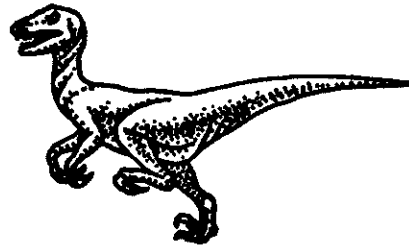


Windows XP



The Microsoft Windows XP operating system is a 32/64-bit preemptive multitasking operating system for AMD K6/K7, Intel IA32/IA64, and later microprocessors. The successor to Windows NT and Windows 2000, Windows XP is also intended to replace the Windows 95/98 operating system. Key goals for the system are security, reliability, ease of use, Windows and POSIX application compatibility, high performance, extensibility, portability, and international support. In this chapter, we discuss the key goals of Windows XP, the layered architecture of the system that makes it so easy to use, the file system, the networking features, and the programming interface.

22.1 History

In the mid-1980s, Microsoft and IBM cooperated to develop the OS/2 operating system, which was written in assembly language for single-processor Intel 80286 systems. In 1988, Microsoft decided to make a fresh start and to develop a “new technology” (or NT) portable operating system that supported both the OS/2 and POSIX application-programming interfaces (APIs). In October 1988, Dave Cutler, the architect of the DEC VAX/VMS operating system, was hired and given the charter of building this new operating system.

Originally, the team planned for NT to use the OS/2 API as its native environment, but during development, NT was changed to use the 32-bit Windows API (or Win32 API), reflecting the popularity of Windows 3.0. The first versions of NT were Windows NT 3.1 and Windows NT 3.1 Advanced Server. (At that time, 16-bit Windows was at version 3.1.) Windows NT version 4.0 adopted the Windows 95 user interface and incorporated Internet web-server and web-browser software. In addition, user-interface routines and all graphics code were moved into the kernel to improve performance, with the side effect of decreased system reliability. Although previous versions of NT had been ported to other microprocessor architectures, the Windows 2000 version, released in February 2000, discontinued support for other than Intel (and compatible) processors due to marketplace factors. Windows 2000 incorporated significant changes over Windows NT. It added Active Directory (an X.500-based directory service), better networking and laptop support, support for plug-and-play

devices, a distributed file system, and support for more processors and more memory.

In October 2001, Windows XP was released as both an update to the Windows 2000 desktop operating system and a replacement for Windows 95/98. In 2002, the server versions of Windows XP became available (called Windows .Net Server). Windows XP updates the graphical user interface (GUI) with a visual design that takes advantage of more recent hardware advances and many new **ease-of-use** features. Numerous features have been added to automatically repair problems in applications and the operating system itself. Windows XP provides better networking and device experience (including zero-configuration wireless, instant messaging, streaming media, and digital photography/video), dramatic performance improvements both for the desktop and large multiprocessors, and better reliability and security than even Windows 2000.

Windows XP uses a client-server architecture (like Mach) to implement multiple operating-system personalities, such as Win32 API and POSIX, with user-level processes called subsystems. The subsystem architecture allows enhancements to be made to one operating-system personality without affecting the application compatibility of any others.

Windows XP is a multiuser operating system, supporting simultaneous access through distributed services or through multiple instances of the graphical user interface via the Windows terminal server. The server versions of Windows XP support simultaneous terminal server sessions from Windows desktop systems. The desktop versions of terminal server multiplex the keyboard, mouse, and monitor between virtual terminal sessions for each logged-on user. This feature, called fast user switching, allows users to preempt each other at the console of a PC without having to log off and onto the system.

Windows XP is the first version of Windows to ship a 64-bit version. The native NT file system (NTFS) and many of the Win32 APIs have always used 64-bit integers where appropriate—so the major extension to 64-bit in Windows XP is support for large addresses.

There are two desktop versions of Windows XP. Windows XP Professional is the premium desktop system for power users at work and at home. For home users migrating from Windows 95/98, Windows XP Personal provides the reliability and ease of use of Windows XP, but lacks the more advanced features needed to work seamlessly with Active Directory or run POSIX applications.

The members of the Windows .Net Server family use the same core components as the desktop versions but add a range of features needed for uses such as webserver farms, print/file servers, clustered systems, and large datacenter machines. The large datacenter machines can have up to 64 GB of memory and 32 processors on IA32 systems and 128 GB and 64 processors on IA64 systems.

22.2

Microsoft's design goals for Windows XP include security, reliability, Windows and POSIX application compatibility, high performance, extensibility, portability, and international support.

22.2.1 Security

Windows XP **security** goals required more than just adherence to the design standards that enabled Windows NT 4.0 to receive a C-2 security classification from the U.S. government (which signifies a moderate level of protection from defective software and malicious attacks). Extensive code review and testing were combined with sophisticated automatic analysis tools to identify and investigate potential defects that might represent security vulnerabilities.

22.2.2 Reliability

Windows 2000 was the most reliable, stable operating system Microsoft had ever shipped to that point. Much of this reliability came from maturity in the source code, extensive stress testing of the system, and automatic detection of many serious errors in drivers. The **reliability** requirements for Windows XP were even more stringent. Microsoft used extensive manual and automatic code review to identify over 63,000 lines in the source files that might contain issues not detected by testing and then set about reviewing each area to verify that the code was indeed correct.

Windows XP extends driver verification to catch more subtle bugs, improves the facilities for catching programming errors in user-level code, and subjects third-party applications, drivers, and devices to a rigorous certification process. Furthermore, Windows XP adds new facilities for monitoring the health of the PC, including downloading fixes for problems before they are encountered by users. The perceived reliability of Windows XP was also improved by making the graphical user interface easier to use through better visual design, simpler menus, and measured improvements in the ease with which users can discover how to perform common tasks.

22.2.3 Windows and POSIX Application Compatibility

Windows XP is not only an update of Windows 2000; it is a replacement for Windows 95/98. Windows 2000 focused primarily on compatibility for business applications. The requirements for Windows XP include a much higher compatibility with consumer applications that run on Windows 95/98. **Application compatibility** is difficult to achieve because each application checks for a particular version of Windows, may have some dependence on the quirks of the implementation of APIs, may have latent application bugs that were masked in the previous system, and so forth.

Windows XP introduces a compatibility layer that falls between applications and the Win32 APIs. This layer makes Windows XP look (almost) bug-for-bug compatible with previous versions of Windows. Windows XP, like earlier NT releases, maintains support for running many 16-bit applications using a *thunking*, or conversion, layer that translates 16-bit API calls into equivalent 32-bit calls. Similarly, the 64-bit version of Windows XP provides a *thunking* layer that translates 32-bit API calls into native 64-bit calls. POSIX support in Windows XP is much improved. A new POSIX subsystem called Interix is now available. Most available UNIX-compatible software compiles and runs under Interix without modification.

22.2.4 High Performance

Windows XP is designed to provide **high performance** on desktop systems (which are largely constrained by I/O performance), server systems (where the CPU is often the bottleneck), and large multithreaded and multiprocessor environments (where locking and cache-line management are key to scalability). High performance has been an increasingly important goal for Windows XP. Windows 2000 with SQL 2000 on Compaq hardware achieved top TPC-C numbers at the time it shipped.

To satisfy performance requirements, NT uses a variety of techniques, such as asynchronous I/O, optimized protocols for networks (for example, optimistic locking of distributed data, batching of requests), kernel-based graphics, and sophisticated caching of file-system data. The memory-management and synchronization algorithms are designed with an awareness of the performance considerations related to cache lines and multiprocessors.

Windows XP has further improved performance by reducing the code-path length in critical functions, using better algorithms and per-processor data structures, using memory coloring for NUMA (non-uniform memory access) machines, and implementing more scalable locking protocols, such as queued spinlocks. The new locking protocols help reduce system bus cycles and include lock-free lists and queues, use of atomic read-modify-write operations (like interlocked increment), and other advanced locking techniques.

The subsystems that constitute Windows XP communicate with one another efficiently by a local procedure call (LPC) facility that provides high-performance message passing. Except while executing in the kernel dispatcher, threads in the subsystems of Windows XP can be preempted by higher-priority threads. Thus, the system responds quickly to external events. In addition, Windows XP is designed for symmetrical multiprocessing; on a multiprocessor computer, several threads can run at the same time.

22.2.5 Extensibility

Extensibility refers to the capacity of an operating system to keep up with advances in computing technology. So that changes over time are facilitated, the developers implemented Windows XP using a layered architecture. The Windows XP executive runs in kernel or protected mode and provides the basic system services. On top of the executive, several server subsystems operate in user mode. Among them are **environmental subsystems** that emulate different operating systems. Thus, programs written for MS-DOS, Microsoft Windows, and POSIX all run on Windows XP in the appropriate environment. (See Section 22.4 for more information on environmental subsystems.) Because of the modular structure, additional environmental subsystems can be added without affecting the executive. In addition, Windows XP uses loadable drivers in the I/O system, so new file systems, new kinds of I/O devices, and new kinds of networking can be added while the system is running. Windows XP uses a client-server model like the Mach operating system and supports distributed processing by remote procedure calls (RPCs) as defined by the Open Software Foundation.

22.2.6 Portability

An operating system is **portable** if it can be moved from one hardware architecture to another with relatively few changes. Windows XP is designed to be portable. As is true of the UNIX operating system, the majority of the system is written in C and C++. Most processor-dependent code is isolated in a dynamic link library (DLL) called the **hardware-abstraction layer (HAL)**. A DLL is a file that is mapped into a process's address space such that any functions in the DLL appear to be part of the process. The upper layers of the Windows XP kernel depend on the HAL interfaces rather than on the underlying hardware, bolstering Windows XP portability. The HAL manipulates hardware directly, isolating the rest of Windows XP from hardware differences among the platforms on which it runs.

Although for market reasons Windows 2000 shipped only on Intel IA32-compatible platforms, it was also tested on IA32 and DEC Alpha platforms until just prior to release to ensure portability. Windows XP runs on IA32-compatible and IA64 processors. Microsoft recognizes the importance of multiplatform development and testing, since, as a practical matter, maintaining portability is a matter of *use it or lose it*.

22.2.7 International Support


Windows XP is also designed for **international** and **multinational** use. It provides support for different locales via the **national-language-support (NLS) API**. The NLS API provides specialized routines to format dates, time, and money in accordance with various national customs. String comparisons are specialized to account for varying character sets. UNICODE is Windows XP's native character code. Windows XP supports ANSI characters by converting them to UNICODE characters before manipulating them (8-bit to 16-bit conversion). System text strings are kept in resource files that can be replaced to localize the system for different languages. Multiple locales can be used concurrently, which is important to multilingual individuals and businesses.

22.3 Architecture of Windows XP

The architecture of Windows XP is a layered system of modules, as shown in Figure 22.1. The main layers are the HAL, the kernel, and the executive, all of which run in protected mode, and a collection of subsystems and services that run in user mode. The user-mode subsystems fall into two categories: the **environmental subsystems**, which emulate different operating systems, and the **protection subsystems**, which provide security functions. One of the chief advantages of this type of architecture is that interactions between modules are kept simple. The remainder of this section describes these layers and subsystems.

22.3.1 Hardware-Abstraction Layer

The HAL is the layer of software that hides hardware differences from upper levels of the operating system, to help make Windows XP portable. The HAL



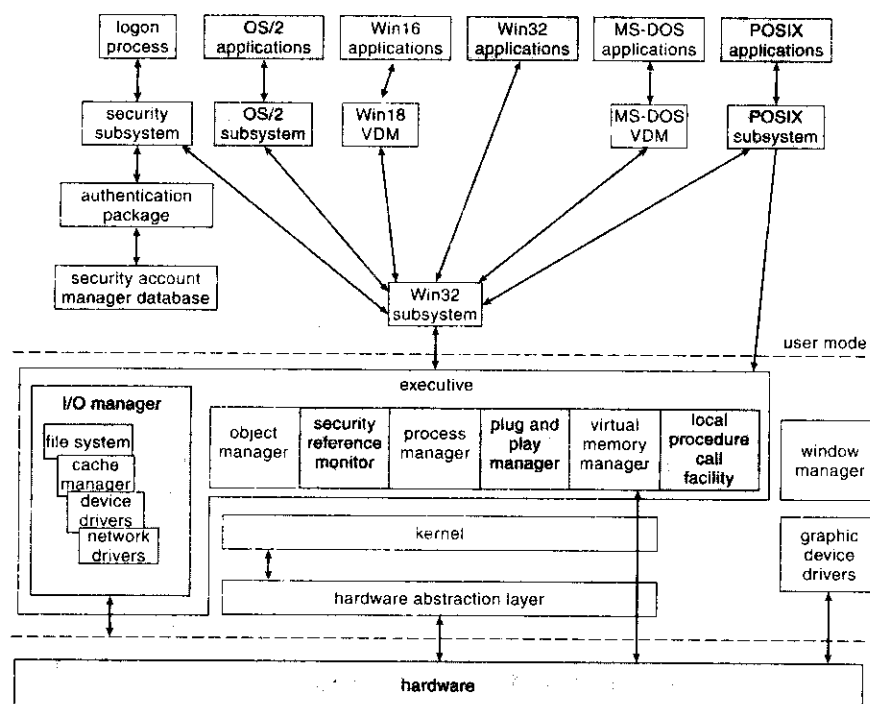


Figure 22.1 Windows XP block diagram.

exports a virtual machine interface that is used by the kernel dispatcher, the executive, and the device drivers. One advantage of this approach is that only a single version of each device driver is required—it runs on all hardware platforms without porting the driver code. The HAL also provides support for symmetric multiprocessing. Device drivers map devices and access them directly, but the administrative details of mapping memory, configuring I/O buses, setting up DMA, and coping with motherboard-specific facilities are all provided by the HAL interfaces.

22.3.2 Kernel

The kernel of Windows XP provides the foundation for the executive and the subsystems. The kernel remains in memory, and its execution is never preempted. It has four main responsibilities: thread scheduling, interrupt and exception handling, low-level processor synchronization, and recovery after a power failure.

The kernel is object oriented. An *object type* in Windows 2000 is a system-defined data type that has a set of attributes (data values) and a set of methods (for example, functions or operations). An *object* is an instance of an object type. The kernel performs its job by using a set of kernel objects whose attributes store the kernel data and whose methods perform the kernel activities.

22.3.2.1 Kernel Dispatcher

The kernel dispatcher provides the foundation for the executive and the subsystems. Most of the dispatcher is never paged out of memory, and its execution is never preempted. Its main responsibilities are thread scheduling, implementation of synchronization primitives, timer management, software interrupts (asynchronous and deferred procedure calls), and exception dispatching.

22.3.2.2 Threads and Scheduling

Like many other modern operating systems, Windows XP uses processes and threads for executable code. The process has a virtual memory address space and information used to initialize each thread, such as a base priority and an affinity for either one or more processors. Each process has one or more threads, each of which is an executable unit dispatched by the kernel. Each thread has its own scheduling state, including actual priority, processor affinity, and CPU-usage information.

The six possible thread states are ready, standby, running, waiting, transition, and terminated. **Ready** indicates that the thread is waiting to run. The highest-priority ready thread is moved to the **standby** state, which means it is the next thread to run. In a multiprocessor system, each process keeps one thread in a standby state. A thread is **running** when it is executing on a processor. It runs until it is preempted by a higher-priority thread, until it terminates, until its allotted execution time (**quantum**) ends, or until it blocks on a dispatcher object, such as an event signaling I/O completion. A thread is in the **waiting** state when it is waiting for a dispatcher object to be signaled. A new thread is in the **transition** state while it waits for resources necessary for execution. A thread enters the **terminated** state when it finishes execution.

The dispatcher uses a 32-level priority scheme to determine the order of thread execution. Priorities are divided into two classes: variable class and real-time class. The variable class contains threads having priorities from 0 to 15, and the real-time class contains threads with priorities ranging from 16 to 31. The dispatcher uses a queue for each scheduling priority and traverses the set of queues from highest to lowest until it finds a thread that is ready to run. If a thread has a particular processor affinity but that processor is not available, the dispatcher skips past it and continues looking for a ready thread that is willing to run on the available processor. If no ready thread is found, the dispatcher executes a special thread called the idle thread.

When a thread's time quantum runs out, the clock interrupt queues a quantum-end deferred procedure call (DPC) to the processor in order to reschedule the processor. If the preempted thread is in the variable-priority class, its priority is lowered. The priority is never lowered below the base priority. Lowering the thread's priority tends to limit the CPU consumption of compute-bound threads. When a variable-priority thread is released from a wait operation, the dispatcher boosts the priority. The amount of the boost depends on the device for which the thread was waiting; for example, a thread waiting for keyboard I/O would get a large priority increase, whereas a thread waiting for a disk operation would get a moderate one. This strategy tends to give good response times to interactive threads using a mouse and windows. It also enables I/O-bound threads to keep the I/O devices busy while permitting compute-bound threads to use spare CPU cycles in the background.

This strategy is used by several time-sharing operating systems, including UNIX. In addition, the thread associated with the user's active GUI window receives a priority boost to enhance its response time.

Scheduling occurs when a thread enters the ready or wait state, when a thread terminates, or when an application changes a thread's priority or processor affinity. If a higher-priority real-time thread becomes ready while a lower-priority thread is running, the lower-priority thread is preempted. This preemption gives a real-time thread preferential access to the CPU when the thread needs such access. Windows XP is not a hard real-time operating system, however, because it does not guarantee that a real-time thread will start to execute within a particular time limit.

22.3.2.3 Implementation of Synchronization Primitives

Key operating-system data structures are managed as objects using common facilities for allocation, reference counting, and security. **Dispatcher objects** control dispatching and synchronization in the system. Examples of these objects are events, mutants, mutexes, semaphores, processes, threads, and timers. The **event object** is used to record an event occurrence and to synchronize the latter with some action. Notification events signal all waiting threads, and synchronization events signal a single waiting thread. The **mutant** provides kernel-mode or user-mode mutual exclusion with the notion of ownership. The **mutex**, available only in kernel mode, provides deadlock-free mutual exclusion. A **semaphore object** acts as a counter or gate to control the number of threads that access a resource. The **thread object** is the entity that is scheduled by the kernel dispatcher and is associated with a **process object**, which encapsulates a virtual address space. **Timer objects** are used to keep track of time and to signal timeouts when operations take too long and need to be interrupted or when a periodic activity needs to be scheduled.

Many of the dispatcher objects are accessed from user mode via an open operation that returns a handle. The user-mode code polls and/or waits on handles to synchronize with other threads as well as the operating system (see Section 22.7.1).

22.3.2.4 Software Interrupts: Asynchronous and Deferred Procedure Calls

The dispatcher implements two types of software interrupts: asynchronous procedure calls and deferred procedure calls. Asynchronous procedure calls (APCs) break into an executing thread and call a procedure. APCs are used to begin execution of a new thread, terminate processes, and deliver notification that an asynchronous (I/O) has completed. APCs are queued to specific threads and allow the system to execute both system and user code within a process's context.

Deferred procedure calls (DPCs) are used to postpone interrupt processing. After handling all blocked device-interrupt processes, the interrupt service routine (ISR) schedules the remaining processing by queuing a DPC. The dispatcher schedules software interrupts at a lower priority than the device interrupts so that DPCs do not block other ISRs. In addition to deferring device-interrupt processing, the dispatcher uses DPCs to process timer expirations and to preempt thread execution at the end of the scheduling quantum.

Execution of DPCs prevents threads from being scheduled on the current processor and also keeps APCs from signaling the completion of I/O. This is done so that DPC routines do not take an extended amount of time to complete. As an alternative, the dispatcher maintains a pool of worker threads. ISRs and DPCs queue work items to the worker threads. DPC routines are restricted so that they cannot take page faults, call system services, or take any other action that might possibly result in an attempt to block execution on a dispatcher object. Unlike APCs, DPC routines make no assumptions about what process context the processor is executing.

22.3.2.5 Exceptions and Interrupts

The kernel dispatcher also provides trap handling for exceptions and interrupts generated by hardware or software. Windows XP defines several architecture-independent exceptions, including:

- Memory-access violation
- Integer overflow
- Floating-point overflow or underflow
- Integer divide by zero
- Floating-point divide by zero
- Illegal instruction
- Data misalignment
- Privileged instruction
- Page-read error
- Access violation
- Paging file quota exceeded
- Debugger breakpoint
- Debugger single step

The trap handlers deal with simple exceptions. Elaborate exception handling is performed by the kernel's exception dispatcher. The **exception dispatcher** creates an exception record containing the reason for the exception and finds an exception handler to deal with it.

When an exception occurs in kernel mode, the exception dispatcher simply calls a routine to locate the exception handler. If no handler is found, a fatal system error occurs and the user is left with the infamous "blue screen of death" that signifies system failure.

Exception handling is more complex for user-mode processes, because an environmental subsystem (such as the POSIX system) sets up a debugger port and an exception port for every process it creates. If a debugger port is registered, the exception handler sends the exception to the port. If the debugger port is not found or does not handle that exception, the dispatcher attempts to find an appropriate exception handler. If no handler is found, the

interrupt levels	types of interrupts
31	machine check or bus error
30	power fail
29	interprocessor notification (request another processor to act; e.g., dispatch a process or update the TLB)
28	clock (used to keep track of time)
27	profile
3–26	traditional PC IRQ hardware interrupts
2	dispatch and deferred procedure call (DPC) (kernel)
1	asynchronous procedure call (APC)
0	passive

Figure 22.2 Windows XP interrupt request levels.

debugger is called again to catch the error for debugging. If no debugger is running, a message is sent to the process's exception port to give the environmental subsystem a chance to translate the exception. For example, the POSIX environment translates Windows XP exception messages into POSIX signals before sending them to the thread that caused the exception. Finally, if nothing else works, the kernel simply terminates the process containing the thread that caused the exception.

The interrupt dispatcher in the kernel handles interrupts by calling either an interrupt service routine (ISR) supplied by a device driver or a kernel trap-handler routine. The interrupt is represented by an interrupt object that contains all the information needed to handle the interrupt. Using an interrupt object makes it easy to associate interrupt-service routines with an interrupt without having to access the interrupt hardware directly.

Different processor architectures, such as Intel and DEC Alpha, have different types and numbers of interrupts. For portability, the interrupt dispatcher maps the hardware interrupts into a standard set. The interrupts are prioritized and are serviced in priority order. There are 32 interrupt request levels (IRQLs) in Windows XP. Eight are reserved for use by the kernel; the remaining 24 represent hardware interrupts via the HAL (although most IA32 systems use only 16). The Windows XP interrupts are defined in Figure 22.2.

The kernel uses an **interrupt-dispatch table** to bind each interrupt level to a service routine. In a multiprocessor computer, Windows XP keeps a separate interrupt-dispatch table for each processor, and each processor's IRQL can be set independently to mask out interrupts. All interrupts that occur at a level equal to or less than the IRQL of a processor are blocked until the IRQL is lowered by a kernel-level thread or by an ISR returning from interrupt processing. Windows XP takes advantage of this property and uses software interrupts to deliver APCs and DPCs, to perform system functions such as synchronizing threads with I/O completion, to start thread dispatches, and to handle timers.

22.3.3 Executive

The Windows XP executive provides a set of services that all environmental subsystems use. The services are grouped as follows: object manager, virtual

memory manager, process manager, local procedure call facility, I/O manager, cache manager, security reference monitor, plug-and-play and security managers, registry, and booting.

22.3.3.1 Object Manager

For managing kernel-mode entities, Windows XP uses a generic set of interfaces that are manipulated by user-mode programs. Windows XP calls these entities *objects*, and the executive component that manipulates them is the **object manager**. Each process has an object table containing entries that track the objects used by the process. User-mode code accesses these objects using an opaque value called a *handle* that is returned by many APIs. Object handles can also be created by duplicating an existing handle, either from the same process or a different process. Examples of objects are semaphores, mutexes, events, processes, and threads. These are all *dispatcher objects*. Threads can block in the kernel dispatcher waiting for any of these objects to be signaled. The process, thread, and virtual memory APIs use process and thread handles to identify the process or thread to be operated on. Other examples of objects include files, sections, ports, and various internal I/O objects. File objects are used to maintain the open state of files and devices. Sections are used to map files. Open files are described in terms of file objects. Local-communication endpoints are implemented as port objects.

The object manager maintains the Windows XP internal name space. In contrast to UNIX, which roots the system name space in the file system, Windows XP uses an abstract name space and connects the file systems as devices.

The object manager provides interfaces for defining both object types and object instances, translating names to objects, maintaining the abstract name space (through internal directories and symbolic links), and managing object creation and deletion. Objects are typically managed using reference counts in protected-mode code and handles in user-mode code. However, some kernel-mode components use the same APIs as user-mode code and thus use handles to manipulate objects. If a handle needs to exist beyond the lifetime of the current process, it is marked as a kernel handle and stored in the object table for the system process. The abstract name space does not persist across reboots but is built up from configuration information stored in the system registry, plug-and-play device discovery, and creation of objects by system components.

The Windows XP executive allows any object to be given a **name**. One process may create a named object, while a second process opens a handle to the object and shares it with the first process. Processes can also share objects by duplicating handles between processes, in which case the objects need not be named.

A name can be either permanent or temporary. A permanent name represents an entity, such as a disk drive, that remains even if no process is accessing it. A temporary name exists only while a process holds a handle to the object.

Object names are structured like file path names in MS-DOS and UNIX. Name space directories are represented by a **directory object** that contains the names of all the objects in the directory. The object name space is extended by the addition of device objects representing volumes containing file systems.

Objects are manipulated by a set of virtual functions with implementations provided for each object type: `create()`, `open()`, `close()`, `delete()`, `query_name()`, `parse()`, and `security()`. The latter three objects need explanation:

`query_name()` is called when a thread has a reference to an object but wants to know the object's name.

`parse()` is used by the object manager to search for an object given the object's name.

`security()` is called to make security checks on all object operations, such as when a process opens or closes an object, makes changes to the security descriptor, or duplicates a handle for an object.

The `parse` procedure is used to extend the abstract name space to include files. The translation of a path name to a file object begins at the root of the abstract name space. Path-name components are separated by back characters (`\`) rather than the slashes (`/`) used in UNIX. Each component is looked up in the current parse directory of the name space. Internal nodes within the name space are either directories or symbolic links. If a leaf object is found and there are no path-name components remaining, the leaf object is returned. Otherwise, the leaf object's `parse` procedure is invoked with the remaining path name.

`Parse` procedures are only used with a small number of objects belonging to the Windows GUI, the configuration manager (registry), and—most notably—device objects representing file systems.

The `parse` procedure for the device object type allocates a file object and initiates an open or create I/O operation on the file system. If successful, the file object fields are filled in to describe the file.

In summary, the path name to a file is used to traverse the object-manager namespace, translating the original absolute path name into a (device object, relative path name) pair. This pair is then passed to the file system via the I/O manager, which fills in the file object. The file object itself has no name but is referred to by a handle.

UNIX file systems have **symbolic links** that permit multiple nicknames—or aliases—for the same file. The **symbolic-link object** implemented by the Windows XP object manager is used within the abstract name space, not to provide files aliases on a file system. Even so, symbolic links are very useful. They are used to organize the name space, similar to the organization of the `/devices` directory in UNIX. They are also used to map standard MS-DOS drive letters to drive names. Drive letters are symbolic links that can be remapped to suit the convenience of the user or administrator.

Drive letters are one place where the abstract name space in Windows XP is not global. Each logged-on user has his or her own set of drive letters so that users can avoid interfering with one another. In contrast, terminal server sessions share all processes within a session. `BaseNamedObjects` contain the named objects created by most applications.

Although the name space is not directly visible across a network, the object manager's `parse()` method is used to help access a named object on another system. When a process attempts to open an object that resides on a remote

computer, the object manager calls the parse method for the device object corresponding to a network redirector. This results in an I/O operation that accesses the file across the network.

Objects are instances of an **object type**. The object type specifies how instances are to be allocated, the definitions of the data fields, and the implementation of the standard set of virtual functions used for all objects. These functions implement operations such as mapping names to objects, closing and deleting, and applying security.

The object manager keeps track of two counts for each object. The pointer count is the number of distinct references made to an object. Protected-mode code that refers to objects must keep a reference on the object to ensure that the object is not deleted while in use. The handle count is the number of handle table entries referring to an object. Each handle is also reflected in the reference count.

When a handle for an object is closed, the object's close routine is called. In the case of file objects, this call causes the I/O manager to do a cleanup operation at the close of the last handle. The cleanup operation tells the file system that the file is no longer accessed by user mode so that sharing restrictions, range locks, and other states specific to the corresponding open routine can be removed.

Each handle close removes a reference from the pointer count, but internal system components may retain additional references. When the final reference is removed, the object's delete procedure is called. Again using file objects as an example, the delete procedure causes the I/O manager to send the file system a close operation on the file object. This causes the file system to deallocate any internal data structures that were allocated for the file object.

After the delete procedure for a temporary object completes, the object is deleted from memory. Objects can be made permanent (at least with respect to the current boot of the system) by asking the object manager to take an extra reference against the object. Thus, permanent objects are not deleted even when the last reference outside the object manager is removed. When a permanent object is made temporary again, the object manager removes the extra reference. If this was the last reference, the object is deleted. Permanent objects are rare, used mostly for devices, drive-letter mappings, and the directory and symbolic link objects.

The job of the object manager is to supervise the use of all managed objects. When a thread wants to use an object, it calls the object manager's `open()` method to get a reference to the object. If the object is being opened from a user-mode API, the reference is inserted into the process's object table, and a handle is returned.

A process gets a handle by creating an object, by opening an existing object, by receiving a duplicated handle from another process, or by inheriting a handle from a **parent process**, similar to the way a UNIX process gets a file descriptor. These handles are all stored in the process's **object table**. An entry in the object table contains the object's access rights and states whether the handle should be inherited by **child processes**. When a process terminates, Windows XP automatically closes all the process's open handles.

Handles are a standardized interface to all kinds of objects. Like a file descriptor in UNIX, an object handle is an identifier unique to a process that confers the ability to access and manipulate a system resource. Handles can be duplicated within a process or between processes. The latter case is used

when child processes are created and when out-of-process execution contexts are implemented.

Since the object manager is the only entity that generates object handles, it is the natural place to check security. The object manager checks whether a process has the right to access an object when the process tries to open the object. The object manager also enforces quotas, such as the maximum amount of memory a process may use, by charging a process for the memory occupied by all its referenced objects and refusing to allocate more memory when the accumulated charges exceed the process's quota.

When the login process authenticates a user, an access token is attached to the user's process. The access token contains information such as the security ID, group IDs, privileges, primary group, and default access-control list. The services and objects a user can access are determined by these attributes.

The token that controls access is associated with the thread making the access. Normally, the thread token is missing and defaults to the process token, but services often need to execute code on behalf of their client. Windows XP allows threads to impersonate temporarily by using a client's token. Thus, the thread token is not necessarily the same as the process token.

In Windows XP, each object is protected by an access-control list that contains the security IDs and access rights granted. When a thread attempts to access an object, the system compares the security ID in the thread's access token with the object's access-control list to determine whether access should be permitted. The check is performed only when an object is opened, so it is not possible to deny access after the open occurs. Operating-system components executing in kernel mode bypass the access check, since kernel-mode code is assumed to be trusted. Therefore, kernel-mode code must avoid security vulnerabilities, such as leaving checks disabled while creating a user-mode-accessible handle in an untrusted process.

Generally, the creator of the object determines the access-control list for the object. If none is explicitly supplied, one may be set to a default by the object type's open routine, or a default list may be obtained from the user's access-token object.

The access token has a field that controls auditing of object accesses. Operations that are being audited are logged to the system's security log with an identification of the user. An administrator monitors this log to discover attempts to break into the system or to access protected objects.

22.3.3.2 Virtual Memory Manager

The executive component that manages the virtual address space, physical memory allocation, and paging is the **virtual memory (VM) manager**. The design of the VM manager assumes that the underlying hardware supports virtual-to-physical mapping, a paging mechanism, and transparent cache coherence on multiprocessor systems, as well as allowing multiple page-table entries to map to the same physical page frame. The VM manager in Windows XP uses a page-based management scheme with a page size of 4 KB on IA32-compatible processors and 8 KB on the IA64. Pages of data allocated to a process that are not in physical memory are either stored in the **paging files** on disk or mapped directly to a regular file on a local or remote file system. Pages can also

be marked zero-fill-on-demand, which fills the page with zeros before being allocated, thus erasing the previous contents.

On IA32 processors, each process has a 4-GB virtual address space. The upper 2 GB are mostly identical for all processes and are used by Windows XP in kernel mode to access the operating-system code and data structures. Key areas of the kernel-mode region that are not identical for all processes are the **page-table self-map**, **hyperspace**, and **session space**. The hardware references a process's page tables using physical page-frame numbers. The VM manager maps the page tables into a single 4-MB region in the process's address space so they are accessed through virtual addresses. Hyperspace maps the current process's working-set information into the kernel-mode address space.

Session space is used to share the Win32 and other session-specific drivers among all the processes in the same terminal-server session rather than all the processes in the system. The lower 2 GB are specific to each process and are accessible by both user- and kernel-mode threads. Certain configurations of Windows XP reserve only 1 GB for operating-system use, allowing a process to use 3 GB of address space. Running the system in 3-GB mode drastically reduces the amount of data caching in the kernel. However, for large applications that manage their own I/O, such as SQL databases, the advantage of a larger user-mode address space may be worth the loss of caching.

The Windows XP VM manager uses a two-step process to allocate user memory. The first step *reserves* a portion of the process's virtual address space. The second step *commits* the allocation by assigning virtual memory space (physical memory or space in the paging files). Windows XP limits the amount of virtual memory space a process consumes by enforcing a quota on committed memory. A process decommits memory that it is no longer using to free up virtual memory for use by other processes. The APIs used to reserve virtual addresses and commit virtual memory take a handle on a process object as a parameter. This allows one process to control the virtual memory of another. Environmental subsystems manage the memory of their client processes in this way.

For performance, the VM manager allows a privileged process to lock selected pages in physical memory, thus ensuring that the pages are not paged out to the paging file. Processes also allocate raw physical memory and then map regions into its virtual address space. IA32 processors with the physical address extension (PAE) feature can have up to 64 GB of physical memory on a system. This memory cannot all be mapped in a process's address space at once, but Windows XP makes it available using the address windowing extension (AWE) APIs, which allocate physical memory and then map regions of virtual addresses in the process's address space onto part of the physical memory. The AWE facility is used primarily by very large applications such as the SQL database.

Windows XP implements shared memory by defining a **section object**. After getting a handle to a section object, a process maps the memory portion it needs into its address space. This portion is called a **view**. A process redefines its view of an object to gain access to the entire object, one region at a time.

A process can control the use of a shared-memory section object in many ways. The maximum size of a section can be bounded. The section can be backed by disk space either in the system-paging file or in a regular file (a **memory-mapped file**). A section can be *based*, meaning the section appears at

the same virtual address for all processes attempting to access it. Finally, the memory protection of pages in the section can be set to read-only, read-write, read-write-execute, execute-only, no access, or copy-on-write. The last two of these protection settings need some explanation:

A *no-access page* raises an exception if accessed; the exception is used, for example, to check whether a faulty program iterates beyond the end of an array. Both the user-mode memory allocator and the special kernel allocator used by the device verifier can be configured to map each allocation onto the end of a page followed by a no-access page in order to detect buffer overruns.

The *copy-on-write mechanism* increases the efficient use of physical memory by the VM manager. When two processes want independent copies of an object, the VM manager places a single shared copy into virtual memory and activates the copy-on-write property for that region of memory. If one of the processes tries to modify data in a copy-on-write page, the VM manager makes a private copy of the page for the process.

The virtual address translation in Windows XP uses a multilevel page table. For IA32 processors without the physical address extensions enabled, each process has a **page directory** that contains 1,024 **page-directory entries (PDEs)** of size 4 bytes. Each PDE points to a **page table** that contains 1,024 **page-table entries (PTEs)** of size 4 bytes. Each PTE points to a 4-KB **page frame** in physical memory. The total size of all page tables for a process is 4 MB, so the VM manager pages out individual tables to disk when necessary. See Figure 22.3 for a diagram of this structure.

The page directory and page tables are referenced by the hardware via physical addresses. To improve performance, the VM manager self-maps

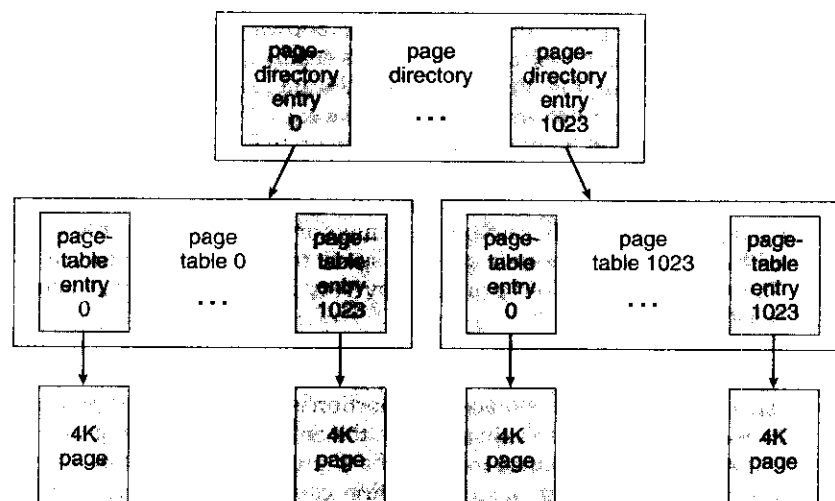


Figure 22.3 Page table layout.

the page directory and page tables into a 4-MB region of virtual addresses. The self-map allows the VM manager to translate a virtual address into the corresponding PDE or PTE without additional memory accesses. When a process context is changed, a single page-directory entry needs to be changed to map the new process's page tables. For a variety of reasons, the hardware requires that each page directory or page table occupy a single page. Thus, the number of PDEs or PTEs that fit in a page determine how virtual addresses are translated.

The following describes how virtual addresses are translated into physical addresses on IA32-compatible processors (without PAE enabled). A 10-bit value can represent all the values from 0 to 1,023. Thus, a 10-bit value can select any entry in the page directory or in a page table. This property is used when a virtual address pointer is translated to a byte address in physical memory. A 32-bit virtual-memory address is split into three values, as shown in Figure 22.4. The first 10 bits of the virtual address are used as an index into the page directory. This address selects one page-directory entry (PDE), which contains the physical page frame of a page table. The memory-management unit (MMU) uses the next 10 bits of the virtual address to select a PTE from the page table. The PTE specifies a page frame in physical memory. The remaining 12 bits of the virtual address are the offset of a specific byte in the page frame. The MMU creates a pointer to the specific byte in physical memory by concatenating the 20 bits from the PTE with the lower 12 bits from the virtual address. Thus, the 32-bit PTE has 12 bits to describe the state of the physical page. The IA32 hardware reserves 3 bits for use by the operating system. The rest of the bits specify whether the page has been accessed or written, the caching attributes, the access mode, whether the page is global, and whether the PTE is valid.

IA32 processors running with PAE enabled use 64-bit PDEs and PTEs in order to represent the larger 24-bit page-frame number field. Thus, the second-level page directories and the page tables contain only 512 PDEs and PTEs, respectively. To provide 4 GB of virtual address space requires an extra level of page directory containing four PDEs. Translation of a 32-bit virtual address uses 2 bits for the top-level directory index and 9 bits for each of the second-level page directories and the page tables.

To avoid the overhead of translating every virtual address by locking up the PDE and PTE, processors use a **translation-lookaside buffer (TLB)**, which contains an associative memory cache for mapping virtual pages to PTEs. Unlike the IA32 architecture, in which the TLB is maintained by the hardware MMU, the IA64 invokes a software-trap routine to supply translations missing from the TLB. This gives the VM manager flexibility in choosing the data structures to use. In Windows XP, a three-level tree structure is chosen for mapping user-mode virtual addresses on the IA64.

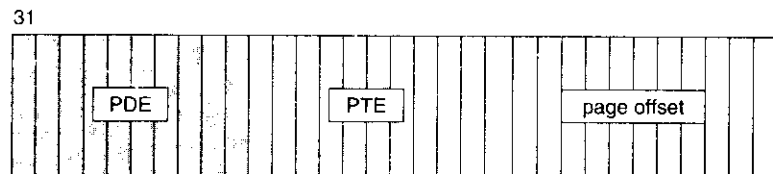


Figure 22.4 Virtual-to-physical address translation on IA32.

On IA64 processors, the page size is 8 KB, but the PTEs occupy 64 bits, so a page still contains only 1,024 (10 bits' worth) of PDEs or PTEs. Therefore, with 10 bits of top-level PDEs, 10 bits of second-level, 10 bits of page table, and 13 bits of page offset, the user portion of the process's virtual address space for Windows XP on the IA64 is 8 TB (43 bits' worth). The 8-TB limitation in the current version of Windows XP is less than the capabilities of the IA64 processor but represents a tradeoff between the number of memory references required to handle TLB misses and the size of the user-mode address space supported.

A physical page can be in one of six states: valid, free, zeroed, modified, standby, bad, or in transition.

- A *valid* page is in use by an active process.
- A *free* page is a page that is not referenced in a PTE.
- A *zeroed* page is a free page that has been zeroed out and is ready for immediate use to satisfy zero-on-demand faults.
- A *modified* page is one that has been written by a process and must be sent to the disk before it is allocated for another process.
- A *standby* page is a copy of information already stored on disk. Standby pages can be pages that were not modified, modified pages that have already been written to the disk, or pages that were prefetched to exploit locality.
- A *bad* page is unusable because a hardware error has been detected.
- Finally, a *transition* page is one that is on its way in from disk to a page frame allocated in physical memory.

When the valid bit in a PTE is zero, the VM manager defines the format of the other bits. Invalid pages can have a number of states represented by bits in the PTE. Page-file pages that have never been faulted in are marked zero-on-demand. Files mapped through section objects encode a pointer to that section object. Pages that have been written to the page file contain enough information to find the page on disk, and so forth.

The actual structure of the page-file PTE is shown in Figure 22.5. The PTE contains 5 bits for page protection, 20 bits for page-file offset, 4 bits to select the paging file, and 3 bits that describe the page state. A page-file PTE is marked to be an invalid virtual address to the MMU. Since executable code and memory-mapped files already have a copy on disk, they do not need space in a paging file. If one of these pages is not in physical memory, the PTE structure is as

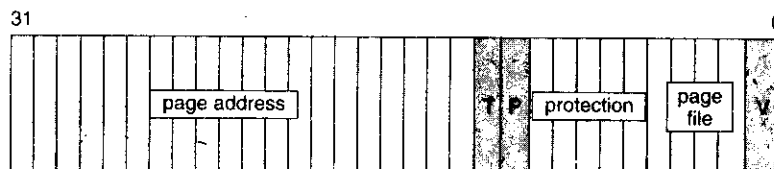


Figure 22.5 Page-file page-table entry. The valid bit is zero.

follows: The most significant bit is used to specify the page protection, the next 28 bits are used to index into a system data structure that indicates a file and offset within the file for the page, and the lower 3 bits specify the page state.

Invalid virtual addresses can also be in a number of temporary states that are part of the paging algorithms. When a page is removed from a process working set, it is moved either to the modified list (to be written to disk) or directly to the standby list. If written to the standby list, the page is reclaimed without being read from disk if it is needed again before it is moved to the free list. When possible, the VM manager uses idle CPU cycles to zero pages on the free list and move them to the zeroed list. Transition pages have been allocated a physical page and are awaiting the completion of the paging I/O before the PTE is marked as valid.

Windows XP uses section objects to describe pages that are sharable between processes. Each process has its own set of virtual page tables, but the section object also includes a set of page tables containing the master (or prototype) PTEs. When a PTE in a process page table is marked valid, it points to the physical page frame containing the page, as it must on IA32 processors, where the hardware MMU reads the page tables directly from memory. But when a shared page is made invalid, the PTE is edited to point to the prototype PTE associated with the section object.

The page tables associated with a section object are virtual insofar as they are created and trimmed as needed. The only prototype PTEs needed are those that describe pages for which there is a currently mapped view. This greatly improves performance and allows more efficient use of kernel virtual addresses.

The prototype PTE contains the page-frame address and the protection and state bits. Thus, the first access by a process to a shared page generates a page fault. After the first access, further accesses are performed in the normal manner. If a process writes to a copy-on-write page marked read-only in the PTE, the VM manager makes a copy of the page and marks the PTE writable, and the process effectively does not have a shared page any longer. Shared pages never appear in the page file but are instead found in the file system.

The VM manager keeps track of all pages of physical memory in a **page-frame database**. There is one entry for every page of physical memory in the system. The entry points to the PTE, which in turn points to the page frame, so the VM manager can maintain the state of the page. Page frames not referenced by a valid PTE are linked to lists according to page type, such as zeroed, modified, or free.

If a shared physical page is marked as valid for any process, the page cannot be removed from memory. The VM manager keeps a count of valid PTEs for each page in the page-frame database. When the count goes to zero, the physical page can be reused once its contents have been written back to disk (if it was marked dirty).

When a page fault occurs, the VM manager finds a physical page to hold the data. For zero-on-demand pages, the first choice is to find a page that has already been zeroed. If none is available, a page from the free list or standby list is chosen, and the page is zeroed before proceeding. If the faulted page has been marked as in transition, it is either already being read in from disk or has been unmapped or trimmed and is still available on the standby or

modified list. The thread either waits for the I/O to complete or, in the latter cases, reclaims the page from the appropriate list.

Otherwise, an I/O must be issued to read the page in from the paging file or file system. The VM manager tries to allocate an available page from either the free list or the standby list. Pages in the modified list cannot be used until they have been written back to disk and transferred to the standby list. If no pages are available, the thread blocks until the working-set manager trims pages from memory or a page in physical memory is unmapped by a process.

Windows XP uses a per-process first-in, first-out (FIFO) replacement policy to take pages from processes that are using more than their minimum working-set size. Windows XP monitors the page faulting of each process that is at its minimum working-set size and adjusts the working-set size accordingly. When a process is started, it is assigned a default minimum working-set size of 50 pages. The VM manager replaces and trims pages in the working set of a process according to their age. The age of a page is determined by how many trimming cycles have occurred without the PTE. Trimmed pages are moved to the standby or modified list, depending on whether the modified bit is set in the page's PTE.

The VM manager does not fault in only the page immediately needed. Research shows that the memory referencing of a thread tends to have a **locality** property; when a page is used, it is likely that adjacent pages will be referenced in the near future. (Think of iterating over an array or fetching sequential instructions that form the executable code for a thread.) Because of locality, when the VM manager faults in a page, it also faults in a few adjacent pages. This prefetching tends to reduce the total number of page faults. Writes are also clustered to reduce the number of independent I/O operations.

In addition to managing committed memory, the VM manager manages each process's reserved memory, or virtual address space. Each process has an associated splay tree that describes the ranges of virtual addresses in use and what the use is. This allows the VM manager to fault in page tables as needed. If the PTE for a faulting address does not exist, the VM manager searches for the address in the process's tree of **virtual address descriptors (VADs)** and uses this information to fill in the missing PTE and retrieve the page. In some cases, a page-table page itself may not exist; such a page must be transparently allocated and initialized by the VM manager.

22.3.3.3 Process Manager

The Windows XP process manager provides services for creating, deleting, and using processes, threads, and jobs. It has no knowledge about parent-child relationships or process hierarchies; those refinements are left to the particular environmental subsystem that owns the process. The process manager is also not involved in the scheduling of processes, other than setting the priorities and affinities in processes and threads when they are created. Thread scheduling takes place in the kernel dispatcher.

Each process contains one or more threads. Processes themselves can be collected together into large units called **job objects**; the use of job objects allows limits on CPU usage, working-set size, and processor affinities that control multiple processes at once. Job objects are used to manage large datacenter machines.

An example of process creation in the Win32 API environment is as follows. When a Win32 API application calls `CreateProcess()`:

A message is sent to the Win32 API subsystem to notify it that the process is being created.

`CreateProcess()` in the original process then calls an API in the process manager of the NT executive to actually create the process.

The process manager calls the object manager to create a process object and returns the object handle to Win32 API.

Win32 API calls the process manager again to create a thread for the process and returns handles to the new process and thread.

The Windows XP APIs for manipulating virtual memory and threads and for duplicating handles take a process handle, so subsystems can perform operations on behalf of a new process without having to execute directly in the new process's context. Once a new process is created, the initial thread is created, and an asynchronous procedure call is delivered to the thread to prompt the start of execution at the user-mode image loader. The loader is `ntdll.dll`, which is a link library automatically mapped into every newly created process. Windows XP also supports a UNIX `fork()` style of process creation in order to support the POSIX environmental subsystem. Although the Win32 API environment calls the process manager from the client process, POSIX uses the cross-process nature of the Windows XP APIs to create the new process from within the subsystem process.

The process manager also implements the queuing and delivery of asynchronous procedure calls (APCs) to threads. APCs are used by the system to initiate thread execution, complete I/O, terminate threads and processes, and attach debuggers. User-mode code can also queue an APC to a thread for delivery of signal-like notifications. To support POSIX, the process manager provides APIs that send alerts to threads to unblock them from system calls.

The debugger support in the process manager includes the capability to suspend and resume threads and to create threads that begin in a suspended mode. There are also process-manager APIs that get and set a thread's register context and access another process's virtual memory.

Threads can be created in the current process; they can also be injected into another process. Within the executive, existing threads can temporarily attach to another process. This method is used by worker threads that need to execute in the context of the process originating a work request.

The process manager also supports impersonation. A thread running in a process with a security token belonging to one user can set a thread-specific token belonging to another user. This facility is fundamental to the client-server computing model, where services need to act on behalf of a variety of clients with different security IDs.

22.3.3.4 Local Procedure Call Facility

The implementation of Windows XP uses a client-server model. The environmental subsystems are servers that implement particular operating-system personalities. The client-server model is used for implementing a variety

of operating-system services besides the environmental subsystems. Security management, printer spooling, web services, network file systems, plug-and-play, and many other features are implemented using this model. To reduce the memory footprint, multiple services are often collected together into a few processes, which then rely on the user-mode thread-pool facilities to share threads and wait for messages (see Section 22.3.3.3).

The operating system uses the local procedure call (LPC) facility to pass requests and results between client and server processes within a single machine. In particular, LPC is used to request services from the various Windows XP subsystems. LPC is similar in many respects to the RPC mechanisms used by many operating systems for distributed processing across networks, but LPC is optimized for use within a single system. The Windows XP implementation of Open Software Foundation (OSF) RPC often uses LPC as a transport on the local machine.

LPC is a message-passing mechanism. The server process publishes a globally visible connection-port object. When a client wants services from a subsystem, it opens a handle to the subsystem's connection-port object and sends a connection request to the port. The server creates a channel and returns a handle to the client. The channel consists of a pair of private communication ports: one for client-to-server messages and the other for server-to-client messages. Communication channels support a callback mechanism, so the client and server can accept requests when they would normally be expecting a reply.

When an LPC channel is created, one of three message-passing techniques must be specified.

1. The first technique is suitable for small messages (up to a couple of hundred bytes). In this case, the port's message queue is used as intermediate storage, and the messages are copied from one process to the other.
2. The second technique is for larger messages. In this case, a shared-memory section object is created for the channel. Messages sent through the port's message queue contain a pointer and size information referring to the section object. This avoids the need to copy large messages. The sender places data into the shared section, and the receiver views them directly.
3. The third technique uses the APIs that read and write directly into a process's address space. The LPC provides functions and synchronization so a server can access the data in a client.

The Win32 API window manager uses its own form of message passing that is independent of the executive LPC facilities. When a client asks for a connection that uses window-manager messaging, the server sets up three objects: (1) a dedicated server thread to handle requests, (2) a 64-KB section object, and (3) an event-pair object. An *event-pair object* is a synchronization object that is used by the Win32 API subsystem to provide notification when the client thread has copied a message to the Win32 API server, or vice versa. The section object passes the messages, and the event-pair object performs synchronization.

Window-manager messaging has several advantages:

The section object eliminates message copying, since it represents a region of shared memory.

The event-pair object eliminates the overhead of using the port object to pass messages containing pointers and lengths.

The dedicated server thread eliminates the overhead of determining which client thread is calling the server, since there is one server thread per client thread.

The kernel gives scheduling preference to these dedicated server threads to improve performance.

22.3.3.5 I/O Manager

The **I/O manager** is responsible for file systems, device drivers, and network drivers. It keeps track of which device drivers, filter drivers, and file systems are loaded, and it also manages buffers for I/O requests. It works with the VM manager to provide memory-mapped file I/O and controls the Windows XP cache manager, which handles caching for the entire I/O system. The I/O manager is fundamentally asynchronous. Synchronous I/O is provided by explicitly waiting for an I/O operation to complete. The I/O manager provides several models of asynchronous I/O completion, including setting of events, delivery of APCs to the initiating thread, and use of I/O completion ports, which allow a single thread to process I/O completions from many other threads.

Device drivers are arranged as a list for each device (called a driver or I/O stack because of how device drivers are added). The I/O manager converts the requests it receives into a standard form called an **I/O request packet (IRP)**. It then forwards the IRP to the first driver in the stack for processing. After each driver processes the IRP, it calls the I/O manager either to forward it to the next driver in the stack or, if all processing is finished, to complete the operation on the IRP.

Completions may occur in a different context from the original I/O request. For example, if a driver is performing its part of an I/O operation and is forced to block for an extended time, it may queue the IRP to a worker thread to continue processing in the system context. In the original thread, the driver returns a status indicating that the I/O request is pending so that the thread can continue executing in parallel with the I/O operation. IRPs may also be processed in interrupt-service routines and completed in an arbitrary context. Because some final processing may need to happen in the context that initiated the I/O, the I/O manager uses an APC to do final I/O-completion processing in the context of the originating thread.

The stack model is very flexible. As a driver stack is built, various drivers have the opportunity to insert themselves into the stack as **filter drivers**. Filter drivers can examine and potentially modify each I/O operation. Mount management, partition management, and disk striping and mirroring are all examples of functionality implemented using filter drivers that execute beneath the file system in the stack. File-system filter drivers execute above the file system and have been used to implement functionality such as hierarchical storage management, single instancing of files for remote boot, and dynamic

format conversion. Third parties also use file-system filter drivers to implement virus detection.

Device drivers for Windows XP are written to the Windows Driver Model (WDM) specification. This model lays out all the requirements for device drivers, including how to layer filter drivers, share common code for handling power and plug-and-play requests, build correct cancellation logic, and so forth.

Because of the richness of the WDM, writing a full WDM device driver for each new hardware device can involve an excessive amount of work. Fortunately, the port/miniport model makes it unnecessary to do this. Within a class of similar devices, such as audio drivers, SCSI devices, or Ethernet controllers, each instance of a device shares a common driver for that class, called a **port driver**. The port driver implements the standard operations for the class and then calls device-specific routines in the device's **miniport driver** to implement device-specific functionality.

22.3.3.6 Cache Manager

In many operating systems, caching is done by the file system. Instead, Windows XP provides a centralized caching facility. The **cache manager** works closely with the VM manager to provide cache services for all components under the control of the I/O manager. Caching in Windows XP is based on files rather than raw blocks.

The size of the cache changes dynamically according to how much free memory is available in the system. Recall that the upper 2 GB of a process's address space comprise the system area; it is available in the context of all processes. The VM manager allocates up to one-half of this space to the system cache. The cache manager maps files into this address space and uses the capabilities of the VM manager to handle file I/O.

The cache is divided into blocks of 256 KB. Each cache block can hold a view (that is, a memory-mapped region) of a file. Each cache block is described by a **virtual address control block (VACB)** that stores the virtual address and file offset for the view, as well as the number of processes using the view. The VACBs reside in a single array maintained by the cache manager.

For each open file, the cache manager maintains a separate VACB index array that describes the caching for the entire file. This array has an entry for each 256-KB chunk of the file; so, for instance, a 2-MB file would have an 8-entry VACB index array. An entry in the VACB index array points to the VACB if that portion of the file is in the cache; it is null otherwise. When the I/O manager receives a file's user-level read request, the I/O manager sends an IRP to the device-driver stack on which the file resides. The file system attempts to look up the requested data in the cache manager (unless the request specifically asks for a noncached read). The cache manager calculates which entry of that file's VACB index array corresponds to the byte offset of the request. The entry either points to the view in the cache or is invalid. If it is invalid, the cache manager allocates a cache block (and the corresponding entry in the VACB array) and maps the view into the cache block. The cache manager then attempts to copy data from the mapped file to the caller's buffer. If the copy succeeds, the operation is completed.

If the copy fails, it does so because of a page fault, which causes the VM manager to send a noncached read request to the I/O manager. The I/O manager

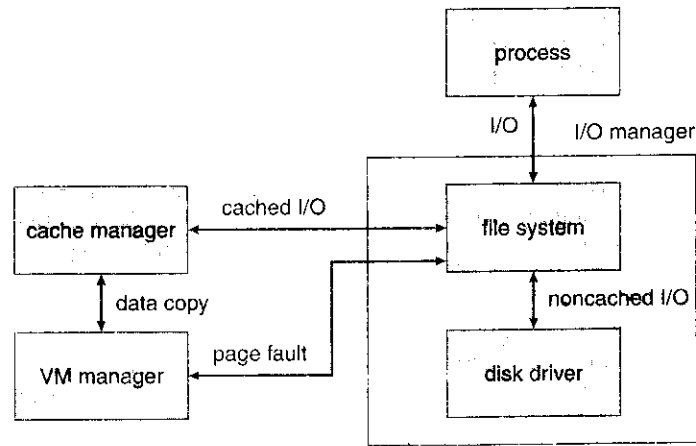


Figure 22.6 File I/O.

sends another request down the driver stack, this time requesting a *paging* operation, which bypasses the cache manager and reads the data from the file directly into the page allocated for the cache manager. Upon completion, the VACB is set to point at the page. The data, now in the cache, are copied to the caller's buffer, and the original I/O request is completed. Figure 22.6 shows an overview of these operations.

When possible, for synchronous operations on cached files, I/O is handled by the **fast I/O mechanism**. This mechanism parallels the normal IRP-based I/O but calls into the driver stack directly rather than passing down an IRP. Because no IRP is involved, the operation should not block for an extended period of time and cannot be queued to a worker thread. Therefore, when the operation reaches the file system and calls the cache manager, the operation fails if the information is not already in cache. The I/O manager then attempts the operation using the normal IRP path.

A kernel-level read operation is similar, except that the data can be accessed directly from the cache, rather than being copied to a buffer in user space. To use file-system metadata (data structures that describe the file system), the kernel uses the cache manager's mapping interface to read the metadata. To modify the metadata, the file system uses the cache manager's pinning interface. **Pinning** a page locks the page into a physical-memory page frame so that the VM manager cannot move or page out the page. After updating the metadata, the file system asks the cache manager to unpin the page. A modified page is marked dirty, and so the VM manager flushes the page to disk. The metadata is stored in a regular file.

To improve performance, the cache manager keeps a small history of read requests and from this history attempts to predict future requests. If the cache manager finds a pattern in the previous three requests, such as sequential access forward or backward, it prefetches data into the cache before the next request is submitted by the application. In this way, the application finds its data already cached and does not need to wait for disk I/O. The Win32 API `OpenFile()` and `CreateFile()` functions can be passed the `FILE_FLAG_SEQUENTIAL_SCAN` flag,

which is a hint to the cache manager to try to prefetch 192 KB ahead of the thread's requests. Typically, Windows XP performs I/O operations in chunks of 64 KB or 16 pages; thus, this read-ahead is three times the normal amount.

The cache manager is also responsible for telling the VM manager to flush the contents of the cache. The cache manager's default behavior is write-back caching: It accumulates writes for 4 to 5 seconds and then wakes up the cache-writer thread. When write-through caching is needed, a process can set a flag when opening the file, or the process can call an explicit cache-flush function.

A fast-writing process could potentially fill all the free cache pages before the cache-writer thread had a chance to wake up and flush the pages to disk. The cache writer prevents a process from flooding the system in the following way. When the amount of free cache memory becomes low, the cache manager temporarily blocks processes attempting to write data and wakes the cache-writer thread to flush pages to disk. If the fast-writing process is actually a network redirector for a network file system, blocking it for too long could cause network transfers to time out and be retransmitted. This retransmission would waste network bandwidth. To prevent such waste, network redirectors can instruct the cache manager to limit the backlog of writes in the cache.

Because a network file system needs to move data between a disk and the network interface, the cache manager also provides a DMA interface to move the data directly. Moving data directly avoids the need to copy data through an intermediate buffer.

22.3.3.7 Security Reference Monitor

Centralizing management of system entities in the object manager enables Windows XP to use a uniform mechanism to perform run-time access validation and audit checks for every user-accessible entity in the system. Whenever a process opens a handle to an object, the **security reference monitor (SRM)** checks the process's security token and the object's access-control list to see whether the process has the necessary rights.

The SRM is also responsible for manipulating the privileges in security tokens. Special privileges are required for users to perform backup or restore operations on file systems, overcome certain checks as an administrator, debug processes, and so forth. Tokens can also be marked as being restricted in their privileges so that they cannot access objects that are available to most users. Restricted tokens are primarily used to restrict the damage that can be done by execution of untrusted code.

Another responsibility of the SRM is logging security audit events. A C-2 security rating requires that the system have the ability to detect and log all attempts to access system resources so that it is easier to trace attempts at unauthorized access. Because the SRM is responsible for making access checks, it generates most of the audit records in the security-event log.

22.3.3.8 Plug-and-Play and Power Managers

The operating system uses the **plug-and-play (PnP) manager** to recognize and adapt to changes in the hardware configuration. For PnP to work, both the device and the driver must support the PnP standard. The PnP manager automatically recognizes installed devices and detects changes in devices as the system operates. The manager also keeps track of resources used by a device,

as well as potential resources that could be used, and takes care of loading the appropriate drivers. This management of hardware resources—primarily interrupts and I/O memory ranges—has the goal of determining a hardware configuration in which all devices are able to operate.

For example, if device B can use interrupt 5 and device A can use 5 or 7, then the PnP manager will assign 5 to B and 7 to A. In previous versions, the user might have had to remove device A and reconfigure it to use interrupt 7 before installing device B. The user thus had to study system resources before installing new hardware and had to determine which devices were using which hardware resources. The proliferation of PCMCIA cards, laptop docks, and USB, IEEE 1394, Infiniband, and other hot-pluggable devices also dictates the support of dynamically configurable resources.

The PnP manager handles dynamic reconfiguration as follows. First, it gets a list of devices from each bus driver (for example, PCI, USB). It loads the installed driver (or installs one, if necessary) and sends an `add-device` request to the appropriate driver for each device. The PnP manager figures out the optimal resource assignments and sends a `start-device` request to each driver, along with the resource assignment for the device. If a device needs to be reconfigured, the PnP manager sends a `query-stop` request, which asks the driver whether the device can be temporarily disabled. If the driver can disable the device, then all pending operations are completed, and new operations are prevented from starting. Next, the PnP manager sends a `stop` request; it can then reconfigure the device with another `start-device` request.

The PnP manager also supports other requests, such as `query-remove`. This request, which is used when the user is getting ready to eject a PCCARD device, operates in a fashion similar to `query-stop`. The `surprise-remove` request is used when a device fails or, more likely, when a user removes a PCCARD device without stopping it first. The `remove` request tells the driver to stop using the device and release all resources allocated to it.

Windows XP supports sophisticated power management. Although these facilities are useful for home systems to reduce power consumption, their primary application is for ease of use (quicker access) and extending the battery life of laptops. The system and individual devices can be moved to low-power mode (called standby or sleep mode) when not in use, so the battery is primarily directed at physical memory (RAM) data retention. The system can turn itself back on when packets are received from the network, a phone line to a modem rings, or a user opens a laptop or pushes a soft power button. Windows XP can also *hibernate* a system by storing physical memory contents to disk and completely shutting down the machine, then restoring the system at a later point before execution continues.

Further strategies for reducing power consumption are supported as well. Rather than allowing it to spin in a processor loop when the CPU is idle, Windows XP moves the system to a state requiring lower power consumption. If the CPU is underutilized, Windows XP reduces the CPU clock speed, which can save significant power.

22.3.3.9 Registry

Windows XP keeps much of its configuration information in an internal database called the **registry**. A registry database is called a **hive**. There are

separate hives for system information, default user preferences, software installation, and security. Because the information in the **system hive** is required in order to boot the system, the registry manager is implemented as a component of the executive.

Every time the system successfully boots, it saves the system hive as *last known good*. If the user installs software, such as a device driver, that produces a system-hive configuration that will not boot, the user can usually boot using the last-known-good configuration.

Damage to the system hive from installing third-party applications and drivers is so common that Windows XP has a component called **system restore** that periodically saves the hives, as well as other software states like driver executables and configuration files, so that the system can be restored to a previously working state in cases where the system boots but no longer operates as expected.

22.3.3.10 Booting

The booting of a Windows XP PC begins when the hardware powers on and the BIOS begins executing from ROM. The BIOS identifies the **system device** to be booted and loads and executes the bootstrap loader from the front of the disk. This loader knows enough about the file-system format to load the NTLDR program from the root directory of the system device. NTLDR is used to determine which **boot device** contains the operating system. Next, the NTLDR loads in the HAL library, the kernel, and the system hive from the boot device. From the system hive, it determines what device drivers are needed to boot the system (the *boot drivers*) and loads them. Finally, NTLDR begins kernel execution.

The kernel initializes the system and creates two processes. The **system process** contains all the internal worker threads and never executes in user mode. The first user-mode process created is SMSS, which is similar to the INIT (initialization) process in UNIX. SMSS does further initialization of the system, including establishing the paging files and loading device drivers, and creates the WINLOGON and CSRSS processes. CSRSS is the Win32 API subsystem. WINLOGON brings up the rest of the system, including the LSASS security subsystem and the remaining services needed to run the system.

The system optimizes the boot process by pre-loading files from disk based on previous boots of the system. Disk access patterns at boot are also used to lay out system files on disk to reduce the number of I/O operations required. The processes required to start the system are reduced by grouping services into one process. All of these approaches contribute to a dramatic reduction in system boot time. Of course, system boot time is less important than it once was because of the sleep and hibernation capabilities of Windows XP, which allow users to power down their computers and then quickly resume where they left off.

22.4

Environmental subsystems are user-mode processes layered over the native Windows XP executive services to enable Windows XP to run programs

developed for other operating systems, including 16-bit Windows, MS-DOS, and POSIX. Each environmental subsystem provides a single application environment.

Windows XP uses the Win32 API subsystem as the main operating environment, and thus this subsystem starts all processes. When an application is executed, the Win32 API subsystem calls the VM manager to load the application's executable code. The memory manager returns a status to Win32 indicating the type of executable. If it is not a native Win32 API executable, the Win32 API environment checks whether the appropriate environmental subsystem is running; if the subsystem is not running, it is started as a user-mode process. The subsystem then takes control over the application startup.

The environmental subsystems use the LPC facility to provide operating-system services to client processes. The Windows XP subsystem architecture keeps applications from mixing API routines from different environments. For instance, a Win32 API application cannot make a POSIX system call, because only one environmental subsystem can be associated with each process.

Since each subsystem is run as a separate user-mode process, a crash in one has no effect on other processes. The exception is Win32 API, which provides all keyboard, mouse, and graphical display capabilities. If it fails, the system is effectively disabled and requires a reboot.

The Win32 API environment categorizes applications as either graphical or character based, where a *character-based application* is one that thinks interactive output goes to a character-based (command) window. Win32 API transforms the output of a character-based application to a graphical representation in the command window. This transformation is easy: Whenever an output routine is called, the environmental subsystem calls a Win32 routine to display the text. Since the Win32 API environment performs this function for all character-based windows, it can transfer screen text between windows via the clipboard. This transformation works for MS-DOS applications, as well as for POSIX command-line applications.

22.4.1 MS-DOS Environment

The MS-DOS environment does not have the complexity of the other Windows XP environmental subsystems. It is provided by a Win32 API application called the **virtual DOS machine (VDM)**. Since the VDM is a user-mode process, it is paged and dispatched like any other Windows XP application. The VDM has an **instruction-execution unit** to execute or emulate Intel 486 instructions. The VDM also provides routines to emulate the MS-DOS ROM BIOS and "int 21" software-interrupt services and has virtual device drivers for the screen, keyboard, and communication ports. The VDM is based on MS-DOS 5.0 source code; it allocates at least 620 KB of memory to the application.

The Windows XP command shell is a program that creates a window that looks like an MS-DOS environment. It can run both 16-bit and 32-bit executables. When an MS-DOS application is run, the command shell starts a VDM process to execute the program.

If Windows XP is running on a IA32-compatible processor, MS-DOS graphical applications run in full-screen mode, and character applications can run full screen or in a window. Not all MS-DOS applications run under the VDM. For example, some MS-DOS applications access the disk hardware directly, so they

fail to run on Windows XP because disk access is restricted to protect the file system. In general, MS-DOS applications that directly access hardware will fail to operate under Windows XP.

Since MS-DOS is not a multitasking environment, some applications have been written in such a way as to “hog” the CPU. For instance, the use of busy loops can cause time delays or pauses in execution. The scheduler in the kernel dispatcher detects such delays and automatically throttles the CPU usage, but this may cause the offending application to operate incorrectly.

22.4.2 16-Bit Windows Environment

The Win16 execution environment is provided by a VDM that incorporates additional software called *Windows on Windows* (WOW32 for 16-bit applications); this software provides the Windows 3.1 kernel routines and stub routines for window-manager and graphical-device-interface (GDI) functions. The stub routines call the appropriate Win32 API subroutines—converting, or *thunking*, 16-bit addresses into 32-bit addresses. Applications that rely on the internal structure of the 16-bit window manager or GDI may not work, because the underlying Win32 API implementation is, of course, different from true 16-bit Windows.

WOW32 can multitask with other processes on Windows XP, but it resembles Windows 3.1 in many ways. Only one Win16 application can run at a time, all applications are single threaded and reside in the same address space, and all share the same input queue. These features imply that an application that stops receiving input will block all the other Win16 applications, just as in Windows 3.x, and one Win16 application can crash other Win16 applications by corrupting the address space. Multiple Win16 environments can coexist, however, by using the command *start /separate win16application* from the command line.

There are relatively few 16-bit applications that users need to continue to run on Windows XP, but some of them include common installation (setup) programs. Thus, the WOW32 environment continues to exist primarily because a number of 32-bit applications cannot be installed on Windows XP without it.

22.4.3 32-Bit Windows Environment on IA64

The native environment for Windows on IA64 uses 64-bit addresses and the native IA64 instruction set. To execute IA32 programs in this environment requires a thunking layer to translate 32-bit Win32 API calls into the corresponding 64-bit calls—just as 16-bit applications require translation on IA32 systems. Thus, 64-bit Windows supports the WOW64 environment. The implementations of 32-bit and 64-bit Windows are essentially identical, and the IA64 processor provides direct execution of IA32 instructions, so WOW64 achieves a higher level of compatibility than WOW32.

22.4.4 Win32 Environment

The main subsystem in Windows XP is the Win32 API. It runs Win32 API applications and manages all keyboard, mouse, and screen I/O. Since it is the controlling environment, it is designed to be extremely robust. Several features of the Win32 API contribute to this robustness. Unlike processes in the

Win16 environment, each Win32 process has its own input queue. The window manager dispatches all input on the system to the appropriate process's input queue, so a failed process does not block input to other processes.

The Windows XP kernel also provides preemptive multitasking, which enables the user to terminate applications that have failed or are no longer needed. The Win32 API also validates all objects before using them, to prevent crashes that could otherwise occur if an application tried to use an invalid or wrong handle. The Win32 API subsystem verifies the type of the object to which a handle points before using the object. The reference counts kept by the object manager prevent objects from being deleted while they are still being used and prevent their use after they have been deleted.

To achieve a high level of compatibility with Windows 95/98 systems, Windows XP allows users to specify that individual applications be run using a **shim layer**, which modifies the Win32 API to better approximate the behavior expected by old applications. For example, some applications expect to see a particular version of the system and fail on new versions. Frequently, applications have latent bugs that become exposed due to changes in the implementation. For example, using memory after freeing it may cause corruption only if the order of memory reuse by the heap changes; or an application may make assumptions about which errors can be returned by a routine or about the number of valid bits in an address. Running an application with the Windows 95/98 shims enabled causes the system to provide behavior much closer to Windows 95/98—though with reduced performance and limited interoperability with other applications.

22.4.5 POSIX Subsystem

The POSIX subsystem is designed to run POSIX applications written to follow the POSIX standard, which is based on the UNIX model. POSIX applications can be started by the Win32 API subsystem or by another POSIX application. POSIX applications use the POSIX subsystem server PSXSS.EXE, the POSIX dynamic link library PSXDLL.DLL, and the POSIX console session manager POSIX.EXE.

Although the POSIX standard does not specify printing, POSIX applications can use printers transparently via the Windows XP redirection mechanism. POSIX applications have access to any file system on the Windows XP system; the POSIX environment enforces UNIX-like permissions on directory trees.

Due to scheduling issues, the POSIX system in Windows XP does not ship with the system but is available separately for professional desktop systems and servers. It provides a much higher level of compatibility with UNIX applications than previous versions of NT. Of the commonly available UNIX applications, most compile and run without change with the latest version of Interix.

22.4.6 Logon and Security Subsystems

Before a user can access objects on Windows XP, that user must be authenticated by the logon service, WINLOGON. WINLOGON is responsible for responding to the secure attention sequence (Control-Alt-Delete). The secure attention sequence is a required mechanism for keeping an application from acting as a Trojan horse. Only WINLOGON can intercept this sequence in order to put up a logon screen, change passwords, and lock the workstation. To be

authenticated, a user must have an account and provide the password for that account. Alternatively, a user logs on by using a smart card and personal identification number, subject to the security policies in effect for the domain.

The local security authority subsystem (LSASS) is the process that generates access tokens to represent users on the system. It calls an **authentication package** to perform authentication using information from the logon subsystem or network server. Typically, the authentication package simply looks up the account information in a local database and checks to see that the password is correct. The security subsystem then generates the access token for the user ID containing the appropriate privileges, quota limits, and group IDs. Whenever the user attempts to access an object in the system, such as by opening a handle to the object, the access token is passed to the security reference monitor, which checks privileges and quotas. The default authentication package for Windows XP domains is Kerberos. LSASS also has the responsibility for implementing security policy such as strong passwords, for authenticating users, and for performing encryption of data and keys.

22.5 File Systems

Historically, MS-DOS systems have used the file-allocation table (FAT) file system. The 16-bit FAT file system has several shortcomings, including internal fragmentation, a size limitation of 2 GB, and a lack of access protection for files. The 32-bit FAT file system has solved the size and fragmentation problems, but its performance and features are still weak by comparison with modern file systems. The NTFS file system is much better. It was designed to include many features, including data recovery, security, fault tolerance, large files and file systems, multiple data streams, UNICODE names, sparse files, encryption, journaling, volume shadow copies, and file compression.

Windows XP uses NTFS as its basic file system, and we focus on it here. Windows XP continues to use FAT16, however, to read floppies and other removable media. And despite the advantages of NTFS, FAT32 continues to be important for interoperability of media with Windows 95/98 systems. Windows XP supports additional file-system types for the common formats used for CD and DVD media.

22.5.1 NTFS Internal Layout

The fundamental entity in NTFS is a volume. A volume is created by the Windows XP logical-disk-management utility and is based on a logical disk partition. A volume may occupy a portion of a disk, may occupy an entire disk, or may span several disks.

NTFS does not deal with individual sectors of a disk but instead uses clusters as the units of disk allocation. A **cluster** is a number of disk sectors that is a power of 2. The cluster size is configured when an NTFS file system is formatted. The default cluster size is the sector size for volumes up to 512 MB, 1 KB for volumes up to 1 GB, 2 KB for volumes up to 2 GB, and 4 KB for larger volumes. This cluster size is much smaller than that for the 16-bit FAT file system, and the small size reduces the amount of internal fragmentation. As an example, consider a 1.6-GB disk with 16,000 files. If you use a FAT-16 file system, 400 MB

may be lost to internal fragmentation because the cluster size is 32 KB. Under NTFS, only 17 MB would be lost when storing the same files.

NTFS uses **logical cluster numbers (LCNs)** as disk addresses. It assigns them by numbering clusters from the beginning of the disk to the end. Using this scheme, the system can calculate a physical disk offset (in bytes) by multiplying the LCN by the cluster size.

A file in NTFS is not a simple byte stream as it is in MS-DOS or UNIX; rather, it is a structured object consisting of typed **attributes**. Each attribute of a file is an independent byte stream that can be created, deleted, read, and written. Some attribute types are standard for all files, including the file name (or names, if the file has aliases, such as an MS-DOS shortname), the creation time, and the security descriptor that specifies access control. User data is stored in *data attributes*.

Most traditional data files have an *unnamed* data attribute that contains all the file's data. However, additional data streams can be created with explicit names. For instance, in Macintosh files stored on a Windows XP server, the resource fork is a named data stream. The IProp interfaces of the Component Object Model (COM) use a named data stream to store properties on ordinary files, including thumbnails of images. In general, attributes may be added as necessary and are accessed using a *file-name:attribute* syntax. NTFS returns the size of the unnamed attribute only in response to file-query operations, such as when running the `dir` command.

Every file in NTFS is described by one or more records in an array stored in a special file called the master file table (MFT). The size of a record is determined when the file system is created; it ranges from 1 to 4 KB. Small attributes are stored in the MFT record itself and are called **resident attributes**. Large attributes, such as the unnamed bulk data, are called **nonresident attributes** and are stored in one or more contiguous **extents** on the disk; a pointer to each extent is stored in the MFT record. For a small file, even the data attribute may fit inside the MFT record. If a file has many attributes—or if it is highly fragmented, so that many pointers are needed to point to all the fragments—one record in the MFT might not be large enough. In this case, the file is described by a record called the **base file record**, which contains pointers to overflow records that hold the additional pointers and attributes.

Each file in an NTFS volume has a unique ID called a **file reference**. The file reference is a 64-bit quantity that consists of a 48-bit file number and a 16-bit sequence number. The file number is the record number (that is, the array slot) in the MFT that describes the file. The sequence number is incremented every time an MFT entry is reused. The sequence number enables NTFS to perform internal consistency checks, such as catching a stale reference to a deleted file after the MFT entry has been reused for a new file.

22.5.1.1 NTFS B+ Tree

As in MS-DOS and UNIX, the NTFS namespace is organized as a hierarchy of directories. Each directory uses a data structure called a **B+ tree** to store an index of the file names in that directory. A B+ tree is used because it eliminates the cost of reorganizing the tree and has the property that the length of every path from the root of the tree to a leaf is the same. The **index root** of a directory contains the top level of the B+ tree. For a large directory, this top level contains

pointers to disk extents that hold the remainder of the tree. Each entry in the directory contains the name and file reference of the file, as well as a copy of the update timestamp and file size taken from the file's resident attributes in the MFT. Copies of this information are stored in the directory, so a directory listing can be efficiently generated. Because all the file names, sizes, and update times are available from the directory itself, there is no need to gather these attributes from the MFT entries for each of the files.

22.5.1.2 NTFS Metadata

The NTFS volume's metadata are all stored in files. The first file is the MFT. The second file, which is used during recovery if the MFT is damaged, contains a copy of the first 16 entries of the MFT. The next few files are also special in purpose. They include the log file, volume file, attribute-definition table, root directory, bitmap file, boot file, and bad-cluster file. We describe the role of each of these files below.

- The **log file** records all metadata updates to the file system.

- The **volume file** contains the name of the volume, the version of NTFS that formatted the volume, and a bit that tells whether the volume may have been corrupted and needs to be checked for consistency.

- The **attribute-definition table** indicates which attribute types are used in the volume and what operations can be performed on each of them.

- The **root directory** is the top-level directory in the file-system hierarchy.

- The **bitmap file** indicates which clusters on a volume are allocated to files and which are free.

- The **boot file** contains the startup code for Windows XP and must be located at a particular disk address so that it can be found easily by a simple ROM bootstrap loader. The boot file also contains the physical address of the MFT.

- The **bad-cluster file** keeps track of any bad areas on the volume; NTFS uses this record for error recovery.

22.5.2 Recovery

In many simple file systems, a power failure at the wrong time can damage the file-system data structures so severely that the entire volume is scrambled. Many versions of UNIX store redundant metadata on the disk, and they recover from crashes using the `fsck` program to check all the file-system data structures and restore them forcibly to a consistent state. Restoring them often involves deleting damaged files and freeing data clusters that had been written with user data but not properly recorded in the file system's metadata structures. This checking can be a slow process and can cause the loss of significant amounts of data.

NTFS takes a different approach to file-system robustness. In NTFS, all file-system data-structure updates are performed inside transactions. Before a data structure is altered, the transaction writes a log record that contains redo and

undo information; after the data structure has been changed, the transaction writes a commit record to the log to signify that the transaction succeeded.

After a crash, the system can restore the file-system data structures to a consistent state by processing the log records, first redoing the operations for committed transactions and then undoing the operations for transactions that did not commit successfully before the crash. Periodically (usually every 5 seconds), a checkpoint record is written to the log. The system does not need log records prior to the checkpoint to recover from a crash. They can be discarded, so the log file does not grow without bounds. The first time after system startup that an NTFS volume is accessed, NTFS automatically performs file-system recovery.

This scheme does not guarantee that all the user-file contents are correct after a crash; it ensures only that the file-system data structures (the metadata files) are undamaged and reflect some consistent state that existed prior to the crash. It would be possible to extend the transaction scheme to cover user files, and Microsoft may do so in the future.

The log is stored in the third metadata file at the beginning of the volume. It is created with a fixed maximum size when the file system is formatted. It has two sections: the **logging area**, which is a circular queue of log records, and the **restart area**, which holds context information, such as the position in the logging area where NTFS should start reading during a recovery. In fact, the restart area holds two copies of its information, so recovery is still possible if one copy is damaged during the crash.

The logging functionality is provided by the Windows XP **log-file service**. In addition to writing the log records and performing recovery actions, the log-file service keeps track of the free space in the log file. If the free space gets too low, the log-file service queues pending transactions, and NTFS halts all new I/O operations. After the in-progress operations complete, NTFS calls the cache manager to flush all data, then resets the log file and performs the queued transactions.

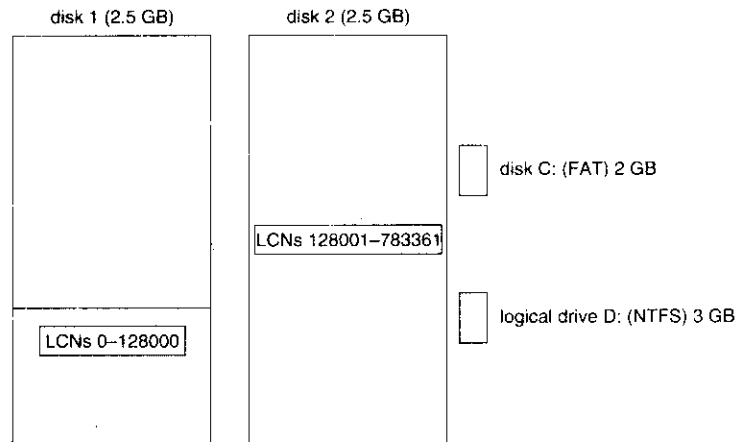


Figure 22.7 Volume set on two drives.

22.5.3 Security

The security of an NTFS volume is derived from the Windows XP object model. Each NTFS file references a security descriptor, which contains the access token of the owner of the file, and an access-control list, which states the access privileges granted to each user having access to the file.

In normal operation, NTFS does not enforce permissions on traversal of directories in file path names. However, for compatibility with POSIX, these checks can be enabled. Traversal checks are inherently more expensive, since modern parsing of file path names uses prefix matching rather than component-by-component opening of directory names.

22.5.4 Volume Management and Fault Tolerance

FtDisk is the fault-tolerant disk driver for Windows XP. When installed, it provides several ways to combine multiple disk drives into one logical volume so as to improve performance, capacity, or reliability.

22.5.4.1 Volume Set

One way to combine multiple disks is to concatenate them logically to form a large logical volume, as shown in Figure 22.7. In Windows XP, this logical volume, called a **volume set**, can consist of up to 32 physical partitions. A volume set that contains an NTFS volume can be extended without disturbance of the data already stored in the file system. The bitmap metadata on the NTFS volume are simply extended to cover the newly added space. NTFS continues to use the same LCN mechanism that it uses for a single physical disk, and the FtDisk driver supplies the mapping from a logical-volume offset to the offset on one particular disk.

22.5.4.2 Stripe Set

Another way to combine multiple physical partitions is to interleave their blocks in round-robin fashion to form what is called a **stripe set**, as shown in Figure 22.8. This scheme is also called RAID level 0, or **disk striping**. FtDisk

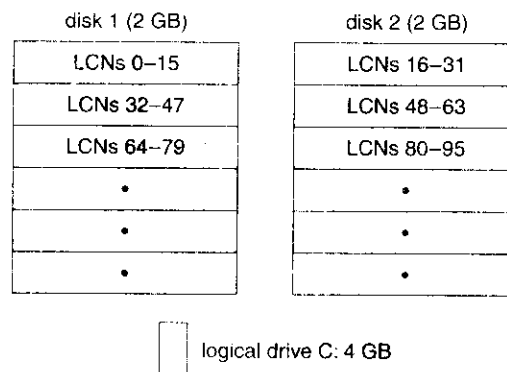


Figure 22.8 Stripe set on two drives.

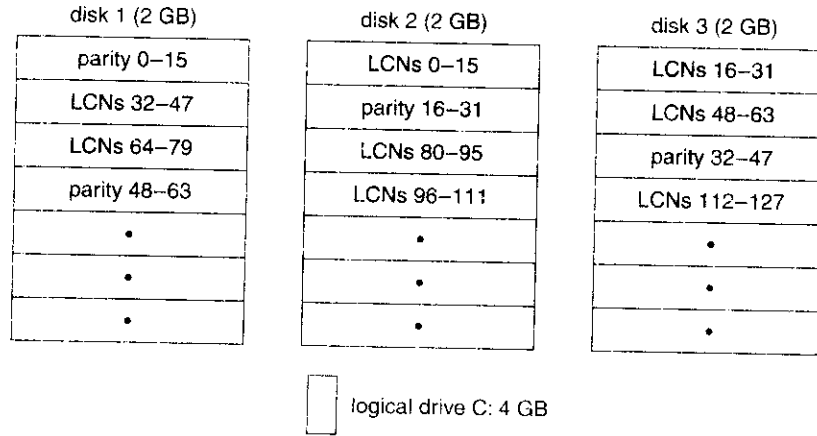


Figure 22.9 Stripe set with parity on three drives.

uses a stripe size of 64 KB: The first 64 KB of the logical volume are stored in the first physical partition, the second 64 KB in the second physical partition, and so on, until each partition has contributed 64 KB of space. Then, the allocation wraps around to the first disk, allocating the second 64-KB block. A stripe set forms one large logical volume, but the physical layout can improve the I/O bandwidth, because, for a large I/O, all the disks can transfer data in parallel.

22.5.4.3 Stripe Set with Parity

A variation of this idea is the **stripe set with parity**, which is shown in Figure 22.9. This scheme is also called RAID level 5. Suppose that a stripe set has eight disks. Seven of the disks will store data stripes, with one data stripe on each disk, and the eighth disk will store a parity stripe for each data stripe. The parity stripe contains the byte-wise exclusive or of the data stripes. If any one of the eight stripes is destroyed, the system can reconstruct the data by calculating the exclusive or of the remaining seven. This ability to reconstruct data makes the disk array much less likely to lose data in case of a disk failure.

Notice that an update to one data stripe also requires recalculation of the parity stripe. Seven concurrent writes to seven different data stripes thus would also require updates to seven parity stripes. If the parity stripes were all on the same disk, that disk could have seven times the I/O load of the data disks. To avoid creating this bottleneck, we spread the parity stripes over all the disks by assigning them in round-robin style. To build a stripe set with parity, we need a minimum of three equal-sized partitions located on three separate disks.

22.5.4.4 Disk Mirroring

An even more robust scheme is called **disk mirroring** or RAID level 1; it is depicted in Figure 22.10. A **mirror set** comprises two equal-sized partitions on two disks. When an application writes data to a mirror set, Ftdisk writes the data to both partitions, so that the data contents of the two partitions are identical. If one partition fails, Ftdisk has another copy safely stored on the

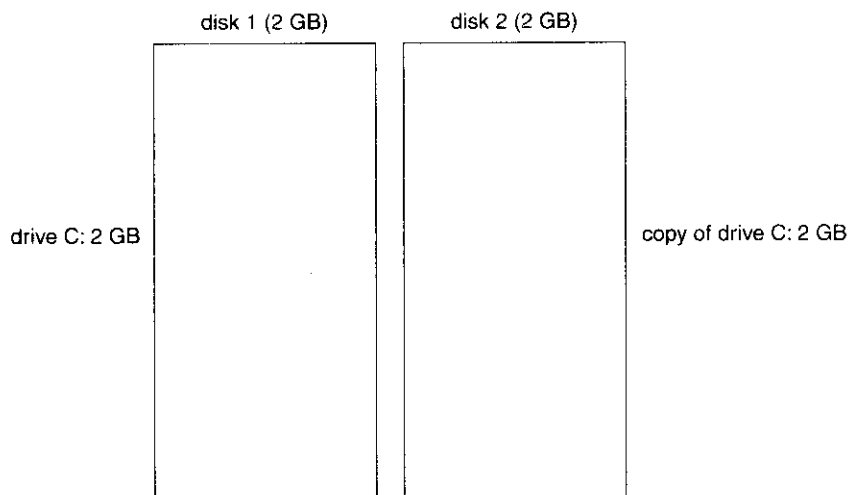


Figure 22.10 Mirror set on two drives.

mirror. Mirror sets can also improve performance, because read requests can be split between the two mirrors, giving each mirror half of the workload. To protect against the failure of a disk controller, we can attach the two disks of a mirror set to two separate disk controllers. This arrangement is called a **duplex set**.

22.5.4.5 Sector Sparring and Cluster Remapping

To deal with disk sectors that go bad, FtDisk uses a hardware technique called **sector sparring**, and NTFS uses a software technique called **cluster remapping**. **Sector sparring** is a hardware capability provided by many disk drives. When a disk drive is formatted, it creates a map from logical block numbers to good sectors on the disk. It also leaves extra sectors unmapped, as spares. If a sector fails, FtDisk instructs the disk drive to substitute a spare. **Cluster remapping** is a software technique performed by the file system. If a disk block goes bad, NTFS substitutes a different, unallocated block by changing any affected pointers in the MFT. NTFS also makes a note that the bad block should never be allocated to any file.

When a disk block goes bad, the usual outcome is a data loss. But sector sparring or cluster remapping can be combined with fault-tolerant volumes to mask the failure of a disk block. If a read fails, the system reconstructs the missing data by reading the mirror or by calculating the **exclusive** or parity in a stripe set with parity. The reconstructed data are stored into a new location that is obtained by sector sparring or cluster remapping.

22.5.5 Compression and Encryption

NTFS can perform data compression on individual files or on all data files in a directory. To compress a file, NTFS divides the file's data into **compression units**, which are blocks of 16 contiguous clusters. When each compression

unit is written, a data-compression algorithm is applied. If the result fits into fewer than 16 clusters, the compressed version is stored. When reading, NTFS can determine whether data have been compressed: If they have been, the length of the stored compression unit is less than 16 clusters. To improve performance when reading contiguous compression units, NTFS prefetches and decompresses ahead of the application requests.

For sparse files or files that contain mostly zeros, NTFS uses another technique to save space. Clusters that contain only zeros because they have never been written are not actually allocated or stored on disk. Instead, gaps are left in the sequence of virtual-cluster numbers stored in the MFT entry for the file. When reading a file, if it finds a gap in the virtual-cluster numbers, NTFS just zero-fills that portion of the caller's buffer. This technique is also used by UNIX.

NTFS supports encryption of files. Individual files or entire directories can be specified for encryption. The security system manages the keys used, and a key-recovery service is available to retrieve lost keys.

22.5.6 Mount Points

Mount points are a form of symbolic link specific to directories on NTFS. They provide a mechanism for administrators to organize disk volumes that is more flexible than the use of global names (like drive letters). Mount points are implemented as a symbolic link with associated data that contain the true volume name. Ultimately, mount points will supplant drive letters completely, but there will be a long transition due to the dependence of many applications on the drive-letter scheme.

22.5.7 Change Journal

NTFS keeps a journal describing all changes that have been made to the file system. User-mode services can receive notifications of changes to the journal and then identify what files have changed. The content-indexing service uses the change journal to identify files that need to be re-indexed. The file-replication service uses it to identify files that need to be replicated across the network.

22.5.8 Volume Shadow Copies

Windows XP implements the capability of bringing a volume to a known state and then creating a shadow copy that can be used to back up a consistent view of the volume. Making a shadow copy of a volume is a form of copy-on-write, where blocks modified after the shadow copy is created have their original contents stashed in the copy. To achieve a consistent state for the volume requires the cooperation of applications, since the system cannot know when the data used by the application are in a stable state from which the application could be safely restarted.

The server version of Windows XP uses shadow copies to efficiently maintain old versions of files stored on file servers. This allows users to see documents stored on file servers as they existed at earlier points in time. The user can use this feature to recover files that were accidentally deleted or simply to look at a previous version of the file, all without pulling out a backup tape.

22.6 Networking

Windows XP supports both peer-to-peer and client-server networking. It also has facilities for network management. The networking components in Windows XP provide data transport, interprocess communication, file sharing across a network, and the ability to send print jobs to remote printers.

22.6.1 Network Interfaces

To describe networking in Windows XP, we must first mention two of the internal networking interfaces: the **network device interface specification (NDIS)** and the **transport driver interface (TDI)**. The NDIS interface was developed in 1989 by Microsoft and 3Com to separate network adapters from transport protocols so that either could be changed without affecting the other. NDIS resides at the interface between the data-link-control and media-access-control layers in the OSI model and enables many protocols to operate over many different network adapters. In terms of the OSI model, the TDI is the interface between the transport layer (layer 4) and the session layer (layer 5). This interface enables any session-layer component to use any available transport mechanism. (Similar reasoning led to the streams mechanism in UNIX.) The TDI supports both connection-based and connectionless transport and has functions to send any type of data.

22.6.2 Protocols

Windows XP implements transport protocols as drivers. These drivers can be loaded and unloaded from the system dynamically, although in practice the system typically has to be rebooted after a change. Windows XP comes with several networking protocols. Next, we discuss a number of the protocols supported in Windows XP to provide a variety of network functionality.

22.6.2.1 Server-Message Block

The **server-message-block (SMB)** protocol was first introduced in MS-DOS 3.1. The system uses the protocol to send I/O requests over the network. The SMB protocol has four message types. The **Session control** messages are commands that start and end a redirector connection to a shared resource at the server. A redirector uses **File** messages to access files at the server. The system uses **Printer** messages to send data to a remote print queue and to receive back status information, and the **Message** message is used to communicate with another workstation. The SMB protocol was published as the **Common Internet File System (CIFS)** and is supported on a number of operating systems.

22.6.2.2 Network Basic Input/Output System

The **network basic input/output system (NetBIOS)** is a hardware-abstraction interface for networks, analogous to the BIOS hardware-abstraction interface devised for PCs running MS-DOS. NetBIOS, developed in the early 1980s, has become a standard network-programming interface. NetBIOS is used to establish logical names on the network, to establish logical connections, or **sessions**, between two logical names on the network, and to support reliable data transfer for a session via either NetBIOS or SMB requests.

22.6.2.3 NetBIOS Extended User Interface

The **NetBIOS extended user interface (NetBEUI)** was introduced by IBM in 1985 as a simple, efficient networking protocol for up to 254 machines. It is the default protocol for Windows 95 peer networking and for Windows for Workgroups. Windows XP uses NetBEUI when it wants to share resources with these networks. Among the limitations of NetBEUI are that it uses the actual name of a computer as the address and that it does not support routing.

22.6.2.4 Transmission Control Protocol/Internet Protocol

The transmission control protocol/Internet protocol (TCP/IP) suite that is used on the Internet has become the de facto standard networking infrastructure. Windows XP uses TCP/IP to connect to a wide variety of operating systems and hardware platforms. The Windows XP TCP/IP package includes the simple network-management protocol (SNM), dynamic host-configuration protocol (DHCP), Windows Internet name service (WINS), and NetBIOS support.

22.6.2.5 Point-to-Point Tunneling Protocol

The **point-to-point tunneling protocol (PPTP)** is a protocol provided by Windows XP to communicate between remote-access server modules running on Windows XP server machines and other client systems that are connected over the Internet. The remote-access servers can encrypt data sent over the connection, and they support multi-protocol **virtual private networks (VPNs)** over the Internet.

22.6.2.6 Novell NetWare Protocols

The Novell NetWare protocols (IPX datagram service on the SPX transport layer) are widely used for PC LANs. The Windows XP NWLink protocol connects the NetBIOS to NetWare networks. In combination with a redirector (such as Microsoft's Client Service for NetWare or Novell's NetWare Client for Windows), this protocol enables a Windows XP client to connect to a NetWare server.

22.6.2.7 Web Distributed Authoring and Versioning Protocol

Web distributed authoring and versioning (WebDAV) is an http-based protocol for collaborative authoring across the network. Windows XP builds a WebDAV redirector into the file system. By building WebDAV support directly into the file system, it can work with other features, such as encryption. Personal files can now be stored securely in a public place.

22.6.2.8 AppleTalk Protocol

The **AppleTalk protocol** was designed as a low-cost connection by Apple to allow Macintosh computers to share files. Windows XP systems can share files and printers with Macintosh computers via AppleTalk if a Windows XP server on the network is running the Windows Services for Macintosh package.

22.6.3 Distributed-Processing Mechanisms

Although Windows XP is not a distributed operating system, it does support distributed applications. Mechanisms that support distributed processing on Windows XP include NetBIOS, named pipes and mailslots, Windows sockets, RPCs, the Microsoft Interface Definition Language, and finally COM.

22.6.3.1 NetBIOS

In Windows XP, NetBIOS applications can communicate over the network using NetBEUI, NWLink, or TCP/IP.

22.6.3.2 Named Pipes

Named pipes are a connection-oriented messaging mechanism. Named pipes were originally developed as a high-level interface to NetBIOS connections over the network. A process can also use named pipes to communicate with other processes on the same machine. Since named pipes are accessed through the file-system interface, the security mechanisms used for file objects also apply to named pipes.

The name of a named pipe has a format called the **uniform naming convention (UNC)**. A UNC name looks like a typical remote file name. The format of a UNC name is `\\server_name\share_name\x\y\z`, where the `server_name` identifies a server on the network; a `share_name` identifies any resource that is made available to network users, such as directories, files, named pipes, and printers; and the `\x\y\z` part is a normal file path name.

22.6.3.3 Mailslots

Mailslots are a connectionless messaging mechanism. They are unreliable when accessed across the network, in that a message sent to a mailslot may be lost before the intended recipient receives it. Mailslots are used for broadcast applications, such as finding components on the network; they are also used by the Windows computer browser service.

22.6.3.4 Winsock

Winsock is the Windows XP sockets API. Winsock is a session-layer interface that is largely compatible with UNIX sockets but has some added Windows XP extensions. It provides a standardized interface to many transport protocols that may have different addressing schemes, so that any Winsock application can run on any Winsock-compliant protocol stack.

22.6.3.5 Remote Procedure Calls

A remote procedure call (RPC) is a client-server mechanism that enables an application on one machine to make a procedure call to code on another machine. The client calls a local procedure—a **stub routine**—that packs its arguments into a message and sends them across the network to a particular server process. The client-side stub routine then blocks. Meanwhile, the server unpacks the message, calls the procedure, packs the return results into a message, and sends them back to the client stub. The client stub unblocks, receives the message, unpacks the results of the RPC, and returns them to

the caller. This packing of arguments is sometimes called **marshalling**. The Windows XP RPC mechanism follows the widely used distributed-computing-environment standard for RPC messages, so programs written to use Windows XP RPCs are highly portable. The RPC standard is detailed. It hides many of the architectural differences among computers, such as the sizes of binary numbers and the order of bytes and bits in computer words, by specifying standard data formats for RPC messages.

Windows XP can send RPC messages using NetBIOS, or Winsock on TCP/IP networks, or named pipes on LAN Manager networks. The LPC facility, discussed earlier, is similar to RPC, except that in the case of LPC the messages are passed between two processes running on the same computer.

22.6.3.6 Microsoft Interface Definition Language

It is tedious and error-prone to write the code to marshal and transmit arguments in the standard format, to unmarshal and execute the remote procedure, to marshal and send the return results, and to unmarshal and return them to the caller. Fortunately, however, much of this code can be generated automatically from a simple description of the arguments and return results.

Windows XP provides the **Microsoft Interface Definition Language** to describe the remote procedure names, arguments, and results. The compiler for this language generates header files that declare the stubs for the remote procedures, as well as the data types for the argument and return-value messages. It also generates source code for the stub routines used at the client side and for an unmarshaller and dispatcher at the server side. When the application is linked, the stub routines are included. When the application executes the RPC stub, the generated code handles the rest.

22.6.3.7 Component Object Model

The **component object model (COM)** is a mechanism for interprocess communication that was developed for Windows. COM objects provide a well-defined interface to manipulate the data in the object. For instance, COM is the infrastructure used by Microsoft's **object linking and embedding (OLE)** technology for inserting spreadsheets into Microsoft Word documents. Windows XP has a distributed extension called **DCOM** that can be used over a network utilizing RPC to provide a transparent method of developing distributed applications.

22.6.4 Redirectors and Servers

In Windows XP, an application can use the Windows XP I/O API to access files from a remote computer as though they were local, provided that the remote computer is running a CIFS server, such as is provided by Windows XP or earlier Windows systems. A **redirector** is the client-side object that forwards I/O requests to remote files, where they are satisfied by a server. For performance and security, the redirectors and servers run in kernel mode.

In more detail, access to a remote file occurs as follows:

1. The application calls the I/O manager to request that a file be opened with a file name in the standard UNC format.

2. The I/O manager builds an I/O request packet, as described in Section 22.3.3.5.
3. The I/O manager recognizes that the access is for a remote file and calls a driver called a **multiple universal-naming-convention provider (MUP)**.
4. The MUP sends the I/O request packet asynchronously to all registered redirectors.
5. A redirector that can satisfy the request responds to the MUP. To avoid asking all the redirectors the same question in the future, the MUP uses a cache to remember which redirector can handle this file.
6. The redirector sends the network request to the remote system.
The remote-system network drivers receive the request and pass it to the server driver.
8. The server driver hands the request to the proper local file-system driver.
9. The proper device driver is called to access the data.
10. The results are returned to the server driver, which sends the data back to the requesting redirector. The redirector then returns the data to the calling application via the I/O manager.

A similar process occurs for applications that use the Win32 API network API, rather than the UNC services, except that a module called a multi-provider router is used instead of a MUP.

For portability, redirectors and servers use the TDI API for network transport. The requests themselves are expressed in a higher-level protocol, which by default is the SMB protocol mentioned in Section 22.6.2. The list of redirectors is maintained in the system registry database.

22.6.4.1 Distributed File System

The UNC names are not always convenient, because multiple file servers may be available to serve the same content, and UNC names explicitly include the name of the server. Windows XP supports a **distributed file system (DFS)** protocol that allows a network administrator to serve up files from multiple servers using a single distributed name space.

22.6.4.2 Folder Redirection and Client-Side Caching

To improve the PC experience for business users who frequently switch among computers, Windows XP allows administrators to give users **roaming profiles**, which keep users preferences and other settings on servers. **Folder redirection** is then used to automatically store a user's documents and other files on a server. This works well until one of the computers is no longer attached to the network, such as a laptop on an airplane. To give users off-line access to their redirected files, Windows XP uses **client-side caching (CSC)**. CSC is used when the computer is online to keep copies of the server files on the local machine for better performance. The files are pushed up to the server as they are changed. If the computer becomes disconnected, the files are still available,

and the update of the server is deferred until the next time the computer is online with a suitably performing network link.

22.6.5 Domains

Many networked environments have natural groups of users, such as students in a computer laboratory at school or employees in one department in a business. Frequently, we want all the members of the group to be able to access shared resources on their various computers in the group. To manage the global access rights within such groups, Windows XP uses the concept of a domain. Previously, these domains had no relationship whatsoever to the domain-name system (DNS) that maps Internet host names to IP addresses. Now, however, they are closely related.

Specifically, a Windows XP domain is a group of Windows XP workstations and servers that share a common security policy and user database. Since Windows XP now uses the Kerberos protocol for trust and authentication, a Windows XP domain is the same thing as a Kerberos realm. Previous versions of NT used the idea of primary and backup domain controllers; now all servers in a domain are domain controllers. In addition, previous versions required the setup of one-way trusts between domains. Windows XP uses a hierarchical approach based on DNS and allows transitive trusts that can flow up and down the hierarchy. This approach reduces the number of trusts required for n domains from $n * (n - 1)$ to $O(n)$. The workstations in the domain trust the domain controller to give correct information about the access rights of each user (via the user's access token). All users retain the ability to restrict access to their own workstations, no matter what any domain controller may say.

22.6.5.1 Domain Trees and Forests

Because a business may have many departments and a school may have many classes, it is often necessary to manage multiple domains within a single organization. A **domain tree** is a contiguous DNS naming hierarchy for managing multiple domains. For example, *bell-labs.com* might be the root of the tree, with *research.bell-labs.com* and *pez.bell-labs.com* as children—domains *research* and *pez*. A **forest** is a set of noncontiguous names. An example would be the trees *bell-labs.com* and/or *lucent.com*. A forest may be made up of only one domain tree, however.

22.6.5.2 Trust Relationships

Trust relationships may be set up between domains in three ways: one-way, transitive, and cross-link. Versions of NT through 4.0 allowed only one-way trusts. A **one-way trust** is exactly what its name implies: Domain A is told it can trust domain B. However, B will not trust A unless another relationship is configured. Under a **transitive trust**, if A trusts B and B trusts C, then A, B, and C all trust one another, since transitive trusts are two-way by default. Transitive trusts are enabled by default for new domains in a tree and can be configured only among domains within a forest. The third type, a **cross-link trust**, is useful to cut down on authentication traffic. Suppose that domains A and B are leaf nodes and that users in A often use resources in B. If a standard transitive trust is used, authentication requests must traverse up to the common ancestor of

the two leaf nodes; but if A and B have a cross-linking trust established, the authentications are sent directly to the other node.

22.6.6 Active Directory

Active Directory is the Windows XP implementation of **lightweight directory-access protocol (LDAP)** services. Active Directory stores the topology information about the domain, keeps the domain-based user and group accounts and passwords, and provides a domain-based store for technologies like **group policies** and **intellimirror**.

Administrators use group policies to establish standards for desktop preferences and software. For many corporate information-technology groups, uniformity drastically reduces the cost of computing. Intellimirror is used in conjunction with group policies to specify what software should be available to each class of user, even automatically installing it on demand from a corporate server.

22.6.7 Name Resolution in TCP/IP Networks

On an IP network, **name resolution** is the process of converting a computer name to an IP address, such as resolving *www.bell-labs.com* to 135.104.1.14. Windows XP provides several methods of name resolution, including Windows Internet name service (WINS), broadcast-name resolution, domain-name system (DNS), a hosts file, and an LMHOSTS file. Most of these methods are used by many operating systems, so we describe only WINS here.

Under WINS, two or more WINS servers maintain a dynamic database of name-to-IP address bindings, along with client software to query the servers. At least two servers are used, so that the WINS service can survive a server failure and so that the name-resolution workload can be spread over multiple machines.

WINS uses the dynamic host-configuration protocol (DHCP). DHCP updates address configurations automatically in the WINS database, without user or administrator intervention, as follows. When a DHCP client starts up, it broadcasts a **discover** message. Each DHCP server that receives the message replies with an **offer** message that contains an IP address and configuration information for the client. The client chooses one of the configurations and sends a **request** message to the selected DHCP server. The DHCP server responds with the IP address and configuration information it gave previously and with a **lease** for that address. The lease gives the client the right to use the IP address for a specified period of time. When the lease time is half expired, the client attempts to renew the lease for the address. If the lease is not renewed, the client must obtain a new one.

22.7 Win32 API

The Win32 API is the fundamental interface to the capabilities of Windows XP. This section describes five main aspects of the Win32 API: access to kernel objects, sharing of objects between processes, process management, interprocess communication, and memory management.

22.7.1 Access to Kernel Objects

The Windows XP kernel provides many services that application programs can use. Application programs obtain these services by manipulating kernel objects. A process gains access to a kernel object named *XXX* by calling the `CreateXXX` function to open a handle to *XXX*. This handle is unique to the process. Depending on which object is being opened, if the `Create()` function fails, it may return 0, or it may return a special constant named `INVALID_HANDLE_VALUE`. A process can close any handle by calling the `CloseHandle()` function, and the system may delete the object if the count of processes using the object drops to 0.

22.7.2 Sharing Objects Between Processes

Windows XP provides three ways to share objects between processes. The first way is for a child process to inherit a handle to the object. When the parent calls the `CreateXXX` function, the parent supplies a `SECURITY_ATTRIBUTES` structure with the `bInheritHandle` field set to `TRUE`. This field creates an inheritable handle. Next, the child process is created, passing a value of `TRUE` to the `CreateProcess()` function's `bInheritHandle` argument. Figure 22.11 shows a code sample that creates a semaphore handle inherited by a child process.

Assuming the child process knows which handles are shared, the parent and child can achieve interprocess communication through the shared objects. In the example in Figure 22.11, the child process gets the value of the handle from the first command-line argument and then shares the semaphore with the parent process.

The second way to share objects is for one process to give the object a name when the object is created and for the second process to open the name. This method has two drawbacks: Windows XP does not provide a way to check whether an object with the chosen name already exists, and the object name space is global, without regard to the object type. For instance, two applications may create an object named *pipe* when two distinct—and possibly different—objects are desired.

Named objects have the advantage that unrelated processes can readily share them. The first process calls one of the `CreateXXX` functions and supplies a name in the `lpzName` parameter. The second process gets a handle to share

```
SECURITY_ATTRIBUTES sa;
sa.nlength = sizeof(sa);
sa.lpSecurityDescriptor = NULL;
sa.bInheritHandle = TRUE;
Handle a_semaphore = CreateSemaphore(&sa, 1, 1, NULL);
char command_line[132];
ostream ostream(command_line, sizeof(command_line));
ostream << a_semaphore << ends;
CreateProcess("another_process.exe", command_line,
    NULL, NULL, TRUE, . . .);
```

Figure 22.11 Code enabling a child to share an object by inheriting a handle.

```

// Process A
. . .
HANDLE a_semaphore = CreateSemaphore(NULL, 1, 1, "MySEM1");
. . .

// Process B
. . .
HANDLE b_semaphore = OpenSemaphore(SEMAPHORE_ALL_ACCESS,
    FALSE, "MySEM1");
. . .

```

Figure 22.12 Code for sharing an object by name lookup.

the object by calling `OpenXXX()` (or `CreateXXX()`) with the same name, as shown in the example of Figure 22.12.

The third way to share objects is via the `DuplicateHandle()` function. This method requires some other method of interprocess communication to pass the duplicated handle. Given a handle to a process and the value of a handle within that process, a second process can get a handle to the same object and thus share it. An example of this method is shown in Figure 22.13.

22.7.3 Process Management

In Windows XP, a **process** is an executing instance of an application, and a **thread** is a unit of code that can be scheduled by the operating system. Thus, a process contains one or more threads. A process is started when some other process calls the `CreateProcess()` routine. This routine loads any dynamic link libraries used by the process and creates a **primary thread**.

```

// Process A wants to give Process B access to a semaphore

// Process A
HANDLE a_semaphore = CreateSemaphore(NULL, 1, 1, NULL);
// send the value of the semaphore to Process B
// using a message or shared memory object
. . .

// Process B
HANDLE process_a = OpenProcess(PROCESS_ALL_ACCESS, FALSE,
    process_id_of_A);
HANDLE b_semaphore;
DuplicateHandle(process_a, a_semaphore,
    GetCurrentProcess(), &b_semaphore,
    0, FALSE, DUPLICATE_SAME_ACCESS);
// use b_semaphore to access the semaphore
. . .

```

Figure 22.13 Code for sharing an object by passing a handle.

Additional threads can be created by the `CreateThread()` function. Each thread is created with its own stack, which defaults to 1 MB unless specified otherwise in an argument to `CreateThread()`. Because some C run-time functions maintain state in static variables, such as `errno`, a multithread application needs to guard against unsynchronized access. The wrapper function `beginthreadex()` provides appropriate synchronization.

22.7.3.1 Instance Handles

Every dynamic link library or executable file loaded into the address space of a process is identified by an **instance handle**. The value of the instance handle is actually the virtual address where the file is loaded. An application can get the handle to a module in its address space by passing the name of the module to `GetModuleHandle()`. If `NULL` is passed as the name, the base address of the process is returned. The lowest 64 KB of the address space are not used, so a faulty program that tries to de-reference a `NULL` pointer gets an access violation.

Priorities in the Win32 API environment are based on the Windows XP scheduling model, but not all priority values may be chosen. Win32 API uses four priority classes:

1. `IDLE_PRIORITY_CLASS` (priority level 4)
2. `NORMAL_PRIORITY_CLASS` (priority level 8)
3. `HIGH_PRIORITY_CLASS` (priority level 13)
4. `REALTIME_PRIORITY_CLASS` (priority level 24)

Processes are typically members of the `NORMAL_PRIORITY_CLASS` unless the parent of the process was of the `IDLE_PRIORITY_CLASS` or another class was specified when `CreateProcess` was called. The priority class of a process can be changed with the `SetPriorityClass()` function or by passing of an argument to the `START` command. For example, the command `START /REALTIME cbsver.exe` would run the `cbsver` program in the `REALTIME_PRIORITY_CLASS`. Only users with the *increase scheduling priority* privilege can move a process into the `REALTIME_PRIORITY_CLASS`. Administrators and power users have this privilege by default.

22.7.3.2 Scheduling Rule

When a user is running an interactive program, the system needs to provide especially good performance for the process. For this reason, Windows XP has a special scheduling rule for processes in the `NORMAL_PRIORITY_CLASS`. Windows XP distinguishes between the foreground process that is currently selected on the screen and the background processes that are not currently selected. When a process moves into the foreground, Windows XP increases the scheduling quantum by some factor—typically by 3. (This factor can be changed via the *performance option* in the system section of the control panel.) This increase gives the foreground process three times longer to run before a time-sharing preemption occurs.

22.7.3.3 Thread Priorities

A thread starts with an initial priority determined by its class. The priority can be altered by the `SetThreadPriority()` function. This function takes an argument that specifies a priority relative to the base priority of its class:

```

THREAD_PRIORITY_LOWEST: base - 2
THREAD_PRIORITY_BELOW_NORMAL: base - 1
THREAD_PRIORITY_NORMAL: base + 0
THREAD_PRIORITY_ABOVE_NORMAL: base + 1
THREAD_PRIORITY_HIGHEST: base + 2

```

Two other designations are also used to adjust the priority. Recall from Section 22.3.2.1 that the kernel has two priority classes: 16–31 for the real-time class and 0–15 for the variable-priority class. `THREAD_PRIORITY_IDLE` sets the priority to 16 for real-time threads and to 1 for variable-priority threads. `THREAD_PRIORITY_TIME_CRITICAL` sets the priority to 31 for real-time threads and to 15 for variable-priority threads.

As we discussed in Section 22.3.2.1, the kernel adjusts the priority of a thread dynamically depending on whether the thread is I/O bound or CPU bound. The Win32 API provides a method to disable this adjustment via `SetProcessPriorityBoost()` and `SetThreadPriorityBoost()` functions.

22.7.3.4 Thread Synchronization

A thread can be created in a **suspended state**; the thread does not execute until another thread makes it eligible via the `ResumeThread()` function. The `SuspendThread()` function does the opposite. These functions set a counter, so if a thread is suspended twice, it must be resumed twice before it can run. To synchronize the concurrent access to shared objects by threads, the kernel provides synchronization objects, such as semaphores and mutexes.

In addition, synchronization of threads can be achieved by use of the `WaitForSingleObject()` and `WaitForMultipleObjects()` functions. Another method of synchronization in the Win32 API is the critical section. A critical section is a synchronized region of code that can be executed by only one thread at a time. A thread establishes a critical section by calling `InitializeCriticalSection()`. The application must call `EnterCriticalSection()` before entering the critical section and `LeaveCriticalSection()` after exiting the critical section. These two routines guarantee that, if multiple threads attempt to enter the critical section concurrently, only one thread at a time will be permitted to proceed; the others will wait in the `EnterCriticalSection()` routine. The critical-section mechanism is faster than using kernel-synchronization objects because it does not allocate kernel objects until it first encounters contention for the critical section.

22.7.3.5 Fibers

A **fiber** is user-mode code that is scheduled according to a user-defined scheduling algorithm. A process may have multiple fibers in it, just as it may

have multiple threads. A major difference between threads and fibers is that whereas threads can execute concurrently, only one fiber at a time is permitted to execute, even on multiprocessor hardware. This mechanism is included in Windows XP to facilitate the porting of those legacy UNIX applications that were written for a fiber-execution model.

The system creates a fiber by calling either `ConvertThreadToFiber()` or `CreateFiber()`. The primary difference between these functions is that `CreateFiber()` does not begin executing the fiber that was created. To begin execution, the application must call `SwitchToFiber()`. The application can terminate a fiber by calling `DeleteFiber()`.

22.7.3.6 Thread Pool

Repeated creation and deletion of threads can be expensive for applications and services that perform small amounts of work in each. The thread pool provides user-mode programs with three services: a queue to which work requests may be submitted (via the `QueueUserWorkItem()` API), an API that can be used to bind callbacks to waitable handles (`RegisterWaitForSingleObject()`), and APIs to bind callbacks to timeouts (`CreateTimerQueue()` and `CreateTimerQueueTimer()`).

The thread pool's goal is to increase performance. Threads are relatively expensive, and a processor can only be executing one thing at a time no matter how many threads are used. The thread pool attempts to reduce the number of outstanding threads by slightly delaying work requests (reusing each thread for many requests) while providing enough threads to effectively utilize the machine's CPUs. The wait and timer-callback APIs allow the thread pool to further reduce the number of threads in a process, using far fewer threads than would be necessary if a process were to devote one thread to servicing each waitable handle or timeout.

22.7.4 Interprocess Communication

Win32 API applications handle interprocess communication in several ways. One way is by sharing kernel objects. Another way is by passing messages, an approach that is particularly popular for Windows GUI applications. One thread can send a message to another thread or to a window by calling `PostMessage()`, `PostThreadMessage()`, `SendMessage()`, `SendThreadMessage()`, or `SendMessageCallback()`. The difference between *posting* a message and *sending* a message is that the post routines are asynchronous: They return immediately, and the calling thread does not know when the message is actually delivered. The send routines are synchronous: They block the caller until the message has been delivered and processed.

In addition to sending a message, a thread can send data with the message. Since processes have separate address spaces, the data must be copied. The system copies data by calling `SendMessage()` to send a message of type `WM_COPYDATA` with a `COPYDATASTRUCT` data structure that contains the length and address of the data to be transferred. When the message is sent, Windows XP copies the data to a new block of memory and gives the virtual address of the new block to the receiving process.

Unlike threads in the 16-bit Windows environment, every Win32 API thread has its own input queue from which it receives messages. (All input is received

```

// allocate 16 MB at the top of our address space
void *buf = VirtualAlloc(0, 0x1000000, MEM_RESERVE | MEM_TOP_DOWN,
    PAGE_READWRITE);
// commit the upper 8 MB of the allocated space
VirtualAlloc(buf + 0x800000, 0x800000, MEM_COMMIT, PAGE_READWRITE);
// do something with the memory
. . .
// now decommit the memory
VirtualFree(buf + 0x800000, 0x800000, MEM_DECOMMIT);
// release all of the allocated address space
VirtualFree(buf, 0, MEM_RELEASE);

```

Figure 22.14 Code fragments for allocating virtual memory.

via messages.) This structure is more reliable than the shared input queue of 16-bit Windows, because, with separate queues, it is no longer possible for one stuck application to block input to the other applications. If a Win32 API application does not call `GetMessage()` to handle events on its input queue, the queue fills up; and after about five seconds, the system marks the application as “Not Responding”.

22.7.5 Memory Management

The Win32 API provides several ways for an application to use memory: virtual memory, memory-mapped files, heaps, and thread-local storage.

22.7.5.1 Virtual Memory

An application calls `VirtualAlloc()` to reserve or commit virtual memory and `VirtualFree()` to decommit or release the memory. These functions enable the application to specify the virtual address at which the memory is allocated. They operate on multiples of the memory page size, and the starting address of an allocated region must be greater than `0x10000`. Examples of these functions appear in Figure 22.14.

A process may lock some of its committed pages into physical memory by calling `VirtualLock()`. The maximum number of pages a process can lock is 30, unless the process first calls `SetProcessWorkingSetSize()` to increase the maximum working-set size.

22.7.5.2 Memory-Mapping Files

Another way for an application to use memory is by memory-mapping a file into its address space. Memory mapping is also a convenient way for two processes to share memory: Both processes map the same file into their virtual memory. Memory mapping is a multistage process, as you can see in the example in Figure 22.15.

If a process wants to map some address space just to share a memory region with another process, no file is needed. The process calls `CreateFileMapping()` with a file handle of `0xffffffff` and a particular size. The resulting file-mapping object can be shared by inheritance, by name lookup, or by duplication.

```

// open the file or create it if it does not exist
HANDLE hfile = CreateFile("somefile", GENERIC_READ | GENERIC_WRITE,
    FILE_SHARE_READ | FILE_SHARE_WRITE, NULL,
    OPEN_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
// create the file mapping 8 MB in size
HANDLE hmap = CreateFileMapping(hfile, PAGE_READWRITE,
    SEC_COMMIT, 0, 0x800000, "SHM.1");
// now get a view of the space mapped
void *buf = MapViewOfFile(hmap, FILE_MAP_ALL_ACCESS,
    0, 0, 0, 0x800000);
// do something with the mapped file
. . .
// now unmap the file
UnMapViewOfFile(buf);
CloseHandle(hmap);
CloseHandle(hfile);

```

Figure 22.15 Code fragments for memory mapping of a file.

22.7.5.3 Heaps

Heaps provide a third way for applications to use memory. A heap in the Win32 environment is a region of reserved address space. When a Win32 API process is initialized, it is created with a 1-MB **default heap**. Since many Win32 API functions use the default heap, access to the heap is synchronized to protect the heap's space-allocation data structures from being damaged by concurrent updates by multiple threads.

Win32 API provides several heap-management functions so that a process can allocate and manage a private heap. These functions are `HeapCreate()`, `HeapAlloc()`, `HeapRealloc()`, `HeapSize()`, `HeapFree()`, and `HeapDestroy()`. The Win32 API also provides the `HeapLock()` and `HeapUnlock()` functions to enable a thread to gain exclusive access to a heap. Unlike `VirtualLock()`, these functions perform only synchronization; they do not lock pages into physical memory.

22.7.5.4 Thread-Local Storage

The fourth way for applications to use memory is through a thread-local storage mechanism. Functions that rely on global or static data typically fail to work properly in a multithreaded environment. For instance, the C runtime function `strtok()` uses a static variable to keep track of its current position while parsing a string. For two concurrent threads to execute `strtok()` correctly, they need separate *current position* variables. The thread-local storage mechanism allocates global storage on a per-thread basis. It provides both dynamic and static methods of creating thread-local storage. The dynamic method is illustrated in Figure 22.16.

To use a thread-local static variable, the application declares the variable as follows to ensure that every thread has its own private copy:

```
__declspec(thread) DWORD cur_pos = 0;
```

```

// reserve a slot for a variable
DWORD var_index = TlsAlloc();
// set it to the value 10
TlsSetValue(var_index, 10);
// get the value
int var = TlsGetValue(var_index);
// release the index
TlsFree(var_index);

```

Figure 22.16 Code for dynamic thread-local storage.

22.8

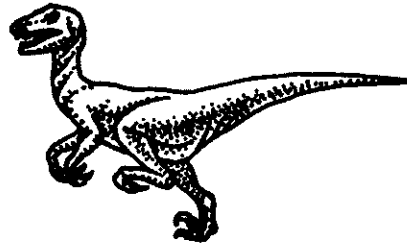
Microsoft designed Windows XP to be an extensible, portable operating system—one able to take advantage of new techniques and hardware. Windows XP supports multiple operating environments and symmetric multiprocessing, including both 32-bit and 64-bit processors and NUMA computers. The use of kernel objects to provide basic services, along with support for client-server computing, enables Windows XP to support a wide variety of application environments. For instance, Windows XP can run programs compiled for MS-DOS, Windows 16, Windows 95, Windows XP, and POSIX. It provides virtual memory, integrated caching, and preemptive scheduling. Windows XP supports a security model stronger than those of previous Microsoft operating systems and includes internationalization features. Windows XP runs on a wide variety of computers, so users can choose and upgrade hardware to match their budgets and performance requirements without needing to alter the applications they run.

- 22.1 Under what circumstances would one use the deferred procedure calls facility in Windows XP?
- 22.2 What is a handle, and how does a process obtain a handle?
- 22.3 Describe a useful application of the no-access page facility provided in Windows XP.
- 22.4 The IA64 processors contain registers that can be used to address a 64-bit address space. However, Windows XP limits the address space of user programs to 8 TB, which corresponds to 43 bits' worth. Why was this decision made?
- 22.5 What manages cache in Windows XP? How is cache managed?
- 22.6 What is the purpose of the Win16 execution environment? What limitations are imposed on the programs executing inside this environment? What are the protection guarantees provided between different applications executing inside the Windows 16 environment? What are

- the protection guarantees provided between an application executing inside the Windows16 environment and a 32-bit application?
- 22.7 How does the NTFS directory structure differ from the directory structure used in Unix operating systems?
 - 22.8 What is a process, and how is it managed in Windows XP?
 - 22.9 What is the fiber abstraction provided by Windows XP? How does it differ from the threads abstraction?

Solomon and Russinovich [2000] give an overview of Windows XP and considerable technical detail about system internals and components. Tate [2000] is a good reference on using Windows XP. The Microsoft Windows XP Server Resource Kit (Microsoft [2000b]) is a six-volume set helpful for using and deploying Windows XP. The Microsoft Developer Network Library (Microsoft [2000a]), issued quarterly, supplies a wealth of information on Windows XP and other Microsoft products.

Iseminger [2000] provides a good reference on the Windows XP Active Directory. Richter [1997] gives a detailed discussion on writing programs that use the Win32 API. Silberschatz et al. [2001] contains a good discussion of B+ trees.



Now that you understand the fundamental concepts of operating systems (CPU scheduling, memory management, processes, and so on), we are in a position to examine how these concepts have been applied in several older and highly influential operating systems. Some of them (such as the XDS-940 and the THE system) were one-of-a-kind systems; others (such as OS/360) are widely used. The order of presentation highlights the similarities and differences of the systems; it is not strictly chronological or ordered by importance. The serious student of operating systems should be familiar with all these systems.

As we describe early systems, we include references to further reading. The papers, written by the designers of the systems, are important both for their technical content and for their style and flavor.

23.1 Early Systems

Early computers were physically enormous machines run from a console. The programmer, who was also the operator of the computer system, would write a program and then would operate the program directly from the operator's console. First, the program would be loaded manually into memory from the front panel switches (one instruction at a time), from paper tape, or from punched cards. Then, the appropriate buttons would be pushed to set the starting address and to start the execution of the program. As the program ran, the programmer/operator could monitor its execution by the display lights on the console. If errors were discovered, the programmer could halt the program, examine the contents of memory and registers, and debug the program directly from the console. Output was printed or was punched onto paper tape or cards for later printing.

23.1.1 Dedicated Computer Systems

As time went on, additional software and hardware were developed. Card readers, line printers, and magnetic tape became commonplace. Assemblers, loaders, and linkers were designed to ease the programming task. Libraries of common functions were created. Common functions could then be copied

into a new program without having to be written again, providing software reusability.

The routines that performed I/O were especially important. Each new I/O device had its own characteristics, requiring careful programming. A special subroutine—called a device driver—was written for each I/O device. A device driver knows how the buffers, flags, registers, control bits, and status bits for a particular device should be used. Each type of device has its own driver. A simple task, such as reading a character from a paper-tape reader, might involve complex sequences of device-specific operations. Rather than writing the necessary code every time, the device driver was simply used from the library.

Later, compilers for FORTRAN, COBOL, and other languages appeared, making the programming task much easier but the operation of the computer more complex. To prepare a FORTRAN program for execution, for example, the programmer would first need to load the FORTRAN compiler into the computer. The compiler was normally kept on magnetic tape, so the proper tape would need to be mounted on a tape drive. The program would be read through the card reader and written onto another tape. The FORTRAN compiler produced assembly-language output, which then needed to be assembled. This procedure required mounting another tape with the assembler. The output of the assembler would need to be linked to supporting library routines. Finally, the binary object form of the program would be ready to execute. It could be loaded into memory and debugged from the console, as before.

A significant amount of **set-up time** could be involved in the running of a job. Each job consisted of many separate steps:

- Loading the FORTRAN compiler tape
- Running the compiler
- Unloading the compiler tape
- Loading the assembler tape
- Running the assembler
- Unloading the assembler tape
- Loading the object program
- Running the object program

If an error occurred during any step, the programmer/operator might have to start over at the beginning. Each job step might involve the loading and unloading of magnetic tapes, paper tapes, and punch cards.

The job set-up time was a real problem. While tapes were being mounted or the programmer was operating the console, the CPU sat idle. Remember that, in the early days, few computers were available, and they were expensive. A computer might have cost millions of dollars, not including the operational costs of power, cooling, programmers, and so on. Thus, computer time was extremely valuable, and owners wanted their computers to be used as much as possible. They needed high **utilization** to get as much as they could from their investments.

23.1.2 Shared Computer Systems

The solution was two-fold. First, a professional computer operator was hired. The programmer no longer operated the machine. As soon as one job was finished, the operator could start the next. Since the operator had more experience with mounting tapes than a programmer, set-up time was reduced. The programmer provided whatever cards or tapes were needed, as well as a short description of how the job was to be run. Of course, the operator could not debug an incorrect program at the console, since the operator would not understand the program. Therefore, in the case of program error, a dump of memory and registers was taken, and the programmer had to debug from the dump. Dumping the memory and registers allowed the operator to continue immediately with the next job but left the programmer with the more difficult debugging problem.

Second, jobs with similar needs were batched together and run through the computer as a group to reduce set-up time. For instance, suppose the operator received one FORTRAN job, one COBOL job, and another FORTRAN job. If she ran them in that order, she would have to set up for FORTRAN (load the compiler tapes and so on), then set up for COBOL, and then set up for FORTRAN again. If she ran the two FORTRAN programs as a batch, however, she could set up only once for FORTRAN, saving operator time.

But there were still problems. For example, when a job stopped, the operator would have to notice that it had stopped (by observing the console), determine *why* it stopped (normal or abnormal termination), dump memory and register (if necessary), load the appropriate device with the next job, and restart the computer. During this transition from one job to the next, the CPU sat idle.

To overcome this idle time, people developed **automatic job sequencing**; with this technique, the first rudimentary operating systems were created. A small program, called a **resident monitor**, was created to transfer control automatically from one job to the next (Figure 23.1). The resident monitor is always in memory (or *resident*).

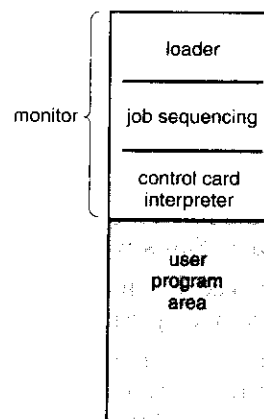


Figure 23.1 Memory layout for a resident monitor.

When the computer was turned on, the resident monitor was invoked, and it would transfer control to a program. When the program terminated, it would return control to the resident monitor, which would then go on to the next program. Thus, the resident monitor would automatically sequence from one program to another and from one job to another.

But how would the resident monitor know which program to execute? Previously, the operator had been given a short description of what programs were to be run on what data. **Control cards** were introduced to provide this information directly to the monitor. The idea is simple: In addition to the program or data for a job, the programmer included the control cards, which contained directives to the resident monitor indicating what program to run. For example, a normal user program might require one of three programs to run: the FORTRAN compiler (FTN), the assembler (ASM), or the user's program (RUN). We could use a separate control card for each of these:

```
$FTN---Execute the FORTRAN compiler.
$ASM---Execute the assembler.
$RUN---Execute the user program.
```

These cards tell the resident monitor which programs to run.

We can use two additional control cards to define the boundaries of each job:

```
$JOB---First card of a job
$END---Final card of a job
```

These two cards might be useful in accounting for the machine resources used by the programmer. Parameters can be used to define the job name, account number to be charged, and so on. Other control cards can be defined for other functions, such as asking the operator to load or unload a tape.

One problem with control cards is how to distinguish them from data or program cards. The usual solution is to identify them by a special character or pattern on the card. Several systems used the dollar-sign character (\$) in the first column to identify a control card. Others used a different code. IBM's Job Control Language (JCL) used slash marks (//) in the first two columns. Figure 23.2 shows a sample card-deck setup for a simple batch system.

A resident monitor thus has several identifiable parts:

- The **control-card interpreter** is responsible for reading and carrying out the instructions on the cards at the point of execution.
- The **loader** is invoked by the control-card interpreter to load system programs and application programs into memory at intervals.
- The **device drivers** are used by both the control-card interpreter and the loader for the system's I/O devices to perform I/O. Often, the system and application programs are linked to these same device drivers, providing continuity in their operation, as well as saving memory space and programming time.

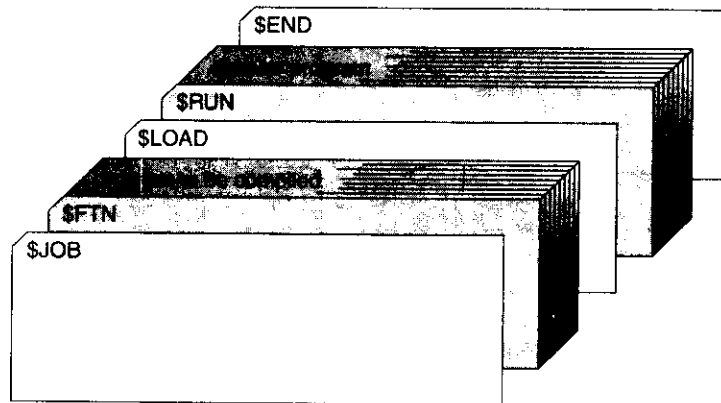


Figure 23.2 Card deck for a simple batch system.

These batch systems work fairly well. The resident monitor provides automatic job sequencing as indicated by the control cards. When a control card indicates that a program is to be run, the monitor loads the program into memory and transfers control to it. When the program completes, it transfers control back to the monitor, which reads the next control card, loads the appropriate program, and so on. This cycle is repeated until all control cards are interpreted for the job. Then, the monitor automatically continues with the next job.

The switch to batch systems with automatic job sequencing was made to improve performance. The problem, quite simply, is that humans are considerably slower than the computer. Consequently, it is desirable to replace human operation with operating-system software. Automatic job sequencing eliminates the need for human set-up time and job sequencing.

As was pointed out above, however, even with this arrangement, the CPU is often idle. The problem is the speed of the mechanical I/O devices, which are intrinsically slower than electronic devices. Even a slow CPU works in the microsecond range, with thousands of instructions executed per second. A fast card reader, in contrast, might read 1,200 cards per minute (or 20 cards per second). Thus, the difference in speed between the CPU and its I/O devices may be three orders of magnitude or more. Over time, of course, improvements in technology resulted in faster I/O devices. Unfortunately, CPU speeds increased even faster, so that the problem was not only unresolved but also exacerbated.

23.1.3 Overlapped I/O

One common solution to the I/O problem was to replace slow card readers (input devices) and line printers (output devices) with magnetic-tape units. The majority of computer systems in the late 1950s and early 1960s were batch systems reading from card readers and writing to line printers or card punches. Rather than have the CPU read directly from cards, however, the cards were first copied onto a magnetic tape via a separate device. When the tape was sufficiently full, it was taken down and carried over to the computer. When a card was needed for input to a program, the equivalent record was read from

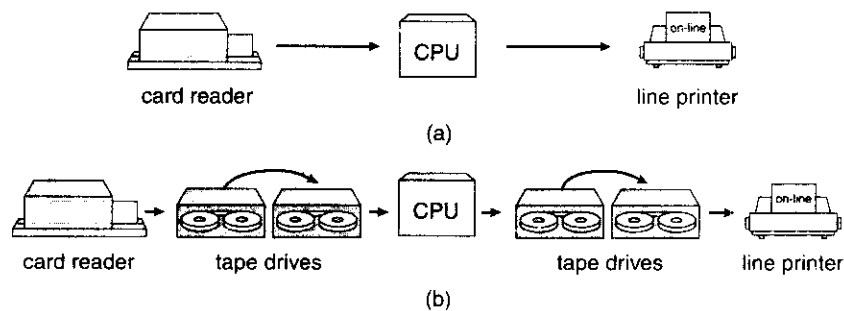


Figure 23.3 Operation of I/O devices (a) online and (b) off-line.

the tape. Similarly, output was written to the tape, and the contents of the tape were printed later. The card readers and line printers were operated *off-line*, rather than by the main computer (Figure 23.3).

An obvious advantage of off-line operation was that the main computer was no longer constrained by the speed of the card readers and line printers but was limited only by the speed of the much faster magnetic tape units. The technique of using magnetic tape for all I/O could be applied with any similar equipment (such as card readers, card punches, plotters, paper tape, and printers).

The real gain in off-line operation comes from the possibility of using multiple reader-to-tape and tape-to-printer systems for one CPU. If the CPU can process input twice as fast as the reader can read cards, then two readers working simultaneously can produce enough tape to keep the CPU busy. There is a disadvantage, too, however—a longer delay in getting a particular job run. The job must first be read onto tape. Then, it must wait until enough other jobs are read onto the tape to “fill” it. The tape must then be rewound, unloaded, hand-carried to the CPU, and mounted on a free tape drive. This process is not unreasonable for batch systems, of course. Many similar jobs can be batched onto a tape before it is taken to the computer.

Although off-line preparation of jobs continued for some time, it was quickly replaced in most systems. Disk systems became widely available and greatly improved on off-line operation. The problem with tape systems was that the card reader could not write onto one end of the tape while the CPU read from the other. The entire tape had to be written before it was rewound and read, because tapes are by nature **sequential-access devices**. Disk systems eliminated this problem by being **random-access devices**. Because the head is moved from one area of the disk to another, a disk can switch rapidly from the area on the disk being used by the card reader to store new cards to the position needed by the CPU to read the “next” card.

In a disk system, cards are read directly from the card reader onto the disk. The location of card images is recorded in a table kept by the operating system. When a job is executed, the operating system satisfies its requests for card-reader input by reading from the disk. Similarly, when the job requests the printer to output a line, that line is copied into a system buffer and is written to the disk. When the job is completed, the output is actually printed. This form of processing is called **spooling** (Figure 23.4); the name is an acronym for

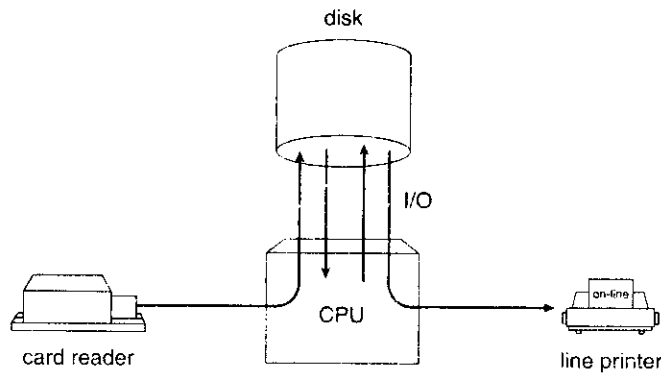


Figure 23.4 Spooling.

simultaneous peripheral operation on-line. Spooling, in essence, uses the disk as a huge buffer for reading as far ahead as possible on input devices and for storing output files until the output devices are able to accept them.

Spooling is also used for processing data at remote sites. The CPU sends the data via communication paths to a remote printer (or accepts an entire input job from a remote card reader). The remote processing is done at its own speed, with no CPU intervention. The CPU just needs to be notified when the processing is completed, so that it can spool the next batch of data.

Spooling overlaps the I/O of one job with the computation of other jobs. Even in a simple system, the spooler may be reading the input of one job while printing the output of a different job. During this time, still another job (or other jobs) may be executed, reading its "cards" from disk and "printing" its output lines onto the disk.

Spooling has a direct beneficial effect on the performance of the system. For the cost of some disk space and a few tables, the computation of one job can overlap with the I/O of other jobs. Thus, spooling can keep both the CPU and the I/O devices working at much higher rates. Spooling leads naturally to multiprogramming, which is the foundation of all modern operating systems.

23.2

The Atlas operating system (Kilburn et al. [1961], Howarth et al. [1961]) was designed at the University of Manchester in England in the late 1950s and early 1960s. Many of its basic features that were novel at the time have become standard parts of modern operating systems. Device drivers were a major part of the system. In addition, system calls were added by a set of special instructions called *extra codes*.

Atlas was a batch operating system with spooling. Spooling allowed the system to schedule jobs according to the availability of peripheral devices, such as magnetic tape units, paper tape readers, paper tape punches, line printers, card readers, and card punches.

The most remarkable feature of Atlas, however, was its memory management. **Core memory** was new and expensive at the time. Many computers, like the IBM 650, used a drum for primary memory. The Atlas system used a drum for its main memory, but it had a small amount of core memory that was used as a cache for the drum. Demand paging was used to transfer information between core memory and the drum automatically.

The Atlas system used a British computer with 48-bit words. Addresses were 24 bits but were encoded in decimal, which allowed only 1 million words to be addressed. At that time, this was an extremely large address space. The physical memory for Atlas was a 98-KB-word drum and 16-KB words of core. Memory was divided into 512-word pages, providing 32 frames in physical memory. An associative memory of 32 registers implemented the mapping from a virtual address to a physical address.

If a page fault occurred, a page-replacement algorithm was invoked. One memory frame was always kept empty, so that a drum transfer could start immediately. The page-replacement algorithm attempted to predict future memory-accessing behavior based on past behavior. A reference bit for each frame was set whenever the frame was accessed. The reference bits were read into memory every 1,024 instructions, and the last 32 values of these bits were retained. This history was used to define the time since the most recent reference (t_1) and the interval between the last two references (t_2). Pages were chosen for replacement in the following order:

Any page with $t_1 > t_2 + 1$. Such a page is considered to be no longer in use.

If $t_1 \leq t_2$ for all pages, then replace the page with the largest $t_2 - t_1$.

The page-replacement algorithm assumes that programs access memory in loops. If the time between the last two references is t_2 , then another reference is expected t_2 time units later. If a reference does not occur ($t_1 > t_2$), it is assumed that the page is no longer being used, and the page is replaced. If all pages are still in use, then the page that will not be needed for the longest time is replaced. The time to the next reference is expected to be $t_2 - t_1$.

23.3

The XDS-940 operating system (Lichtenberger and Pirtle [1965]) was designed at the University of California at Berkeley. Like the Atlas system, it used paging for memory management. Unlike the Atlas system, it was a time-shared system.

The paging was used only for relocation; it was not used for demand paging. The virtual memory of any user process was made up of 16-KB words, whereas the physical memory was made up of 64-KB words. Each page was made up of 2-KB words. The page table was kept in registers. Since physical memory was larger than virtual memory, several user processes could be in memory at the same time. The number of users could be increased by sharing of pages when the pages contained read-only reentrant code. Processes were kept on a drum and were swapped in and out of memory as necessary.

The XDS-940 system was constructed from a modified XDS-930. The modifications were typical of the changes made to a basic computer to allow an operating system to be written properly. A user-monitor mode was added. Certain instructions, such as I/O and halt, were defined to be privileged. An attempt to execute a privileged instruction in user mode would trap to the operating system.

A system-call instruction was added to the user-mode instruction set. This instruction was used to create new resources, such as files, allowing the operating system to manage the physical resources. Files, for example, were allocated in 256-word blocks on the drum. A bit map was used to manage free drum blocks. Each file had an index block with pointers to the actual data blocks. Index blocks were chained together.

The XDS-940 system also provided system calls to allow processes to create, start, suspend, and destroy subprocesses. A programmer could construct a system of processes. Separate processes could share memory for communication and synchronization. Process creation defined a tree structure, where a process is the root and its subprocesses are nodes below it in the tree. Each of the subprocesses could, in turn, create more subprocesses.

23.4

The THE operating system (Dijkstra [1968], McKeag and Wilson [1976]) was designed at the Technische Hogeschool at Eindhoven in the Netherlands. It was a batch system running on a Dutch computer, the EL X8, with 32 KB of 27-bit words. The system was mainly noted for its clean design, particularly its layer structure, and its use of a set of concurrent processes employing semaphores for synchronization.

Unlike the XDS-940 system, however, the set of processes in the THE system was static. The operating system itself was designed as a set of cooperating processes. In addition, five user processes were created that served as the active agents to compile, execute, and print user programs. When one job was finished, the process would return to the input queue to select another job.

A priority CPU-scheduling algorithm was used. The priorities were recomputed every 2 seconds and were inversely proportional to the amount of CPU time used recently (in the last 8 to 10 seconds). This scheme gave higher priority to I/O-bound processes and to new processes.

Memory management was limited by the lack of hardware support. However, since the system was limited and user programs could be written only in Algol, a software paging scheme was used. The Algol compiler automatically generated calls to system routines, which made sure the requested information was in memory, swapping if necessary. The backing store was a 512-KB-word drum. A 512-word page was used, with an LRU page-replacement strategy.

Another major concern of the THE system was deadlock control. The banker's algorithm was used to provide deadlock avoidance.

Closely related to the THE system is the Venus system (Liskov [1972]). The Venus system was also a layer-structured design, using semaphores to synchronize processes. The lower levels of the design were implemented in microcode, however, providing a much faster system. The memory management was

changed to a paged-segmented memory. The system was also designed as a time-sharing system, rather than a batch system.

23.5

The RC 4000 system, like the THE system, was notable primarily for its design concepts. It was designed for the Danish 4000 computer by Regnecentralen, particularly by Brinch-Hansen (Brinch-Hansen [1970], Brinch-Hansen [1973]). The objective was not to design a batch system, or a time-sharing system, or any other specific system. Rather, the goal was to create an operating-system nucleus, or kernel, on which a complete operating system could be built. Thus, the system structure was layered, and only the lower levels—comprising the kernel—were provided.

The kernel supported a collection of concurrent processes. A round-robin CPU scheduler was used. Although processes could share memory, the primary communication and synchronization mechanism was the **message system** provided by the kernel. Processes could communicate with each other by exchanging fixed-sized messages of eight words in length. All messages were stored in buffers from a common buffer pool. When a message buffer was no longer required, it was returned to the common pool.

A **message queue** was associated with each process. It contained all the messages that had been sent to that process but had not yet been received. Messages were removed from the queue in FIFO order. The system supported four primitive operations, which were executed atomically:

send-message (*in receiver, in message, out buffer*)

wait-message (*out sender, out message, out buffer*)

send-answer (*out result, in message, in buffer*)

wait-answer (*out result, out message, in buffer*)

The last two operations allowed processes to exchange several messages at a time.

These primitives required that a process service its message queue in FIFO order and that it block itself while other processes were handling its messages. To remove these restrictions, the developers provided two additional communication primitives that allowed a process to wait for the arrival of the next message or to answer and service its queue in any order:

wait-event (*in previous-buffer, out next-buffer, out result*)

get-event (*out buffer*)

I/O devices were also treated as processes. The device drivers were code that converted the device interrupts and registers into messages. Thus, a process would write to a terminal by sending that terminal a message. The device driver would receive the message and output the character to the terminal. An input character would interrupt the system and transfer to

a device driver. The device driver would create a message from the input character and send it to a waiting process.

23.6

The Compatible Time-Sharing System (CTSS) (Corbato et al. [1962]) was designed at MIT as an experimental time-sharing system. It was implemented on an IBM 7090 and eventually supported up to 32 interactive users. The users were provided with a set of interactive commands that allowed them to manipulate files and to compile and run programs through a terminal.

The 7090 had a 32-KB memory made up of 36-bit words. The monitor used 5-KB words, leaving 27 KB for the users. User memory images were swapped between memory and a fast drum. CPU scheduling employed a multilevel-feedback-queue algorithm. The time quantum for level i was $2 * i$ time units. If a program did not finish its CPU burst in one time quantum, it was moved down to the next level of the queue, giving it twice as much time. The program at the highest level (with the shortest quantum) was run first. The initial level of a program was determined by its size, so that the time quantum was at least as long as the swap time.

CTSS was extremely successful and was in use as late as 1972. Although it was limited, it succeeded in demonstrating that time sharing was a convenient and practical mode of computing. One result of CTSS was increased development of time-sharing systems. Another result was the development of MULTICS.

23.7

The MULTICS operating system (Corbato and Vyssotsky [1965], Organick [1972]) was designed at MIT as a natural extension of CTSS. CTSS and other early time-sharing systems were so successful that they created an immediate desire to proceed quickly to bigger and better systems. As larger computers became available, the designers of CTSS set out to create a time-sharing utility. Computing service would be provided like electrical power. Large computer systems would be connected by telephone wires to terminals in offices and homes throughout a city. The operating system would be a time-shared system running continuously with a vast file system of shared programs and data.

MULTICS was designed by a team from MIT, GE (which later sold its computer department to Honeywell), and Bell Laboratories (which dropped out of the project in 1969). The basic GE 635 computer was modified to a new computer system called the GE 645, mainly by the addition of paged-segmentation memory hardware.

A virtual address was composed of an 18-bit segment number and a 16-bit word offset. The segments were then paged in 1-KB-word pages. The second-chance page-replacement algorithm was used.

The segmented virtual address space was merged into the file system; each segment was a file. Segments were addressed by the name of the file. The file system itself was a multilevel tree structure, allowing users to create their own subdirectory structures.

Like CTSS, MULTICS used a multilevel feedback queue for CPU scheduling. Protection was accomplished through an access list associated with each file and a set of protection rings for executing processes. The system, which was written almost entirely in PL/1, comprised about 300,000 lines of code. It was extended to a multiprocessor system, allowing a CPU to be taken out of service for maintenance while the system continued running.

23.8 OS/360

The longest line of operating-system development is undoubtedly that of IBM computers. The early IBM computers, such as the IBM 7090 and the IBM 7094, are prime examples of the development of common I/O subroutines, followed by development of a resident monitor, privileged instructions, memory protection, and simple batch processing. These systems were developed separately, often by each site independently. As a result, IBM was faced with many different computers, with different languages and different system software.

The IBM/360 was designed to alter this situation. The IBM/360 was designed as a family of computers spanning the complete range from small business machines to large scientific machines. Only one set of software would be needed for these systems, which all used the same operating system: OS/360 (Mealy et al. [1966]). This arrangement was intended to reduce maintenance problems for IBM and to allow users to move programs and applications freely from one IBM system to another.

Unfortunately, OS/360 tried to be all things for all people. As a result, it did none of its tasks especially well. The file system included a type field that defined the type of each file, and different file types were defined for fixed-length and variable-length records and for blocked and unblocked files. Contiguous allocation was used, so the user had to guess the size of each output file. The Job Control Language (JCL) added parameters for every possible option, making it incomprehensible to the average user.

The memory-management routines were hampered by the architecture. Although a base-register addressing mode was used, the program could access and modify the base register, so that absolute addresses were generated by the CPU. This arrangement prevented dynamic relocation; the program was bound to physical memory at load time. Two separate versions of the operating system were produced: OS/MFT used fixed regions and OS/MVT used variable regions.

The system was written in assembly language by thousands of programmers, resulting in millions of lines of code. The operating system itself required large amounts of memory for its code and tables. Operating-system overhead often consumed one-half of the total CPU cycles. Over the years, new versions were released to add new features and to fix errors. However, fixing one error often caused another in some remote part of the system, so that the number of known errors in the system remained fairly constant.

Virtual memory was added to OS/360 with the change to the IBM 370 architecture. The underlying hardware provided a segmented-paged virtual memory. New versions of OS used this hardware in different ways. OS/VS1 created one large virtual address space and ran OS/MFT in that virtual memory. Thus, the operating system itself was paged, as well as user programs. OS/VS2

Release 1 ran OS/MVT in virtual memory. Finally, OS/VS2 Release 2, which is now called MVS, provided each user with his own virtual memory.

MVS is still basically a batch operating system. The CTSS system was run on an IBM 7094, but MIT decided that the address space of the 360, IBM's successor to the 7094, was too small for MULTICS, so they switched vendors. IBM then decided to create its own time-sharing system, TSS/360 (Lett and Konigsford [1968]). Like MULTICS, TSS/360 was supposed to be a large, time-shared utility. The basic 360 architecture was modified in the model 67 to provide virtual memory. Several sites purchased the 360/67 in anticipation of TSS/360.

TSS/360 was delayed, however, so other time-sharing systems were developed as temporary systems until TSS/360 was available. A time-sharing option (TSO) was added to OS/360. IBM's Cambridge Scientific Center developed CMS as a single-user system and CP/67 to provide a virtual machine to run it on (Meyer and Seawright [1970], Parmelee et al. [1972]).

When TSS/360 was eventually delivered, it was a failure. It was too large and too slow. As a result, no site would switch from its temporary system to TSS/360. Today, time sharing on IBM systems is largely provided either by TSO under MVS or by CMS under CP/67 (renamed VM).

Both TSS/360 and MULTICS did not achieve commercial success. What went wrong with these systems? Part of the problem was that these advanced systems were too large and too complex to be understood. Another problem was the assumption that computing power would be available from a large, remote computer. It now appears that most computing will be done by small individual machines—personal computers—not by large, remote, time-shared systems that try to be all things to all users.

23.9 Mach

The Mach operating system traces its ancestry to the Accent operating system developed at Carnegie Mellon University (CMU) (Rashid and Robertson [1981]). Mach's communication system and philosophy are derived from Accent, but many other significant portions of the system (for example, the virtual memory system, task and thread management) were developed from scratch (Rashid [1986], Tevanian et al. [1989], and Accetta et al. [1986]). The Mach scheduler was described in detail by Tevanian et al. [1987a] and Black [1990]. An early version of the Mach shared memory and memory-mapping system was presented by Tevanian et al. [1987b].

The Mach operating system was designed with the following three critical goals in mind:

- Emulate 4.3BSD UNIX so that the executable files from a UNIX system can run correctly under Mach.
- Be a modern operating system that supports many memory models, as well as parallel and distributed computing.
- Have a kernel that is simpler and easier to modify than is 4.3BSD.

Mach's development followed an evolutionary path from BSD UNIX systems. Mach code was initially developed inside the 4.2BSD kernel, with BSD

kernel components replaced by Mach components as the Mach components were completed. The BSD components were updated to 4.3BSD when that became available. By 1986, the virtual memory and communication subsystems were running on the DEC VAX computer family, including multiprocessor versions of the VAX. Versions for the IBM RT/PC and for SUN 3 workstations followed shortly. Then, 1987 saw the completion of the Encore Multimax and Sequent Balance multiprocessor versions, including task and thread support, as well as the first official releases of the system, Release 0 and Release 1.

Through Release 2, Mach provided compatibility with the corresponding BSD systems by including much of BSD's code in the kernel. The new features and capabilities of Mach made the kernels in these releases larger than the corresponding BSD kernels. Mach 3 moved the BSD code outside of the kernel, leaving a much smaller microkernel. This system implements only basic Mach features in the kernel; all UNIX-specific code has been evicted to run in user-mode servers. Excluding UNIX-specific code from the kernel allows the replacement of BSD with another operating system or the simultaneous execution of multiple operating-system interfaces on top of the microkernel. In addition to BSD, user-mode implementations have been developed for DOS, the Macintosh operating system, and OSF/1. This approach has similarities to the virtual machine concept, but here the virtual machine is defined by software (the Mach kernel interface), rather than by hardware. With Release 3.0, Mach became available on a wide variety of systems, including single-processor SUN, Intel, IBM, and DEC machines and multiprocessor DEC, Sequent, and Encore systems.

Mach was propelled into the forefront of industry attention when the Open Software Foundation (OSF) announced in 1989 that it would use Mach 2.5 as the basis for its new operating system, OSF/1. The initial release of OSF/1 occurred a year later, and this system competed with UNIX System V, Release 4, the operating system of choice at that time among UNIX International (UI) members. OSF members included key technological companies such as IBM, DEC, and HP. OSF has since changed its direction, and only DEC UNIX is based on the Mach kernel.

Mach 2.5 is also the basis for the operating system on the NeXT workstation, the brainchild of Steve Jobs, of Apple Computer fame.

Unlike UNIX, which was developed without regard for multiprocessing, Mach incorporates multiprocessing support throughout. Its multiprocessing support is also exceedingly flexible, ranging from shared-memory systems to systems with no memory shared between processors. Mach uses lightweight processes, in the form of multiple threads of execution within one task (or address space), to support multiprocessing and parallel computation. Its extensive use of messages as the only communication method ensures that protection mechanisms are complete and efficient. By integrating messages with the virtual memory system, Mach also ensures that messages can be handled efficiently. Finally, by having the virtual memory system use messages to communicate with the daemons managing the backing store, Mach provides great flexibility in the design and implementation of these memory-object-managing tasks. By providing low-level, or primitive, system calls from which more complex functions can be built, Mach reduces the size of the kernel while permitting operating-system emulation at the user level, much like IBM's virtual-machine systems.

Previous editions of *Operating System Concepts* included an entire chapter on Mach. This chapter, as it appeared in the fourth edition, is available on the Web (<http://www.os-book.com>).

23.10

There are, of course, other operating systems, and most of them have interesting properties. The MCP operating system for the Burroughs computer family (McKeag and Wilson [1976]) was the first to be written in a system-programming language. It supported segmentation and multiple CPUs. The SCOPE operating system for the CDC 6600 (McKeag and Wilson [1976]) was also a multi-CPU system. The coordination and synchronization of the multiple processes were surprisingly well designed. Tenex (Bobrow et al. [1972]) was an early demand-paging system for the PDP-10 that has had a great influence on subsequent time-sharing systems, such as TOPS-20 for the DEC-20. The VMS operating system for the VAX is based on the RSX operating system for the PDP-11. CP/M was the most common operating system for 8-bit microcomputer systems, few of which exist today; MS-DOS is the most common system for 16-bit microcomputers. Graphical user interfaces (GUIs) have become popular to make computers easier to use; the Macintosh Operating System and Microsoft Windows are the two leaders in this area.

- 23.1 Discuss what considerations the computer operator took into account in deciding in the sequences in which programs would be run on early computer systems that were manually operated.
- 23.2 What optimizations were used to minimize the discrepancy between CPU and I/O speeds on early computer systems?
- 23.3 Consider the page replacement algorithm used by Atlas. In what ways is it different from the clock algorithm discussed in Section 9.4.5.2?
- 23.4 Consider the multilevel feedback queue used by CTSS and MULTICS. Suppose a program consistently uses seven time units every time it is scheduled before it performs an I/O operation and blocks. How many time units are allocated to this program when it is scheduled for execution at different points in time?
- 23.5 What are the implications of supporting BSD functionality in user-mode servers within the Mach operating system?

