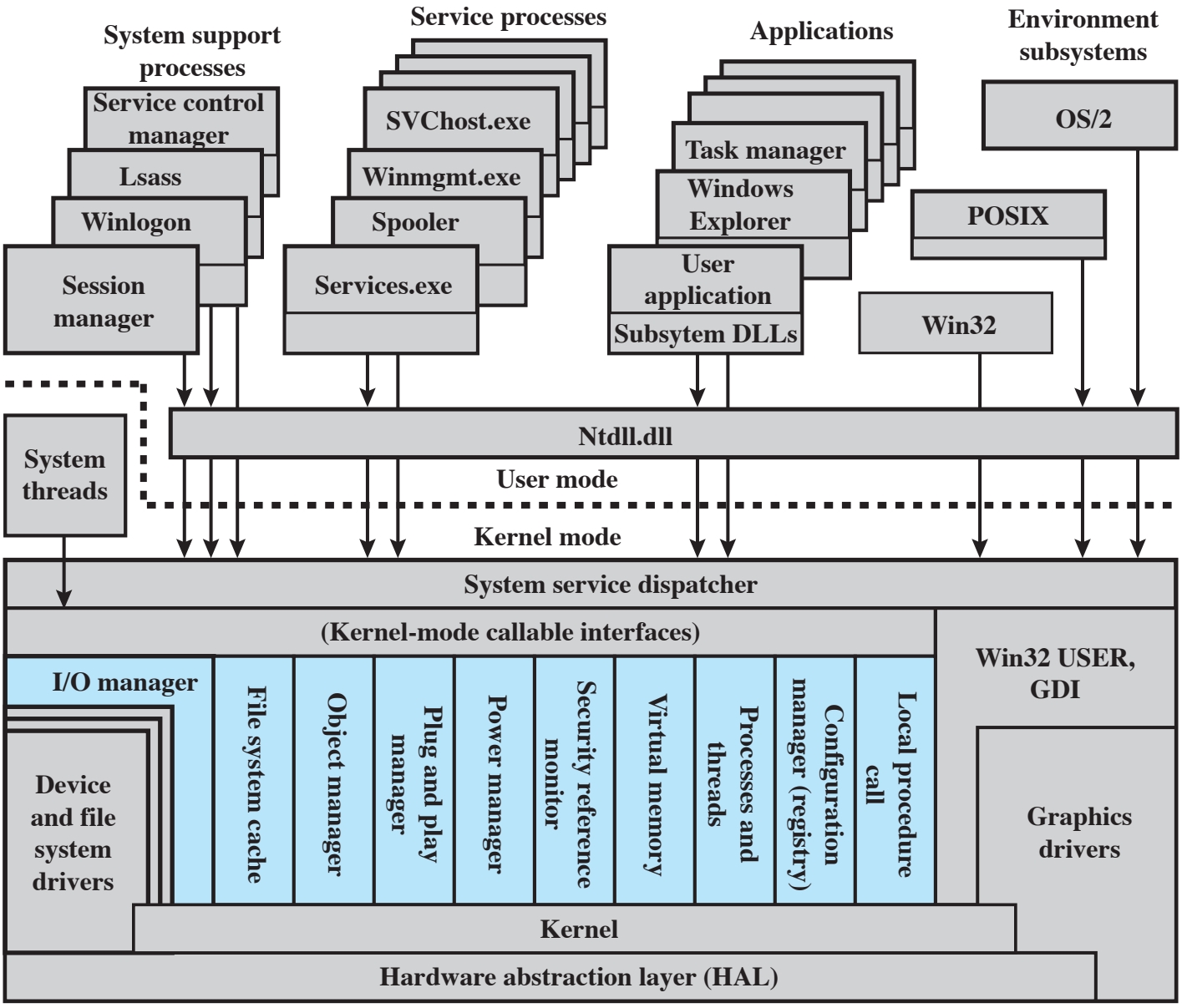
Architecture

Figure 2.13 illustrates the overall structure of Windows 2000; later releases of Windows have essentially the same structure at this level of detail. Its modular structure gives Windows considerable flexibility. It is designed to execute on a variety of hardware platforms and supports applications written for a variety of other operating systems. As of this writing, Windows is only implemented on the Intel Pentium/x86 and Itanium hardware platforms.

As with virtually all operating systems, Windows separates application-oriented software from operating system 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 does not have a pure microkernel architecture but what Microsoft refers to as a modified microkernel architecture. As with a pure microkernel architecture, Windows is highly modular. Each system function is managed by just one component of the operating system. The



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 2000 Architecture [SOLO00]

rest of the operating system and all applications access that function through the responsible component using a standard interface. 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). However, unlike a pure microkernel system, Windows is configured so that many of the system functions outside the microkernel run in kernel mode. The reason is performance. The Windows developers found that using the pure microkernel approach, many non-microkernel functions required several process or thread switches, mode switches, and the use of extra memory buffers.

The kernel-mode components of Windows are the following:

- **Executive:** Contains the base operating system services, such as memory management, process and thread management, security, I/O, and interprocess communication.
- **Kernel:** Consists of the most used and most fundamental components of the operating system. 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. Hence, it is the only part of the operating system that is not preemptible or pageable.
- **Hardware abstraction layer (HAL):** Maps between generic hardware commands and responses and those unique to a specific platform. It isolates the operating system from platform-specific hardware differences. The HAL makes each machine's system bus, direct memory access (DMA) controller, interrupt controller, system timers, and memory module look the same to the kernel. It also delivers the support needed for symmetric multiprocessing (SMP), explained subsequently.
- **Device drivers:** Include both file system and hardware device drivers that translate user I/O function calls into specific hardware device I/O requests.
- **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 modules 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 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 disk 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 consumption by putting the processor to sleep.
- **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:** Maps 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 and deletes objects and tracks 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 systemwide and per-user settings of various parameters.
- **Local procedure call (LPC) Facility:** Enforces a client/server relationship between applications and executive subsystems within a single system, in a manner similar to a 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 support processes:** Include services not provided as part of the Windows operating system, such as the logon process and the session manager.
- **Service processes:** Other Windows services such as the event logger.
- **Environment subsystems:** Expose the native Windows services to user applications and thus provide an operating system environment or personality. The supported subsystems are Win32, Posix, and OS/2. Each environment subsystem includes dynamic link libraries (DLLs) that convert the user application calls to Windows calls.
- **User applications:** Can be one of five types: Win32, Posix, OS/2, Windows 3.1, or MS-DOS.

Windows is structured to support applications written for Windows 2000 and later releases, Windows 98, and several other operating systems. Windows provides this support using a single, compact Executive through protected environment subsystems. The protected subsystems are those parts of Windows that interact with the end user. Each subsystem is a separate process, and the Executive protects its address space from that of other subsystems and applications. A protected subsystem provides a graphical or command-line user interface that defines the look and feel of the operating system 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 operating system interface that they see is the same as that for which they were written. So, for example, OS/2-based applications can run under the Windows operating system without modification. Furthermore, because the Windows system is itself designed to be platform independent, through the use of the hardware abstraction layer (HAL), it should be relatively easy to port both the protected subsystems and the applications they support from one hardware platform to another. In many cases, a recompile is all that should be required.

The most important subsystem is Win32. Win32 is the API implemented on both Windows 2000 and later releases and Windows 98. Some of the features of Win32 are not available in Windows 98, but those features implemented on Windows 98 are identical with those of Windows 2000 and later releases.

Client/Server Model

The Executive, 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.

Each environment subsystem and executive service subsystem is implemented as one or more processes. Each process waits for a request from a client for one of its services (for

example, memory services, process creation services, or processor scheduling services). A client, which can be an application program or another operating system module, 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 without any conflicts or duplications in the Executive. New APIs can be added easily.
- It improves reliability. Each executive services module runs on a separate process, with its own partition of memory, protected from other modules. Furthermore, the clients cannot directly access hardware or modify memory in which the Executive is stored. A single server can fail without crashing or corrupting the rest of the operating system.
- It provides a uniform means for applications to communicate with the Executive via LPCs without restricting flexibility. The message-passing process is hidden from the client applications by function stubs, which are nonexecutable placeholders kept in dynamic link libraries (DLLs). When an application makes an API call to an environment subsystem, 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. [CUST93] lists the following features of Windows that support threads and SMP:

- Operating system 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 operating system 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:** This is not supported at the user level but is supported to some extent within the Executive. For example, Directory objects are examples of container objects. One property of a container object is that the objects they contain can inherit properties from the container itself. As an example, suppose you have a directory in the file system that has its compressed flag set. Then any files you might create within that directory container will also have their compressed flag set.
- **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 accessible only by the kernel. 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). These kernel object data structures are accessible only by the kernel; it is impossible for an application to locate 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 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.

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. 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 have 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.

As an example of the objects managed by Windows, we list the two categories of objects managed by the kernel;

- **Control objects:** Used to control the operation of the kernel in areas not affecting dispatching and synchronization. Table 2.5 lists the kernel control objects.

- **Dispatcher objects:** Control the dispatching and synchronization of system operations. These are described in Chapter 6.

Windows is not a full-blown object-oriented operating system. 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 operating system design.

Table 2.5 Windows Microkernel Control Objects [MS96]

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.
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.
Profile	Used to measure the distribution of run time within a block of code. Both user and system code can be profiled.

4.4 WINDOWS THREAD AND SMP MANAGEMENT

Windows process design is driven by the need to provide support for a variety of operating system environments. Processes supported by different operating system 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 operating system 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 [SOLO00], illustrates the way in which a process relates to the resources it controls or uses. Each process is assigned a security access token, called the primary token of the process. When a user first logs on, Windows creates an access token that includes a 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

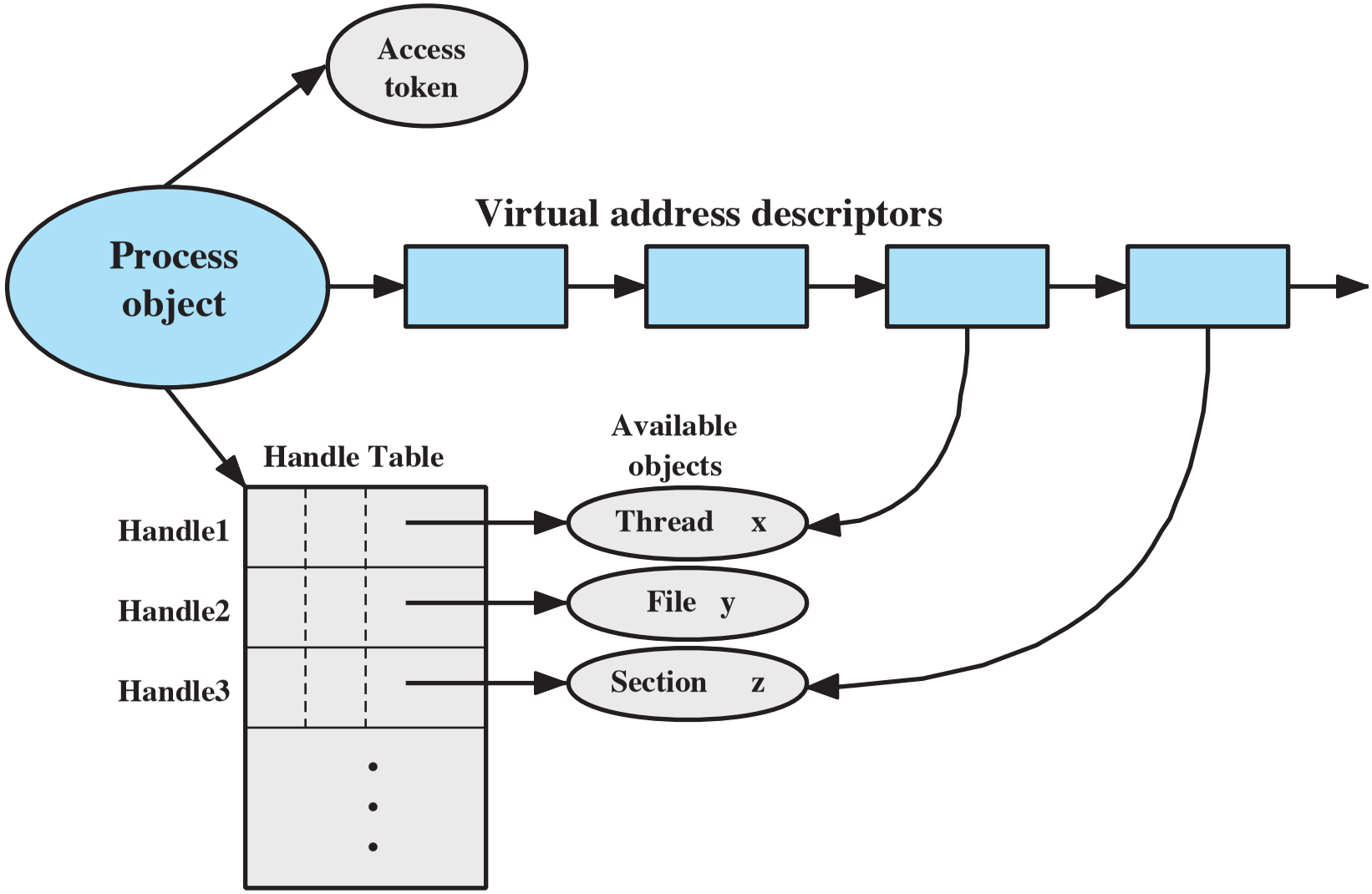


Figure 4.12 A Windows Process and Its Resources

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 upon receipt of the appropriate message; the only way to invoke such a service is by means of messages to a process object that provides that service. 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

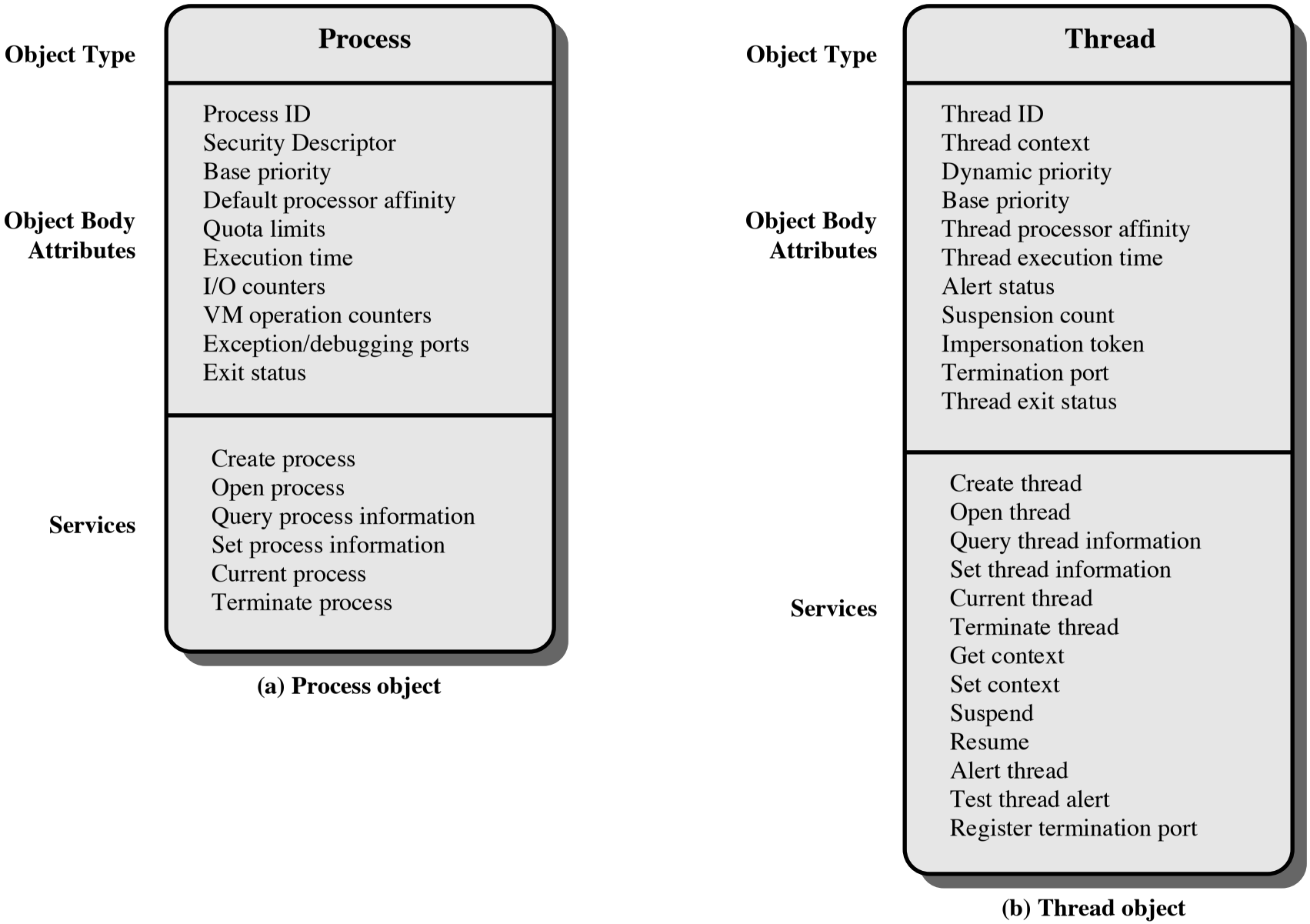


Figure 4.13 Windows Process and Thread Objects

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.

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.
Exit status	The reason for a process's termination.

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 the thread should 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.

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. Each client request triggers the creation of a new thread within the server.

Thread States

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

- **Ready:** May be scheduled for execution. The microkernel dispatcher keeps track of all ready threads and schedules them in priority order.
- **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.
- **Running:** Once the microkernel performs a thread or process switch, the standby thread enters the running state and begins execution and continues execution until it is preempted, exhausts its time slice, blocks, or terminates. In the first two cases, it goes back to the ready state.

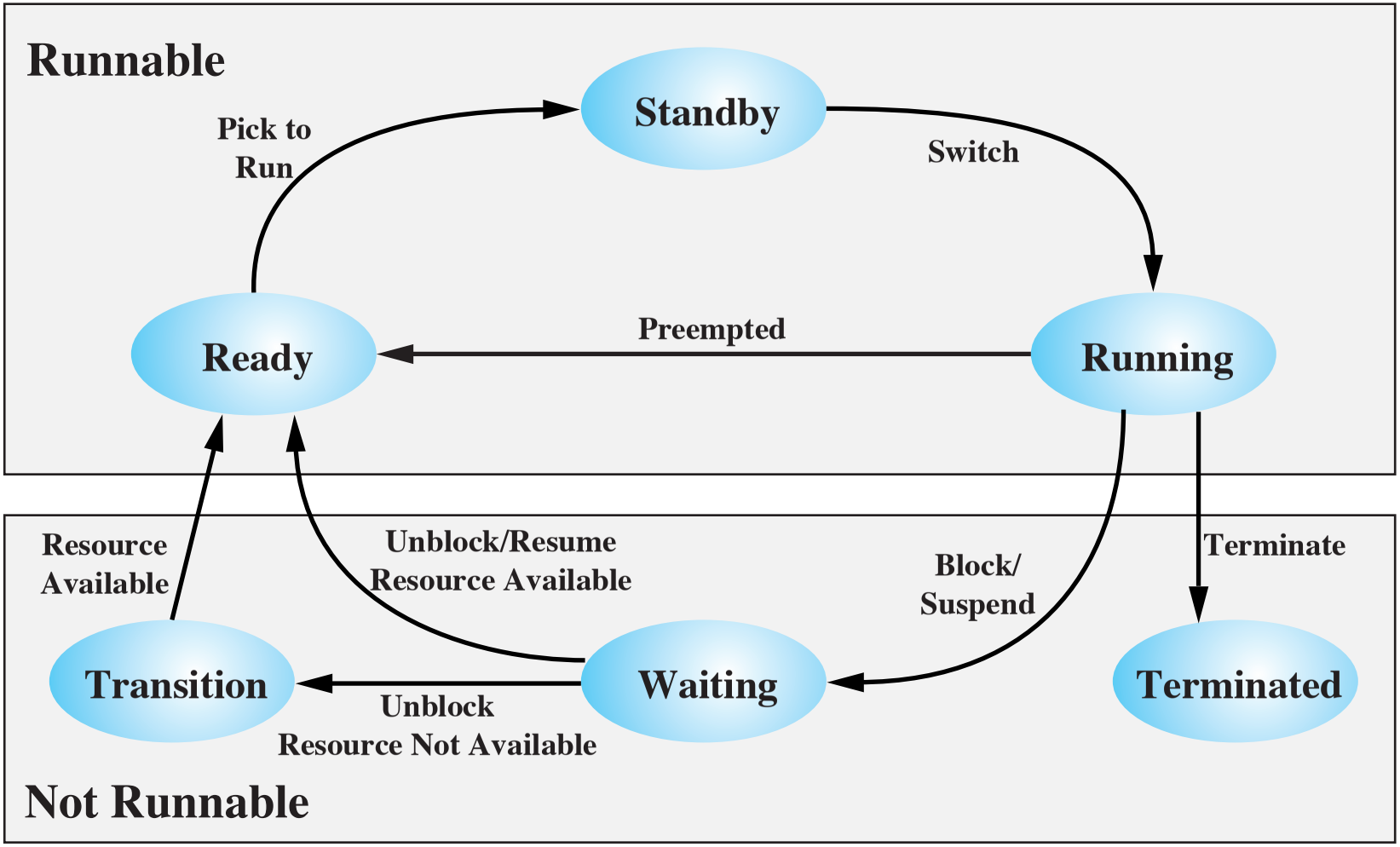


Figure 4.14 Windows Thread States

- **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[1] 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 subsystem to exploit the Windows process and thread features to emulate the process and thread facilities of its corresponding operating system. 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 and OS/2, 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

---

[1] 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.

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 or OS/2, 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 XP and 2003 provide synchronization among threads as part of the object architecture. The two most important methods of synchronization are synchronization objects and critical section objects. Synchronization objects make use of wait functions. We first describe wait functions and then look at the two object types.

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.

Synchronization Objects

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

The first four 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 synchronization 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 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 all thread objects that are waiting on that synchronization 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 Section Objects

Critical section objects provide a synchronization mechanism similar to that provided by mutex objects, except that critical section objects 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 section objects provide a slightly 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 object 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.

Table 6.7 Windows Synchronization Objects

Object Type	Definition	Set to Signaled State When	Effect on Waiting Threads
Event	An announcement that a system event has occurred	Thread sets the event	All 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 change notification	A notification of any file system changes.	Change occurs in file system that matches filter criteria of this object	One thread released
Console input	A text window screen buffer (e.g., used to handle screen I/O for an MS-DOS application)	Input is available for processing	One thread released
Job	An instance of an opened file or I/O device	I/O operation completes	All released
Memory resource notification	A notification of change to a memory resource	Specified type of change occurs within physical memory	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: Colored rows correspond to objects that exist for the sole purpose of synchronization.

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, PowerPC, and MIPS platforms have 4096 bytes per page and DEC Alpha platforms have 8192 bytes per page.

Windows Virtual Address Map

Each Windows user process sees a separate 32-bit address space, allowing 4 Gbytes of memory per process. By default, a portion of this memory is reserved for the operating system, so each user actually has 2 Gbyte 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. The Windows documentation indicates that 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.

Figure 8.26 shows the default virtual address space seen by a 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, microkernel, and device drivers.

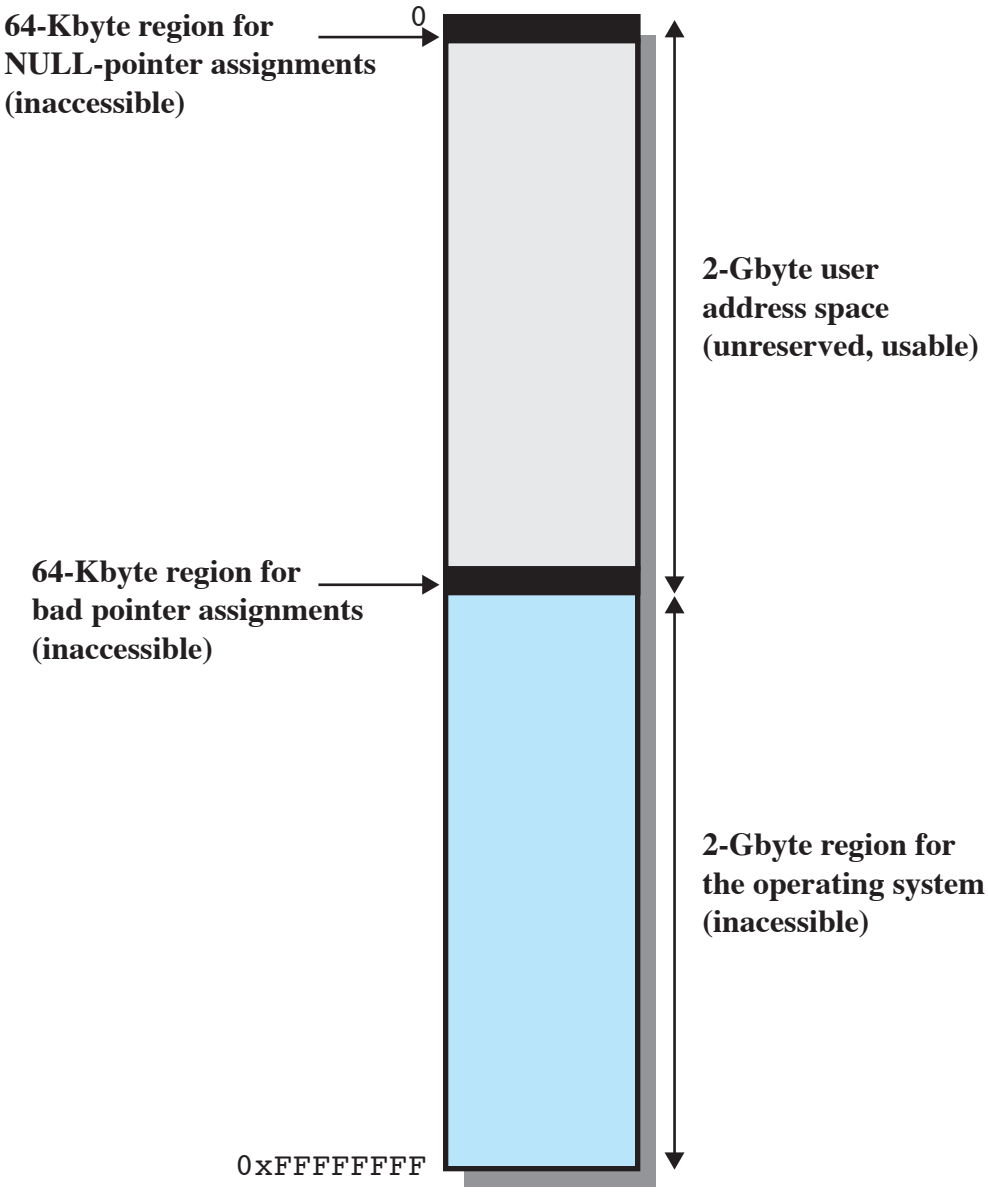


Figure 8.26 Windows Default Virtual Address Space

Windows Paging

When a process is created, it can in principle make use of the entire user space of 2 Gbytes (minus 128 Kbytes). This space is divided into fixed-size pages, any of which can be brought into main memory. In practice, to simplify the accounting, a page can be in one of three states:

- **Available:** Pages not currently used by this process.
- **Reserved:** A set of contiguous pages that the virtual memory manager sets aside for a process but does not count against the process's memory quota until used. When a process needs to write to memory, some of the reserved memory is committed to the process.
- **Committed:** Pages for which the virtual memory manager has set aside space in its paging file (e.g., the disk file to which it writes pages when removing them from main memory).

The distinction between reserved and committed memory is useful because it (1) minimizes the amount of disk space set aside for a particular process, keeping that disk space free for other processes; and (2) enables a thread or process to declare an amount of memory that can be quickly allocated as needed.

The resident set management scheme used by Windows is variable allocation, local scope (see Table 8.4). When a process is first activated, it is assigned a certain number of page frames of main memory as its working set. When a process references a page not in memory, one of the resident pages of that process is swapped out and the new page is brought in. Working sets of active processes are adjusted using the following general conventions:

- When main memory is plentiful, the virtual memory manager allows the resident sets of active processes to grow. To do this, when a page fault occurs, a new page is brought into memory but no older page is swapped out, resulting in an increase of the resident set of that process by one page.

- When memory becomes scarce, the virtual memory manager recovers memory for the system by moving less recently used pages out of the working sets of active processes, reducing the size of those resident sets.

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.

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 10.14). 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 variable priority class, a thread's priority begins at some initial assigned value and then may change, up or down, during the thread's lifetime. Thus, there is a FIFO queue at each priority level, but a process may migrate to one of the other queues within the variable priority class. However, a thread at priority level 15 cannot be promoted 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. One of the attributes of a process object is process base priority, which 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

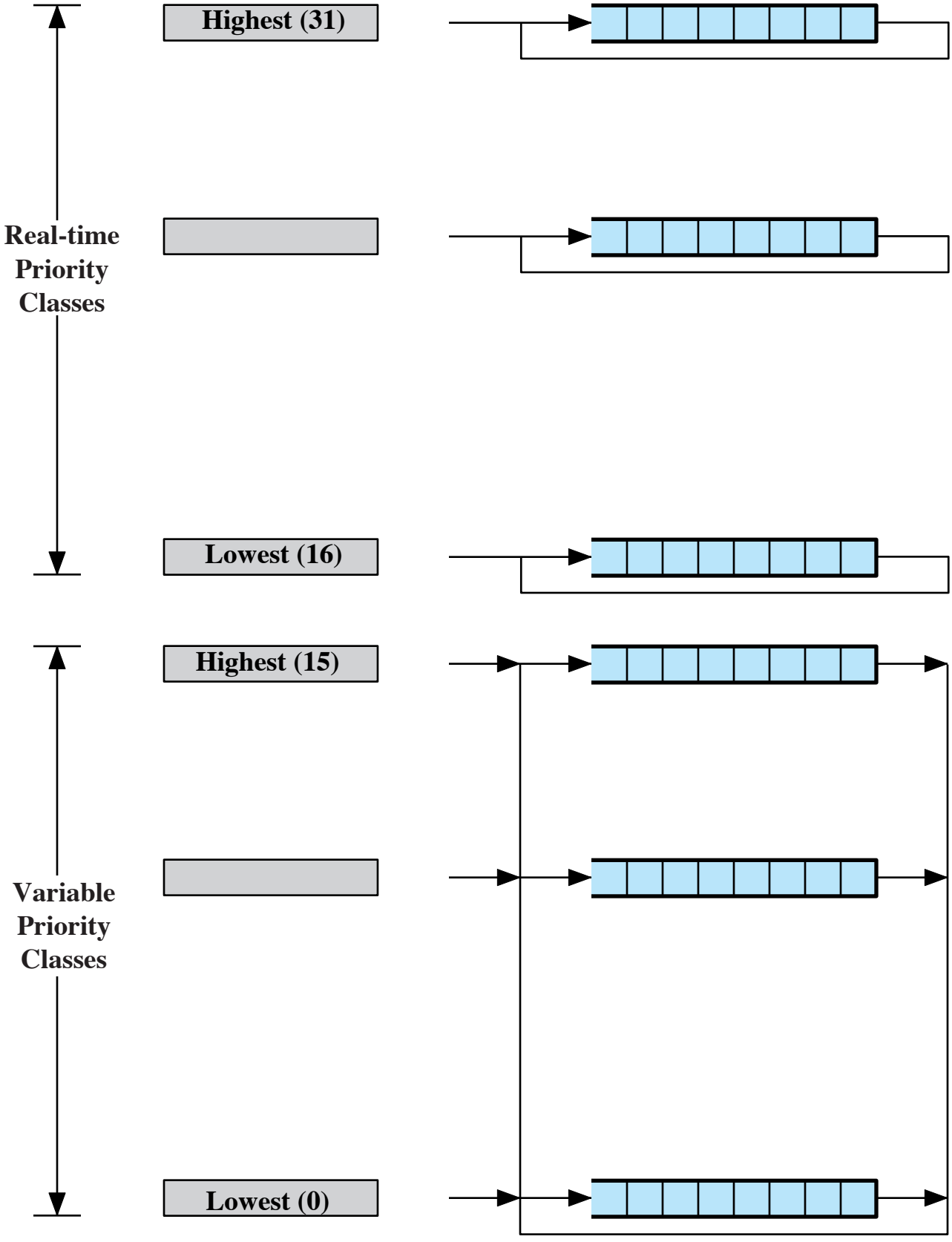


Figure 10.14 Windows Thread Dispatching Priorities

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 dynamic priority, may fluctuate within given boundaries. The dynamic priority may never fall below the lower range of the thread's base priority and it may never exceed 15. Figure 10.15 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. The dynamic priority for each thread may fluctuate in the range from 2 through 15. If a thread is interrupted because it has used up its current time quantum, the Windows executive lowers its priority. If a thread is interrupted to wait on an I/O event, the Windows executive raises 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 executive raises the priority more for interactive waits (e.g., wait on keyboard or display) than for other 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 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 $(N - 1)$ highest priority threads are always active, running exclusively on the $(N - 1)$ extra processors. The remaining, lower-priority, threads share the single remaining processor. For example, if there are three processors, the two highest-priority threads run on two processors, while all remaining threads run on the remaining processor.

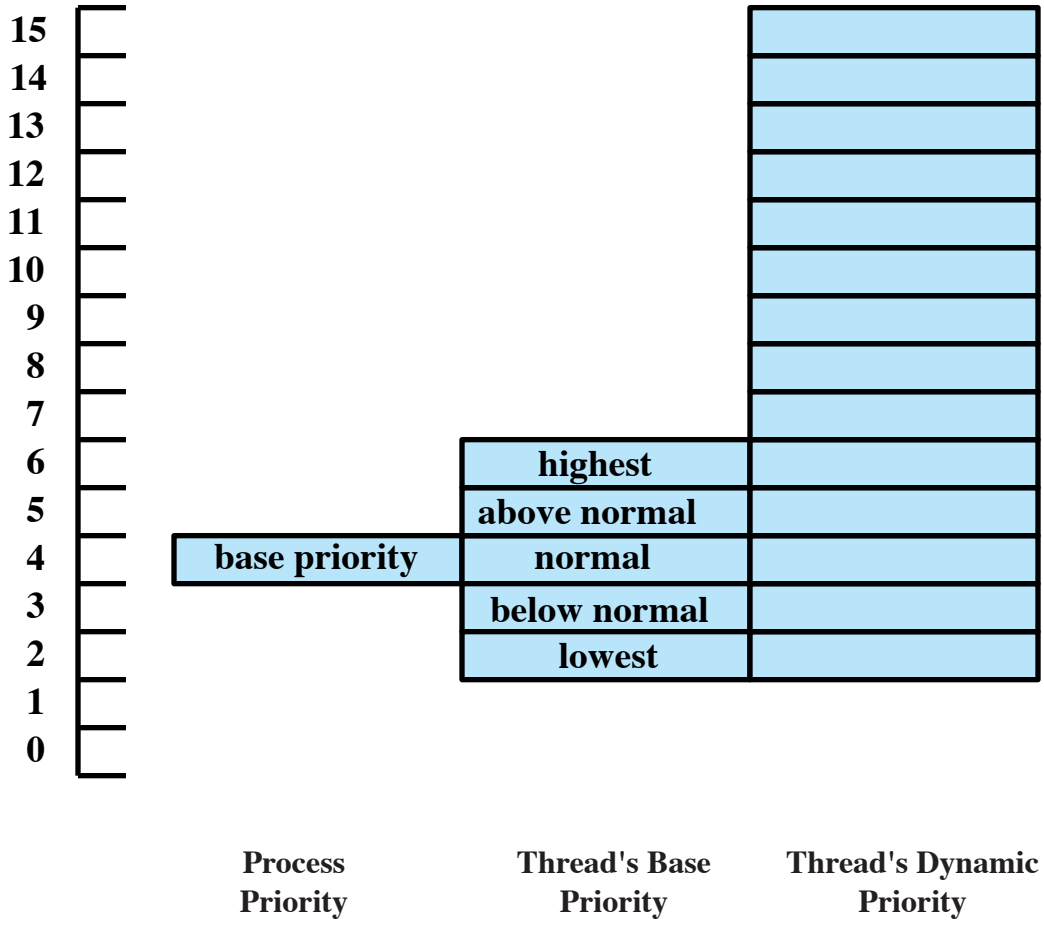


Figure 10.15 Example of Windows Priority Relationship

The foregoing 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 executive schedules the next available thread.

11.10 WINDOWS I/O

Figure 11.15 shows 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 Modules

The I/O manager consists of four modules:

- **Cache manager:** The cache manager handles caching for the entire I/O subsystem. The cache manager provides a caching service in main memory to all file systems and network components. It can dynamically increase and decrease the size of the cache devoted to a particular activity as the amount of available physical memory varies. Cache manager includes two services to improve overall performance:
  - —**Lazy write:** The system records updates in the cache only and not on disk. Later, when demand on the processor is low, the cache manager writes the changes to disk. If a particular cache block is updated in the meantime, there is a net savings.
  - —**Lazy commit:** This is similar to lazy write for transaction processing. Instead of immediately marking a transaction as successfully completed, the system caches the committed information and later writes it to the file system log by a background process.
- **File system drivers:** The I/O manager treats a file system driver as just another device driver and routes message for certain volumes to the appropriate software driver for that device adapter.
- **Network drivers:** Windows includes integrated networking capabilities and support for distributed applications.
- **Hardware device drivers:** These drivers access the hardware registers of the peripheral devices through entry points in Windows Executive dynamic link libraries. A set of these routines exists for every platform that Windows supports; because the routine names are

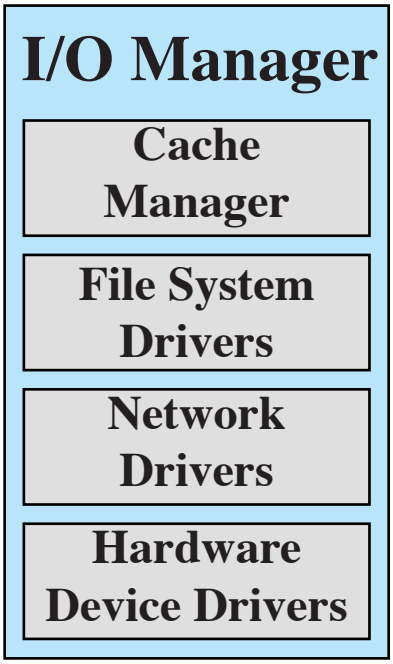


Figure 11.15 Windows I/O Manager

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 four different techniques for signaling I/O completion:

- **Signaling a device kernel object:** With this approach, an indicator associated with a device 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 kernel 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 the entire collection of requests.

- **Alertable I/O:** 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, and the I/O manager places the results of these requests in the calling thread's APC queue.
- **I/O completion ports:** This technique is used on a Windows server to optimize the use of threads. In essence, a pool of threads is available for use so that it is not necessary to create a new thread to handle a new request.

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*.

12.9 WINDOWS FILE SYSTEM

Windows supports a number of file systems, including the file allocation table (FAT) that runs on Windows 95, MS-DOS, and OS/2. But the developers of Windows also designed a new file system, the Windows File System (NTFS), that is intended to meet high-end requirements for workstations and servers. Examples of high-end applications:

- 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 which shall see, on an elegantly simple file system model. The most noteworthy features of NTFS include:

- **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 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.
- **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.
- **General indexing facility:** NTFS associates a collection of attributes with each file. The set of file descriptions in the file management system is organized as a relational database, so that files can be indexed by any attribute.

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.

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 12.6 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.

Table 12.6 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

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 12.17 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 as well as information about available unallocated space. In essence, the MFT is a list of all contents 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
- **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 variable-length rows, called records. Each row describes a file or a folder on this volume,

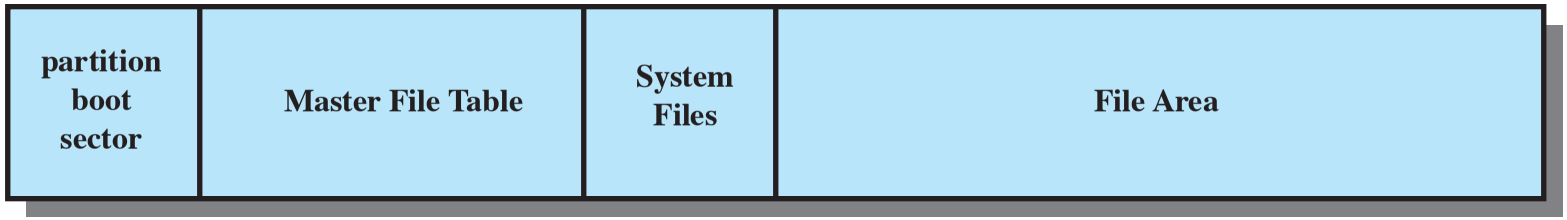


Figure 12.17 NTFS Volume Layout

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 12.7 lists the attributes that may be found in a row, with the required attributes indicated by shading.

Table 12.7 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.

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 (Figure 12.18):

- **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.
- **Log file service:** Maintains a log of disk writes. The log file is used to recover an NTFS-formatted volume in the case of a system failure.
- **Cache manager:** Responsible for caching file reads and writes to enhance performance. The cache manager optimizes disk I/O by using the lazy write and lazy commit techniques described in Section 11.8.
- **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 data, 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 [CUST94]:

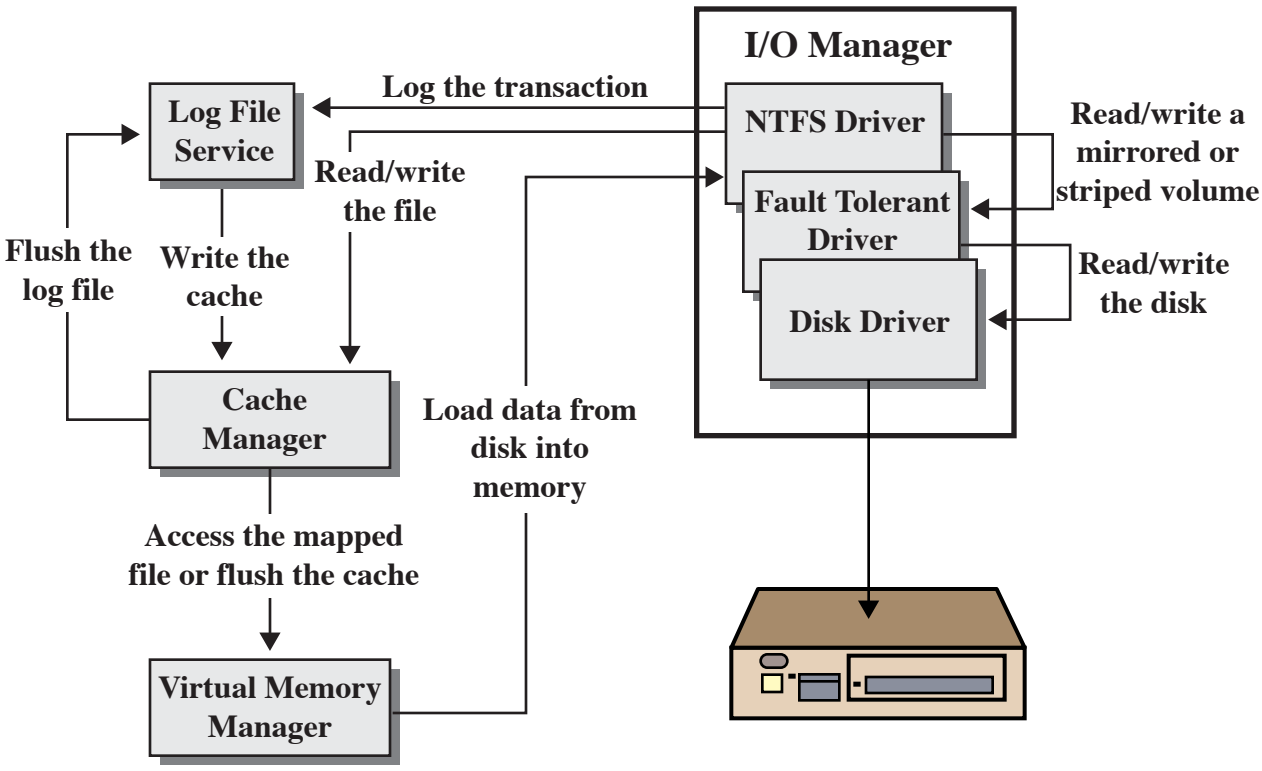


Figure 12.18 Windows NTFS Components

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.

14.5 WINDOWS CLUSTER SERVER

Windows Cluster Server (formerly code named Wolfpack) is a shared-nothing cluster, in which each disk volume and other resources are owned by a single system at a time.

The Windows Cluster Server design makes use of the following concepts:

- **Cluster Service:** The collection of software on each node that manages all cluster-specific activity.
- **Resource:** An item managed by the cluster service. All resources are objects representing actual resources in the system, including hardware devices such as disk drives and network cards and logical items such as logical disk volumes, TCP/IP addresses, entire applications, and databases.
- **Online:** A resource is said to be online at a node when it is providing service on that specific node.
- **Group:** A collection of resources managed as a single unit. Usually, a group contains all of the elements needed to run a specific application and for client systems to connect to the service provided by that application.

The concept of group is of particular importance. A group combines resources into larger units that are easily managed, both for failover and load balancing. Operations performed on a group, such as transferring the group to another node, automatically affect all of the resources in that group. Resources are implemented as dynamically linked libraries (DLLs) and managed by a resource monitor. The resource monitor interacts with the cluster service via remote procedure calls and responds to cluster service commands to configure and move resource groups.

Figure 14.15 depicts the Windows Cluster Server components and their relationships in a single system of a cluster. The **node manager** is responsible for maintaining this node's membership in the cluster. Periodically, it sends heartbeat messages to the node managers on

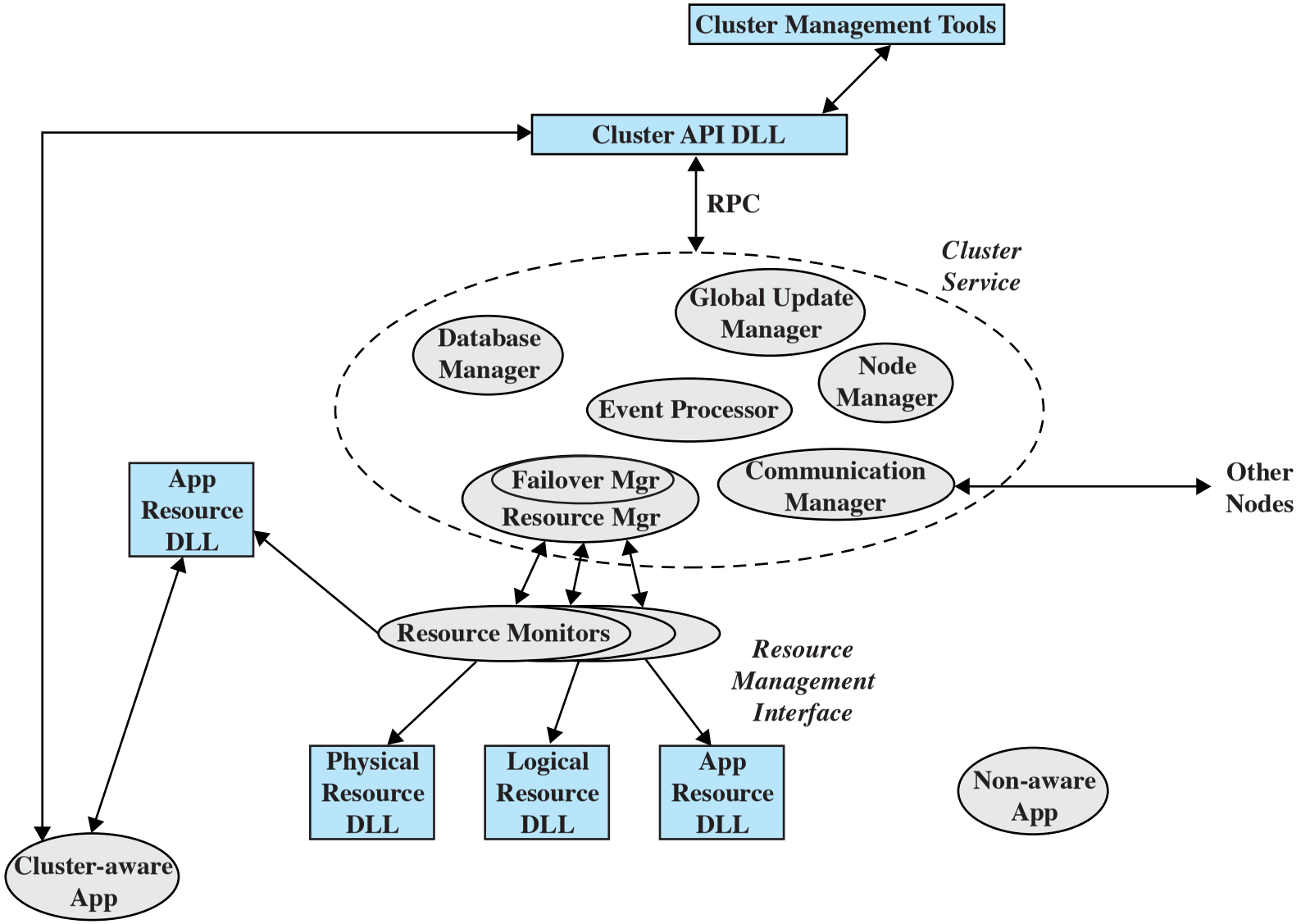


Figure 14.15 Windows Cluster Server Block Diagram [SHOR97]

other nodes in the cluster. In the event that one node manager detects a loss of heartbeat messages from another cluster node, it broadcasts a message to the entire cluster, causing all members to exchange messages to verify their view of current cluster membership. If a node manager does not respond, it is removed from the cluster and its active groups are transferred to one or more other active nodes in the cluster.

The **configuration database manager** maintains the cluster configuration database. The database contains information about resources and groups and node ownership of groups. The database managers on each of the cluster nodes cooperate to maintain a consistent picture of configuration information. Fault-tolerant transaction software is used to assure that changes in the overall cluster configuration are performed consistently and correctly.

The **resource manager/failover manager** makes all decisions regarding resource groups and initiates appropriate actions such as startup, reset, and failover. When failover is required, the failover managers on the active node cooperate to negotiate a distribution of resource groups from the failed system to the remaining active systems. When a system restarts after a failure, the failover manager can decide to move some groups back to this system. In particular, any group may be configured with a preferred owner. If that owner fails and then restarts, the group is moved back to the node in a rollback operation.

The **event processor** connects all of the components of the cluster service, handles common operations, and controls cluster service initialization. The communications manager manages message exchange with all other nodes of the cluster. The global update manager provides a service used by other components within the cluster service.

16.6 WINDOWS 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. When any additional processes are spawned by the initial user 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 keep all of the security information for a user in one systemwide place, 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.

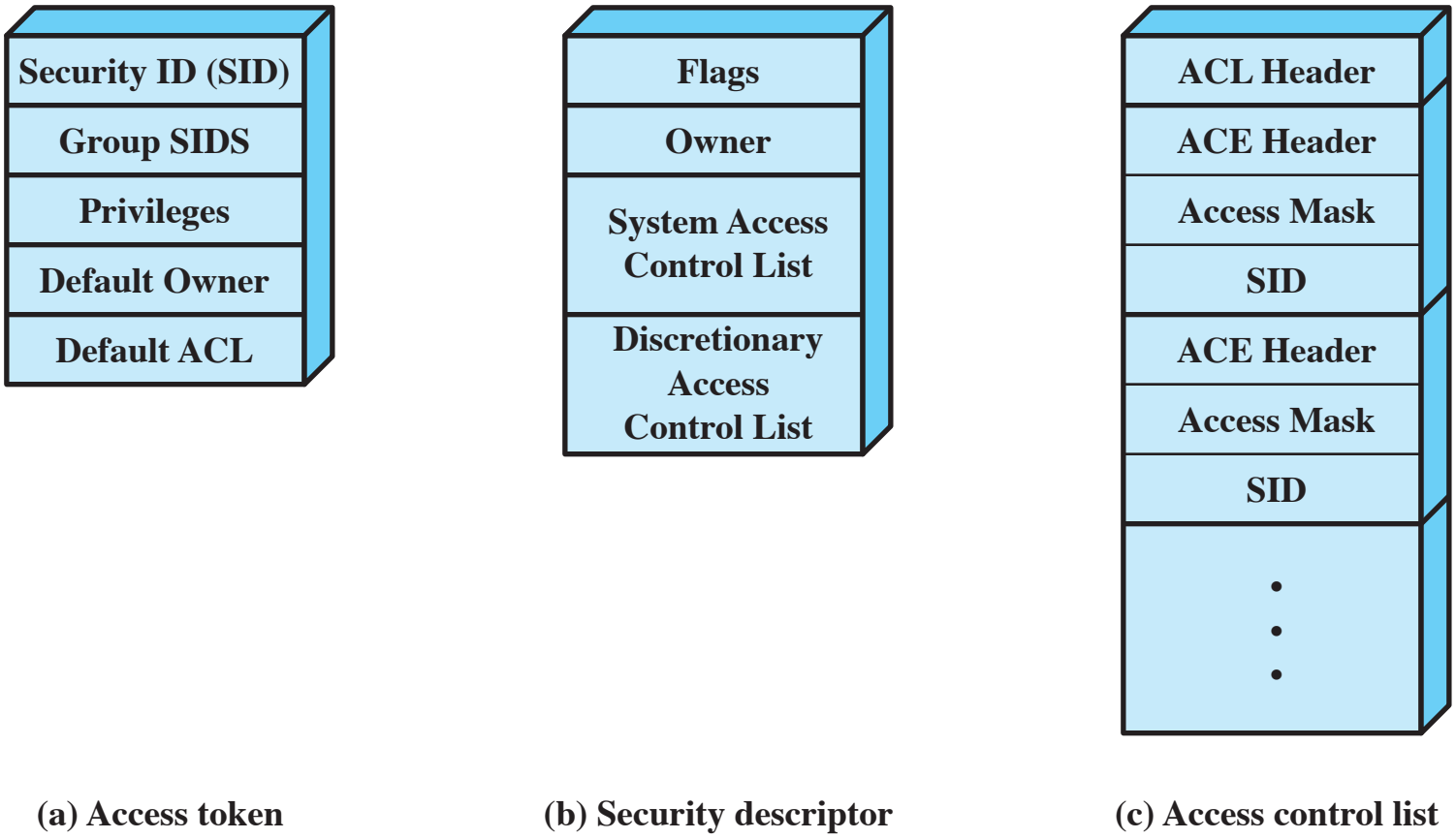
When an application opens a reference to a securable object, Windows verifies that the object's security descriptor grants the application's user access. If the check succeeds, Windows caches the resulting granted access rights.

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 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 16.12a 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



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

Figure 16.12 Windows Security Structures

unique group SID. Access to an object can be defined on the basis of group SIDs, individual SIDs, or a combination.

- **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. Most users will have no privileges.
- **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 16.12b 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).
- **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 16.12c). 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 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 has the access rights specified by the access mask in that ACE.

Figure 16.13 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.

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:

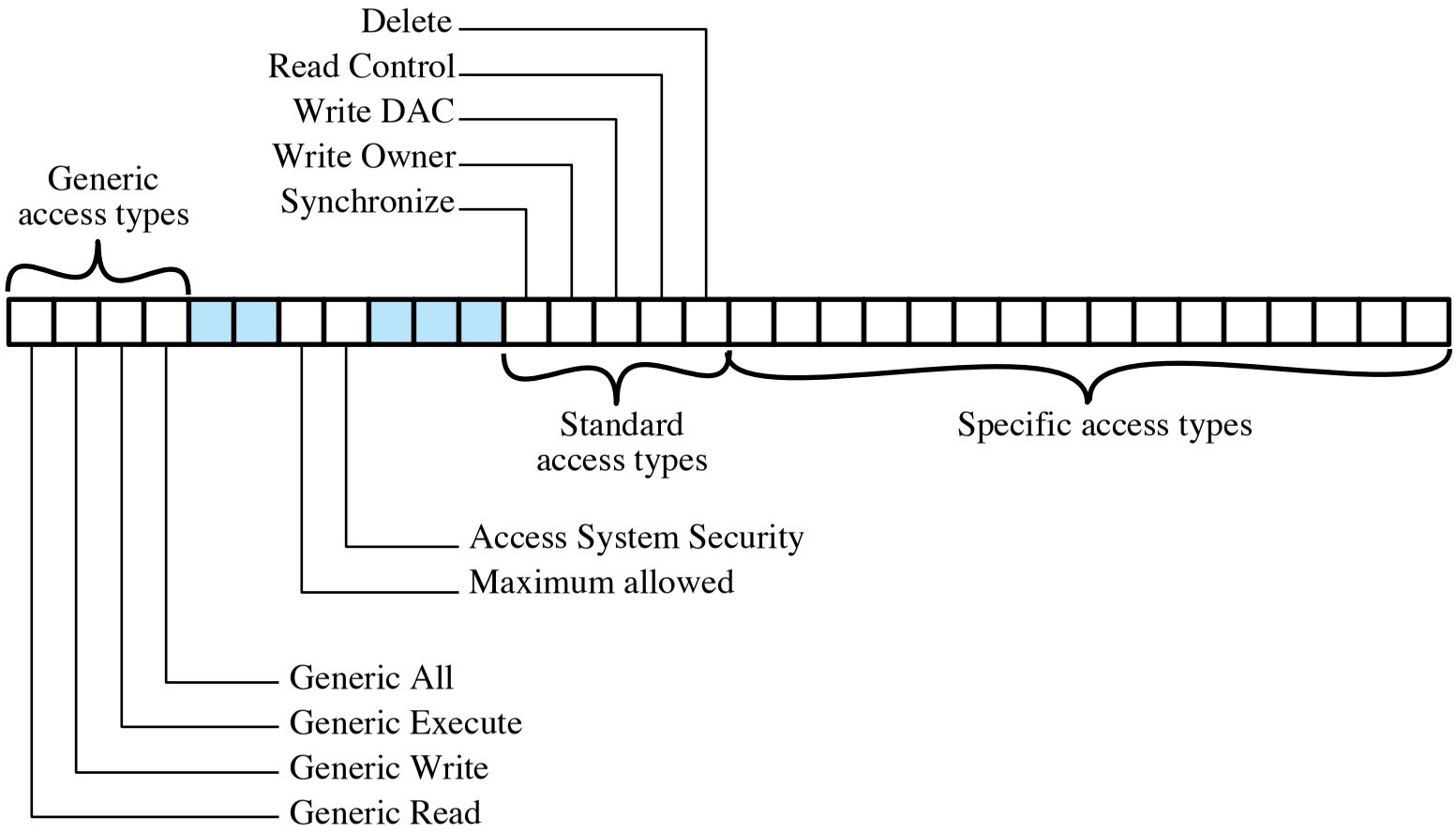


Figure 16.13 Access Mask

- **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.