Appendix B

Windows 7

Updated by Dave Probert

The Microsoft Windows 7 operating system is a 32-/64-bit preemptive multitasking client operating system for microprocessors implementing the Intel IA-32 and AMD64 instruction set architectures (ISAs). Microsoft's corresponding server operating system, Windows Server 2008 R2, is based on the same code as Windows 7 but supports only the 64-bit AMD64 and IA64 (Itanium) ISAs. Windows 7 is the latest in a series of Microsoft operating systems based on its NT code, which replaced the earlier systems based on Windows 95/98. In this appendix, we discuss the key goals of Windows 7, the layered architecture of the system that has made it so easy to use, the file system, the networking features, and the programming interface.

CHAPTER OBJECTIVES

- Explore the principles underlying Windows 7's design and the specific components of the system.
- Provide a detailed discussion of the Windows 7 file system.
- Illustrate the networking protocols supported in Windows 7.
- Describe the interface available in Windows 7 to system and application programmers.
- Describe the important algorithms implemented with Windows 7.

B.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 end the joint effort with IBM and develop its own "new technology" (or NT) portable operating system to support 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 Microsoft's new operating system.

Originally, the team planned to use the OS/2 API as NT's native environment, but during development, NT was changed to use a new 32-bit Windows API (called Win32), based on the popular 16-bit API used in 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, supported only Intel (and compatible) processors due to marketplace factors. Windows 2000 incorporated significant changes. 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 edition of Windows XP became available (called Windows .Net Server). Windows XP updated the graphical user interface (GUI) with a visual design that took advantage of more recent hardware advances and many new ***ease-of-use features***. Numerous features were added to automatically repair problems in applications and the operating system itself. As a result of these changes, Windows XP provided better networking and device experience (including zero-configuration wireless, instant messaging, streaming media, and digital photography/video), dramatic performance improvements for both the desktop and large multiprocessors, and better reliability and security than earlier Windows operating systems.

The long-awaited update to Windows XP, called Windows Vista, was released in November 2006, but it was not well received. Although Windows Vista included many improvements that later showed up in Windows 7, these improvements were overshadowed by Windows Vista's perceived sluggishness and compatibility problems. Microsoft responded to criticisms of Windows Vista by improving its engineering processes and working more closely with the makers of Windows hardware and applications. The result was **Windows 7**, which was released in October 2009, along with corresponding server editions of Windows. Among the significant engineering changes is the increased use of **execution tracing** rather than counters or profiling to analyze system behavior. Tracing runs constantly in the system, watching hundreds of scenarios execute. When one of these scenarios fails, or when it succeeds but does not perform well, the traces can be analyzed to determine the cause.

Windows 7 uses a client–server architecture (like Mach) to implement two operating-system personalities, Win32 and POSIX, with user-level processes called subsystems. (At one time, Windows also supported an OS/2 subsystem, but it was removed in Windows XP due to the demise of OS/2.) The subsystem architecture allows enhancements to be made to one operating-system personality without affecting the application compatibility of the other. Although the POSIX subsystem continues to be available for Windows 7, the Win32 API has become very popular, and the POSIX APIs are used by only a few sites. The subsystem approach continues to be interesting to study from an operating-system

perspective, but machine-virtualization technologies are now becoming the dominant way of running multiple operating systems on a single machine.

Windows 7 is a multiuser operating system, supporting simultaneous access through distributed services or through multiple instances of the GUI via the Windows terminal services. The server editions of Windows 7 support simultaneous terminal server sessions from Windows desktop systems. The desktop editions 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 log on.

We noted earlier that some GUI implementation moved into kernel mode in Windows NT 4.0. It started to move into user mode again with Windows Vista, which included the **desktop window manager** (**DWM**) as a user-mode process. DWM implements the desktop compositing of Windows, providing the Windows ***Aero*** interface look on top of the Windows DirectX graphic software. DirectX continues to run in the kernel, as does the code implementing Windows' previous windowing and graphics models (Win32k and GDI). Windows 7 made substantial changes to the DWM, significantly reducing its memory footprint and improving its performance.

Windows XP was the first version of Windows to ship a 64-bit version (for the IA64 in 2001 and the AMD64 in 2005). Internally, 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 was support for large virtual addresses. However, 64-bit editions of Windows also support much larger physical memories. By the time Windows 7 shipped, the AMD64 ISA had become available on almost all CPUs from both Intel and AMD. In addition, by that time, physical memories on client systems frequently exceeded the 4-GB limit of the IA-32. As a result, the 64-bit version of Windows 7 is now commonly installed on larger client systems. Because the AMD64 architecture supports high-fidelity IA-32 compatibility at the level of individual processes, 32- and 64-bit applications can be freely mixed in a single system.

In the rest of our description of Windows 7, we will not distinguish between the client editions of Windows 7 and the corresponding server editions. They are based on the same core components and run the same binary files for the kernel and most drivers. Similarly, although Microsoft ships a variety of different editions of each release to address different market price points, few of the differences between editions are reflected in the core of the system. In this chapter, we focus primarily on the core components of Windows 7.

B.2 Design Principles

Microsoft's design goals for Windows included security, reliability, Windows and POSIX application compatibility, high performance, extensibility, portability, and international support. Some additional goals, energy efficiency and dynamic device support, have recently been added to this list. Next, we discuss each of these goals and how it is achieved in Windows 7.

B.2.1 Security

Windows 7 security goals required more than just adherence to the design standards that had enabled Windows NT 4.0 to receive a C2 security classification

from the U.S. government. (A C2 classification signifies a moderate level of protection from defective software and malicious attacks. Classifications were defined by the Department of Defense Trusted Computer System Evaluation Criteria, also known as the **Orange Book**.) Extensive code review and testing were combined with sophisticated automatic analysis tools to identify and investigate potential defects that might represent security vulnerabilities.

Windows bases security on discretionary access controls. System objects, including files, registry settings, and kernel objects, are protected by **access-control lists** (**ACLs**) (see Section 13.4.2). ACLs are vulnerable to user and programmer errors, however, as well as to the most common attacks on consumer systems, in which the user is tricked into running code, often while browsing the web. Windows 7 includes a mechanism called **integrity levels** that acts as a rudimentary ***capability*** system for controlling access. Objects and processes are marked as having low, medium, or high integrity. Windows does not allow a process to modify an object with a higher integrity level, no matter what the setting of the ACL.

Other security measures include **address-space layout randomization** (**ASLR**), nonexecutable stacks and heaps, and encryption and **digital signature** facilities. ASLR thwarts many forms of attack by preventing small amounts of injected code from jumping easily to code that is already loaded in a process as part of normal operation. This safeguard makes it likely that a system under attack will fail or crash rather than let the attacking code take control.

Recent chips from both Intel and AMD are based on the AMD64 architecture, which allows memory pages to be marked so that they cannot contain executable instruction code. Windows tries to mark stacks and memory heaps so that they cannot be used to execute code, thus preventing attacks in which a program bug allows a buffer to overflow and then is tricked into executing the contents of the buffer. This technique cannot be applied to all programs, because some rely on modifying data and executing it. A column labeled "data execution prevention" in the Windows task manager shows which processes are marked to prevent these attacks.

Windows uses encryption as part of common protocols, such as those used to communicate securely with websites. Encryption is also used to protect user files stored on disk from prying eyes. Windows 7 allows users to easily encrypt virtually a whole disk, as well as removable storage devices such as USB flash drives, with a feature called BitLocker. If a computer with an encrypted disk is stolen, the thieves will need very sophisticated technology (such as an electron microscope) to gain access to any of the computer's files. Windows uses digital signatures to ***sign*** operating system binaries so it can verify that the files were produced by Microsoft or another known company. In some editions of Windows, a **code integrity** module is activated at boot to ensure that all the loaded modules in the kernel have valid signatures, assuring that they have not been tampered with by an off-line attack.

B.2.2 Reliability

Windows matured greatly as an operating system in its first ten years, leading to Windows 2000. At the same time, its reliability increased due to such factors as maturity in the source code, extensive stress testing of the system, improved CPU architectures, and automatic detection of many serious errors in drivers from both Microsoft and third parties. Windows has subsequently extended

the tools for achieving reliability to include automatic analysis of source code for errors, tests that include providing invalid or unexpected input parameters (known as **fuzzing**) to detect validation failures, and an application version of the driver verifier that applies dynamic checking for an extensive set of common user-mode programming errors. Other improvements in reliability have resulted from moving more code out of the kernel and into user-mode services. Windows provides extensive support for writing drivers in user mode. System facilities that were once in the kernel and are now in user mode include the Desktop Window Manager and much of the software stack for audio.

One of the most significant improvements in the Windows experience came from adding memory diagnostics as an option at boot time. This addition is especially valuable because so few consumer PCs have error-correcting memory. When bad RAM starts to drop bits here and there, the result is frustratingly erratic behavior in the system. The availability of memory diagnostics has greatly reduced the stress levels of users with bad RAM.

Windows 7 introduced a fault-tolerant memory heap. The heap learns from application crashes and automatically inserts mitigations into future execution of an application that has crashed. This makes the application more reliable even if it contains common bugs such as using memory after freeing it or accessing past the end of the allocation.

Achieving high reliability in Windows is particularly challenging because almost one billion computers run Windows. Even reliability problems that affect only a small percentage of users still impact tremendous numbers of human beings. The complexity of the Windows ecosystem also adds to the challenges. Millions of instances of applications, drivers, and other software are being constantly downloaded and run on Windows systems. Of course, there is also a constant stream of malware attacks. As Windows itself has become harder to attack directly, exploits increasingly target popular applications.

To cope with these challenges, Microsoft is increasingly relying on communications from customer machines to collect large amounts of data from the ecosystem. Machines can be sampled to see how they are performing, what software they are running, and what problems they are encountering. Customers can send data to Microsoft when systems or software crashes or hangs. This constant stream of data from customer machines is collected very carefully, with the users' consent and without invading privacy. The result is that Microsoft is building an ever-improving picture of what is happening in the Windows ecosystem that allows continuous improvements through software updates, as well as providing data to guide future releases of Windows.

B.2.3 Windows and POSIX Application Compatibility

As mentioned, Windows XP was both an update of Windows 2000 and a replacement for Windows 95/98. Windows 2000 focused primarily on compatibility for business applications. The requirements for Windows XP included a much greater compatibility with the consumer applications that ran on Windows 95/98. Application compatibility is difficult to achieve because many applications check for a particular version of Windows, may depend to some extent on the quirks of the implementation of APIs, may have latent application bugs that were masked in the previous system, and so forth. Applications may

also have been compiled for a different instruction set. Windows 7 implements several strategies to run applications despite incompatibilities.

Like Windows XP, Windows 7 has a compatibility layer that sits between applications and the Win32 APIs. This layer makes Windows 7 look (almost) bug-for-bug compatible with previous versions of Windows. Windows 7, 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 7 provides a thunking layer that translates 32-bit API calls into native 64-bit calls.

The Windows subsystem model allows multiple operating-system personalities to be supported. As noted earlier, although the API most commonly used with Windows is the Win32 API, some editions of Windows 7 support a POSIX subsystem. POSIX is a standard specification for UNIX that allows most available UNIX-compatible software to compile and run without modification.

As a final compatibility measure, several editions of Windows 7 provide a virtual machine that runs Windows XP inside Windows 7. This allows applications to get bug-for-bug compatibility with Windows XP.

B.2.4 High Performance

Windows was 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 performance and cache-line management are keys to scalability). To satisfy performance requirements, NT used a variety of techniques, such as asynchronous I/O, optimized protocols for networks, kernel-based graphics rendering, and sophisticated caching of file-system data. The memory-management and synchronization algorithms were designed with an awareness of the performance considerations related to cache lines and multiprocessors.

Windows NT was designed for symmetrical multiprocessing (SMP); on a multiprocessor computer, several threads can run at the same time, even in the kernel. On each CPU, Windows NT uses priority-based preemptive scheduling of threads. Except while executing in the kernel dispatcher or at interrupt level, threads in any process running in Windows can be preempted by higher-priority threads. Thus, the system responds quickly (see Chapter 5).

The subsystems that constitute Windows NT communicate with one another efficiently through a **local procedure call** (**LPC**) facility that provides high-performance message passing. When a thread requests a synchronous service from another process through an LPC, the servicing thread is marked ***ready***, and its priority is temporarily boosted to avoid the scheduling delays that would occur if it had to wait for threads already in the queue.

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

By the time Windows 7 was developed, several major changes had come to computing. Client/server computing had increased in importance, so an advanced local procedure call (ALPC) facility was introduced to provide higher performance and more reliability than LPC. The number of CPUs and the amount of physical memory available in the largest multiprocessors had increased substantially, so quite a lot of effort was put into improving operating-system scalability.

The implementation of SMP in Windows NT used bitmasks to represent collections of processors and to identify, for example, which set of processors a particular thread could be scheduled on. These bitmasks were defined as fitting within a single word of memory, limiting the number of processors supported within a system to 64. Windows 7 added the concept of **processor groups** to represent arbitrary numbers of CPUs, thus accommodating more CPU cores. The number of CPU cores within single systems has continued to increase not only because of more cores but also because of cores that support more than one logical thread of execution at a time.

All these additional CPUs created a great deal of contention for the locks used for scheduling CPUs and memory. Windows 7 broke these locks apart. For example, before Windows 7, a single lock was used by the Windows scheduler to synchronize access to the queues containing threads waiting for events. In Windows 7, each object has its own lock, allowing the queues to be accessed concurrently. Also, many execution paths in the scheduler were rewritten to be lock-free. This change resulted in good scalability performance for Windows even on systems with 256 hardware threads.

Other changes are due to the increasing importance of support for parallel computing. For years, the computer industry has been dominated by Moore's Law, leading to higher densities of transistors that manifest themselves as faster clock rates for each CPU. Moore's Law continues to hold true, but limits have been reached that prevent CPU clock rates from increasing further. Instead, transistors are being used to build more and more CPUs into each chip. New programming models for achieving parallel execution, such as Microsoft's Concurrency RunTime (ConcRT) and Intel's Threading Building Blocks (TBB), are being used to express parallelism in C++ programs. Where Moore's Law has governed computing for forty years, it now seems that Amdahl's Law, which governs parallel computing, will rule the future.

To support task-based parallelism, Windows 7 provides a new form of **user-mode scheduling** (**UMS**). UMS allows programs to be decomposed into tasks, and the tasks are then scheduled on the available CPUs by a scheduler that operates in user mode rather than in the kernel.

The advent of multiple CPUs on the smallest computers is only part of the shift taking place to parallel computing. Graphics processing units (GPUs) accelerate the computational algorithms needed for graphics by using **SIMD** architectures to execute a single instruction for multiple data at the same time. This has given rise to the use of GPUs for general computing, not just graphics. Operating-system support for software like OpenCL and CUDA is allowing programs to take advantage of the GPUs. Windows supports use of GPUs through software in its DirectX graphics support. This software, called DirectCompute, allows programs to specify **computational kernels** using the same HLSL (high-level shader language) programming model used to program the SIMD hardware for **graphics shaders**. The computational kernels run very

quickly on the GPU and return their results to the main computation running on the CPU.

B.2.5 Extensibility

Extensibility refers to the capacity of an operating system to keep up with advances in computing technology. To facilitate change over time, the developers implemented Windows using a layered architecture. The Windows executive runs in kernel mode and provides the basic system services and abstractions that support shared use of the system. 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 the Win32 APIs and POSIX all run on Windows in the appropriate environment. Because of the modular structure, additional environmental subsystems can be added without affecting the executive. In addition, Windows 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 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.

B.2.6 Portability

An operating system is **portable** if it can be moved from one CPU architecture to another with few changes. Windows was designed to be portable. Like the UNIX operating system, Windows is written primarily in C and C++. The architecture-specific source code is relatively small, and there is very little use of assembly code. Porting Windows to a new architecture mostly affects the Windows kernel, since the user-mode code in Windows is almost exclusively written to be architecture independent. To port Windows, the kernel's architecture-specific code must be ported, and sometimes conditional compilation is needed in other parts of the kernel because of changes in major data structures, such as the page-table format. The entire Windows system must then be recompiled for the new CPU instruction set.

Operating systems are sensitive not only to CPU architecture but also to CPU support chips and hardware boot programs. The CPU and support chips are collectively known as a **chipset**. These chipsets and the associated boot code determine how interrupts are delivered, describe the physical characteristics of each system, and provide interfaces to deeper aspects of the CPU architecture, such as error recovery and power management. It would be burdensome to have to port Windows to each type of support chip as well as to each CPU architecture. Instead, Windows isolates most of the chipset-dependent code in a dynamic link library (DLL), called the **hardware-abstraction layer** (**HAL**), that is loaded with the kernel. The Windows kernel depends on the HAL interfaces rather than on the underlying chipset details. This allows the single set of kernel and driver binaries for a particular CPU to be used with different chipsets simply by loading a different version of the HAL.

Over the years, Windows has been ported to a number of different CPU architectures: Intel IA-32-compatible 32-bit CPUs, AMD64-compatible and IA64 64-bit CPUs, the DEC Alpha, and the MIPS and PowerPC CPUs. Most of these CPU architectures failed in the market. When Windows 7 shipped, only the IA-

32 and AMD64 architectures were supported on client computers, along with AMD64 and IA64 on servers.

B.2.7 International Support

Windows was 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 national customs. String comparisons are specialized to account for varying character sets. UNICODE is Windows's native character code. Windows 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.

B.2.8 Energy Efficiency

Increasing energy efficiency for computers causes batteries to last longer for laptops and netbooks, saves significant operating costs for power and cooling of data centers, and contributes to green initiatives aimed at lowering energy consumption by businesses and consumers. For some time, Windows has implemented several strategies for decreasing energy use. The CPUs are moved to lower power states—for example, by lowering clock frequency—whenever possible. In addition, when a computer is not being actively used, Windows may put the entire computer into a low-power state (sleep) or may even save all of memory to disk and shut the computer off (hibernation). When the user returns, the computer powers up and continues from its previous state, so the user does not need to reboot and restart applications.

Windows 7 added some new strategies for saving energy. The longer a CPU can stay unused, the more energy can be saved. Because computers are so much faster than human beings, a lot of energy can be saved just while humans are thinking. The problem is that too many programs are constantly polling to see what is happening in the system. A swarm of software timers are firing, keeping the CPU from staying idle long enough to save much energy. Windows 7 extends CPU idle time by skipping clock ticks, coalescing software timers into smaller numbers of events, and "parking" entire CPUs when systems are not heavily loaded.

B.2.9 Dynamic Device Support

Early in the history of the PC industry, computer configurations were fairly static. Occasionally, new devices might be plugged into the serial, printer, or game ports on the back of a computer, but that was it. The next steps toward dynamic configuration of PCs were laptop docks and PCMIA cards. A PC could suddenly be connected to or disconnected from a whole set of peripherals. In a contemporary PC, the situation has completely changed. PCs are designed to let users to plug and unplug a huge host of peripherals all the time; external disks, thumb drives, cameras, and the like are constantly coming and going.

Support for dynamic configuration of devices is continually evolving in Windows. The system can automatically recognize devices when they are

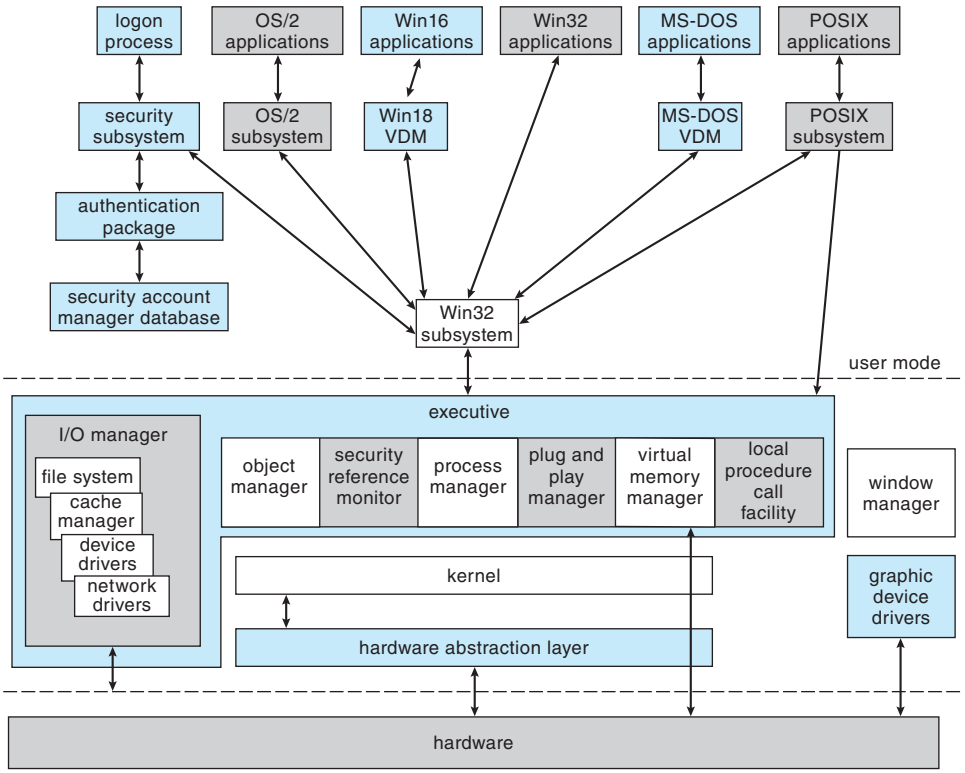


Figure B.1 Windows block diagram.

plugged in and can find, install, and load the appropriate drivers—often without user intervention. When devices are unplugged, the drivers automatically unload, and system execution continues without disrupting other software.

B.3 System Components

The architecture of Windows is a layered system of modules, as shown in Figure B.1. The main layers are the HAL, the kernel, and the executive, all of which run in kernel 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.

B.3.1 Hardware-Abstraction Layer

The HAL is the layer of software that hides hardware chipset differences from upper levels of the operating system. The HAL exports a virtual hardware interface that is used by the kernel dispatcher, the executive, and the device drivers. Only a single version of each device driver is required for

each CPU architecture, no matter what support chips might be present. Device drivers map devices and access them directly, but the chipset-specific details of mapping memory, configuring I/O buses, setting up DMA, and coping with motherboard-specific facilities are all provided by the HAL interfaces.

B.3.2 Kernel

The kernel layer of Windows has four main responsibilities: thread scheduling, low-level processor synchronization, interrupt and exception handling, and switching between user mode and kernel mode. The kernel is implemented in the C language, using assembly language only where absolutely necessary to interface with the lowest level of the hardware architecture.

The kernel is organized according to object-oriented design principles. An **object type** in Windows 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.

B.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 and context switching, implementation of synchronization primitives, timer management, software interrupts (asynchronous and deferred procedure calls), and exception dispatching.

B.3.2.2 Threads and Scheduling

Like many other modern operating systems, Windows uses processes and threads for executable code. Each process has one or more threads, and each thread has its own scheduling state, including actual priority, processor affinity, and CPU usage information.

There are six possible thread states: `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 processor 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 waits 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 thread is in the `transition` state while it waits for resources necessary for execution; for example, it may be waiting for its kernel stack to be swapped in from disk. 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 1 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***. Priority class 0 is reserved for 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. Queuing the DPC results in a software interrupt when the processor returns to normal interrupt priority. The software interrupt causes the dispatcher to reschedule the processor to execute the next available thread at the preempted thread's priority level.

The priority of the preempted thread may be modified before it is placed back on the dispatcher queues. 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 versus I/O-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. 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 thread becomes ready while a lower-priority thread is running, the lower-priority thread is preempted. This preemption gives the higher-priority thread preferential access to the CPU. Windows 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; threads are blocked indefinitely while DPCs and **interrupt service routines** (**ISRs**) are running (as discussed further below).

Traditionally, operating-system schedulers used sampling to measure CPU utilization by threads. The system timer would fire periodically, and the timer interrupt handler would take note of what thread was currently scheduled and whether it was executing in user or kernel mode when the interrupt occurred. This sampling technique was necessary because either the CPU did not have a high-resolution clock or the clock was too expensive or unreliable to access frequently. Although efficient, sampling was inaccurate and led to anomalies such as incorporating interrupt servicing time as thread time and dispatching threads that had run for only a fraction of the quantum. Starting with Windows Vista, CPU time in Windows has been tracked using the hardware **timestamp counter** (**TSC**) included in recent processors. Using the TSC results in more accurate accounting of CPU usage, and the scheduler will not preempt threads before they have run for a full quantum.

B.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 include the following:

- The **event object** is used to record an event occurrence and to synchronize this occurrence 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 associated with the notion of ownership.
- The **mutex**, available only in kernel mode, provides deadlock-free mutual exclusion.
- The **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. It is associated with a **process object**, which encapsulates a virtual address space. The thread object is signaled when the thread exits, and the process object, when the process exits.
- The **timer object** is 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 or waits on handles to synchronize with other threads as well as with the operating system (see Section B.7.1).

B.3.2.4 Software Interrupts: Asynchronous and Deferred Procedure Calls

The dispatcher implements two types of software interrupts: **asynchronous procedure calls** (**APCs**) and deferred procedure calls (DPCs, mentioned earlier). An asynchronous procedure call breaks into an executing thread and calls a procedure. APCs are used to begin execution of new threads, suspend or resume existing threads, terminate threads or processes, deliver notification that an asynchronous I/O has completed, and extract the contents of the CPU registers from a running thread. APCs are queued to specific threads and allow the system to execute both system and user code within a process's context. User-mode execution of an APC cannot occur at arbitrary times, but only when the thread is waiting in the kernel and marked ***alertable***.

DPCsare used to postpone interrupt processing. After handling all urgent device-interrupt processing, the ISR schedules the remaining processing by queuing a DPC. The associated software interrupt will not occur until the CPU is next at a priority lower than the priority of all I/O device interrupts but higher than the priority at which threads run. Thus, DPCs do not block other device 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 completion of DPC routines does not take an extended amount of time. As an alternative, the dispatcher maintains a pool of worker threads. ISRs and DPCs may queue work items to the worker threads where they will be executed using normal thread scheduling. DPC routines are restricted so that they cannot take page faults (be paged out of memory), call system services, or take any other action that might result in an attempt to wait for a dispatcher object to be signaled. Unlike APCs, DPC routines make no assumptions about what process context the processor is executing.

B.3.2.5 Exceptions and Interrupts

The kernel dispatcher also provides trap handling for exceptions and interrupts generated by hardware or software. Windows 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. (For details on ports,

see Section B.3.3.4.) 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 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 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.

When Windows fails to handle an exception, it may construct a description of the error that occurred and request permission from the user to send the information back to Microsoft for further analysis. In some cases, Microsoft's automated analysis may be able to recognize the error immediately and suggest a fix or workaround.

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 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. Eight are reserved for use by the kernel; the remaining 24 represent hardware interrupts via the HAL (although most IA-32 systems use only 16). The Windows interrupts are defined in Figure B.2.

The kernel uses an **interrupt-dispatch table** to bind each interrupt level to a service routine. In a multiprocessor computer, Windows keeps a separate interrupt-dispatch table (IDT) 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

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 B.2 Windows interrupt-request levels.

by a kernel-level thread or by an ISR returning from interrupt processing. Windows 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 execution, and to handle timers.

B.3.2.6 Switching between User-Mode and Kernel-Mode Threads

What the programmer thinks of as a thread in traditional Windows is actually two threads: a **user-mode thread** (**UT**) and a **kernel-mode thread** (**KT**). Each has its own stack, register values, and execution context. A UT requests a system service by executing an instruction that causes a trap to kernel mode. The kernel layer runs a trap handler that switches between the UT and the corresponding KT. When a KT has completed its kernel execution and is ready to switch back to the corresponding UT, the kernel layer is called to make the switch to the UT, which continues its execution in user mode.

Windows 7 modifies the behavior of the kernel layer to support user-mode scheduling of the UTs. User-mode schedulers in Windows 7 support cooperative scheduling. A UT can explicitly yield to another UT by calling the user-mode scheduler; it is not necessary to enter the kernel. User-mode scheduling is explained in more detail in Section B.7.3.7.

B.3.3 Executive

The Windows executive provides a set of services that all environmental subsystems use. The services are grouped as follows: object manager, virtual memory manager, process manager, advanced local procedure call facility, I/O manager, cache manager, security reference monitor, plug-and-play and power managers, registry, and booting.

B.3.3.1 Object Manager

For managing kernel-mode entities, Windows uses a generic set of interfaces that are manipulated by user-mode programs. Windows calls these entities ***objects***, and the executive component that manipulates them is the **object manager**. Examples of objects are semaphores, mutexes, events, processes, and threads; all these are ***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. Local-communication endpoints are implemented as port objects.

User-mode code accesses these objects using an opaque value called a **handle**, which is returned by many APIs. Each process has a **handle table** containing entries that track the objects used by the process. The **system process**, which contains the kernel, has its own handle table, which is protected from user code. The handle tables in Windows are represented by a tree structure, which can expand from holding 1,024 handles to holding over 16 million. Kernel-mode code can access an object by using either a handle or a **referenced pointer**.

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 the parent process. When a process exits, all its open handles are implicitly closed. 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.

The object manager keeps track of two counts for each object: the number of handles for the object and the number of referenced pointers. The handle count is the number of handles that refer to the object in the handle tables of all processes, including the system process that contains the kernel. The referenced pointer count is incremented whenever a new pointer is needed by the kernel and decremented when the kernel is done with the pointer. The purpose of these reference counts is to ensure that an object is not freed while it is still referenced by either a handle or an internal kernel pointer.

The object manager maintains the Windows internal name space. In contrast to UNIX, which roots the system name space in the file system, Windows uses an abstract name space and connects the file systems as devices. Whether a Windows object has a name is up to its creator. Processes and threads are created without names and referenced either by handle or through a separate numerical identifier. Synchronization events usually have names, so that they can be opened by unrelated processes. 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. The object manager supports directories and symbolic links in the name space. As an example, MS-DOS drive letters are implemented using symbolic links; `\Global??\C:` is a symbolic link to the device object `\Device\HarddiskVolume2`, representing a mounted file-system volume in the `\Device` directory.

Each object, as mentioned earlier, is an instance of an ***object type***. The object type specifies how instances are to be allocated, how the data fields are to be defined, and how the standard set of virtual functions used for all objects are to be implemented. The standard functions implement operations such as mapping names to objects, closing and deleting, and applying security checks. Functions that are specific to a particular type of object are implemented by system services designed to operate on that particular object type, not by the methods specified in the object type.

The `parse()` function is the most interesting of the standard object functions. It allows the implementation of an object. The file systems, the registry configuration store, and GUI objects are the most notable users of parse functions to extend the Windows name space.

Returning to our Windows naming example, device objects used to represent file-system volumes provide a parse function. This allows a name like `\Global??\C:\foo\bar.doc` to be interpreted as the file `\foo\bar.doc` on the volume represented by the device object `HarddiskVolume2`. We can illustrate how naming, parse functions, objects, and handles work together by looking at the steps to open the file in Windows:

1. An application requests that a file named `C:\foo\bar.doc` be opened.
2. The object manager finds the device object `HarddiskVolume2`, looks up the parse procedure `IopParseDevice` from the object's type, and invokes it with the file's name relative to the root of the file system.
3. `IopParseDevice()` allocates a file object and passes it to the file system, which fills in the details of how to access `C:\foo\bar.doc` on the volume.
4. When the file system returns, `IopParseDevice()` allocates an entry for the file object in the handle table for the current process and returns the handle to the application.

If the file cannot successfully be opened, `IopParseDevice()` deletes the file object it allocated and returns an error indication to the application.

B.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 uses a page-based management scheme with page sizes of 4 KB and 2 MB on AMD64 and IA-32-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. A page can also be marked zero-fill-on-demand, which initializes the page with zeros before it is allocated, thus erasing the previous contents.

On IA-32 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 in kernel mode to access the operating-system code and data structures. For the AMD64 architecture, Windows provides a 8-TB virtual address space for user mode out of the 16 EB supported by existing hardware for each process.

Key areas of the kernel-mode region that are not identical for all processes are the self-map, hyperspace, and session space. The hardware references a process's page table using physical page-frame numbers, and the **page table self-map** makes the contents of the process's page table accessible using virtual addresses. **Hyperspace** maps the current process's working-set information into the kernel-mode address space. **Session space** is used to share an instance of the Win32 and other session-specific drivers among all the processes in the same terminal-server (TS) session. Different TS sessions share different instances of these drivers, yet they are mapped at the same virtual addresses. The lower, user-mode region of virtual address space is specific to each process and accessible by both user- and kernel-mode threads.

The Windows VM manager uses a two-step process to allocate virtual memory. The first step ***reserves*** one or more pages of virtual addresses in 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 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 space 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.

Windows implements shared memory by defining a **section object**. After getting a handle to a section object, a process maps the memory of the section to a range of addresses, called a **view**. A process can establish a view of the entire section or only the portion it needs. Windows allows sections to be mapped not just into the current process but into any process for which the caller has a handle.

Sections can be used in many ways. A 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 that it appears at the same virtual address for all processes attempting to access it. Sections can also represent physical memory, allowing a 32-bit process to access more physical memory than can fit in its virtual address space. 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.

Let's look more closely at the last two of these protection settings:

- A ***no-access page*** raises an exception if accessed. The exception can be used, for example, to check whether a faulty program iterates beyond the end of an array or simply to detect that the program attempted to access virtual addresses that are not committed to memory. User- and kernel-mode stacks use no-access pages as **guard pages** to detect stack overflows. Another use is to look for heap buffer overruns. 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 to detect programming errors that access beyond the end of an allocation.
- The ***copy-on-write mechanism*** enables the VM manager to use physical memory more efficiently. When two processes want independent copies of data from the same section 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 uses a multilevel page table. For IA-32 and AMD64 processors, each process has a **page directory** that contains 512 **page-directory entries** (**PDEs**) 8 bytes in size. Each PDE points to a **PTE table** that contains 512 **page-table entries** (**PTEs**) 8 bytes in size. Each PTE points to a 4-KB **page frame** in physical memory. For a variety of reasons, the hardware requires that the page directories or PTE tables at each level of a multilevel page table occupy a single page. Thus, the number of PDEs or PTEs that fit in a page determine how many virtual addresses are translated by that page. See Figure B.3 for a diagram of this structure.

The structure described so far can be used to represent only 1 GB of virtual address translation. For IA-32, a second page-directory level is needed, con-

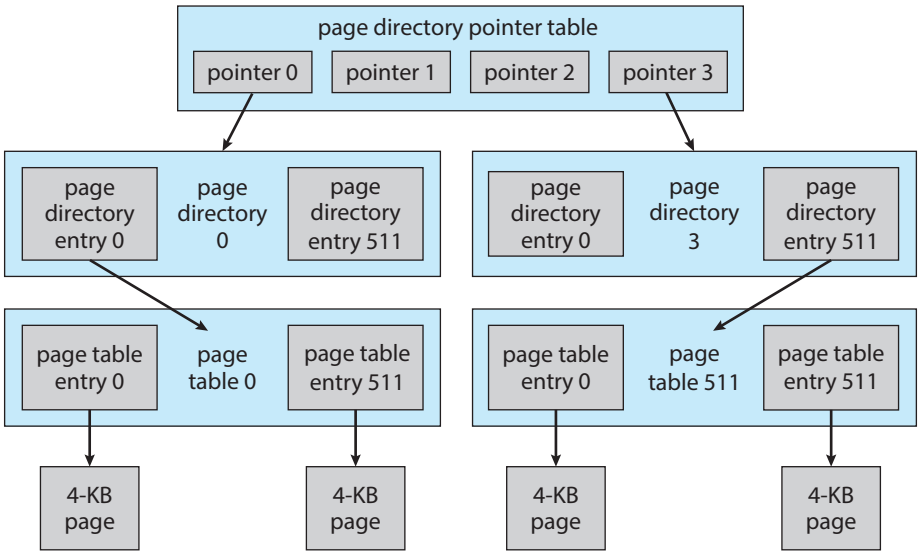


Figure B.3 Page-table layout.

taining only four entries, as shown in the diagram. On 64-bit processors, more levels are needed. For AMD64, Windows uses a total of four full levels. The total size of all page-table pages needed to fully represent even a 32-bit virtual address space for a process is 8 MB. The VM manager allocates pages of PDEs and PTEs as needed and moves page-table pages to disk when not in use. The page-table pages are faulted back into memory when referenced.

We next consider how virtual addresses are translated into physical addresses on IA-32-compatible processors. A 2-bit value can represent the values 0, 1, 2, 3. A 9-bit value can represent values from 0 to 511; a 12-bit value, values from 0 to 4,095. Thus, a 12-bit value can select any byte within a 4-KB page of memory. A 9-bit value can represent any of the 512 PDEs or PTEs in a page directory or PTE-table page. As shown in Figure B.4, translating a virtual address pointer to a byte address in physical memory involves breaking the 32-bit pointer into four values, starting from the most significant bits:

- Two bits are used to index into the four PDEs at the top level of the page table. The selected PDE will contain the physical page number for each of the four page-directory pages that map 1 GB of the address space.

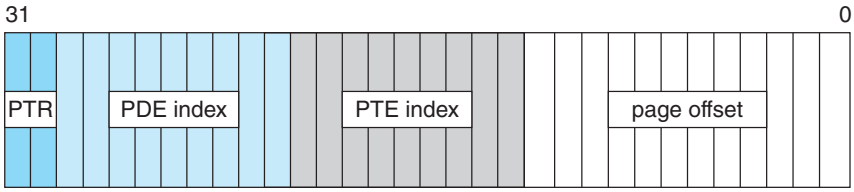


Figure B.4 Virtual-to-physical address translation on IA-32.

- Nine bits are used to select another PDE, this time from a second-level page directory. This PDE will contain the physical page numbers of up to 512 PTE-table pages.
- Nine bits are used to select one of 512 PTEs from the selected PTE-table page. The selected PTE will contain the physical page number for the byte we are accessing.
- Twelve bits are used as the byte offset into the page. The physical address of the byte we are accessing is constructed by appending the lowest 12 bits of the virtual address to the end of the physical page number we found in the selected PTE.

The number of bits in a physical address may be different from the number of bits in a virtual address. In the original IA-32 architecture, the PTE and PDE were 32-bit structures that had room for only 20 bits of physical page number, so the physical address size and the virtual address size were the same. Such systems could address only 4 GB of physical memory. Later, the IA-32 was extended to the larger 64-bit PTE size used today, and the hardware supported 24-bit physical addresses. These systems could support 64 GB and were used on server systems. Today, all Windows servers are based on either the AMD64 or the IA64 and support very, very large physical addresses—more than we can possibly use. (Of course, once upon a time 4 GB seemed optimistically large for physical memory.)

To improve performance, the VM manager maps the page-directory and PTE-table pages into the same contiguous region of virtual addresses in every process. This self-map allows the VM manager to use the same pointer to access the current PDE or PTE corresponding to a particular virtual address no matter what process is running. The self-map for the IA-32 takes a contiguous 8-MB region of kernel virtual address space; the AMD64 self-map occupies 512 GB. Although the self-map occupies significant address space, it does not require any additional virtual memory pages. It also allows the page table's pages to be automatically paged in and out of physical memory.

In the creation of a self-map, one of the PDEs in the top-level page directory refers to the page-directory page itself, forming a "loop" in the page-table translations. The virtual pages are accessed if the loop is not taken, the PTE-table pages are accessed if the loop is taken once, the lowest-level page-directory pages are accessed if the loop is taken twice, and so forth.

The additional levels of page directories used for 64-bit virtual memory are translated in the same way except that the virtual address pointer is broken up into even more values. For the AMD64, Windows uses four full levels, each of which maps 512 pages, or 9+9+9+9+12 = 48 bits of virtual address.

To avoid the overhead of translating every virtual address by looking up the PDE and PTE, processors use **translation look-aside buffer** (**TLB**) hardware, which contains an associative memory cache for mapping virtual pages to PTEs. The TLB is part of the **memory-management unit** (**MMU**) within each processor. The MMU needs to "walk" (navigate the data structures of) the page table in memory only when a needed translation is missing from the TLB.

The PDEs and PTEs contain more than just physical page numbers. They also have bits reserved for operating-system use and bits that control how the hardware uses memory, such as whether hardware caching should be used for

each page. In addition, the entries specify what kinds of access are allowed for both user and kernel modes.

A PDE can also be marked to say that it should function as a PTE rather than a PDE. On a IA-32, the first 11 bits of the virtual address pointer select a PDE in the first two levels of translation. If the selected PDE is marked to act as a PTE, then the remaining 21 bits of the pointer are used as the offset of the byte. This results in a 2-MB size for the page. Mixing and matching 4-KB and 2-MB page sizes within the page table is easy for the operating system and can significantly improve the performance of some programs by reducing how often the MMU needs to reload entries in the TLB, since one PDE mapping 2 MB replaces 512 PTEs each mapping 4 KB.

Managing physical memory so that 2-MB pages are available when needed is difficult, however, as they may continually be broken up into 4-KB pages, causing external fragmentation of memory. Also, the large pages can result in very significant internal fragmentation. Because of these problems, it is typically only Windows itself, along with large server applications, that use large pages to improve the performance of the TLB. They are better suited to do so because operating-system and server applications start running when the system boots, before memory has become fragmented.

Windows manages physical memory by associating each physical page with one of seven states: free, zeroed, modified, standby, bad, transition, or valid.

- A ***free*** page is a page that has no particular content.
- 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 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 may be pages that were not modified, modified pages that have already been written to the disk, or pages that were prefetched because they are expected to be used soon.
- A ***bad*** page is unusable because a hardware error has been detected.
- A ***transition*** page is on its way in from disk to a page frame allocated in physical memory.
- A ***valid*** page is part of the working set of one or more processes and is contained within these processes' page tables.

While valid pages are contained in processes' page tables, pages in other states are kept in separate lists according to state type. The lists are constructed by linking the corresponding entries in the **page frame number** (**PFN**) database, which includes an entry for each physical memory page. The PFN entries also include information such as reference counts, locks, and NUMA information. Note that the PFN database represents pages of physical memory, whereas the PTEs represent pages of virtual memory.

When the valid bit in a PTE is zero, hardware ignores all the other bits, and the VM manager can define them for its own use. Invalid pages can have a number of states represented by bits in the PTE. Page-file pages that have never

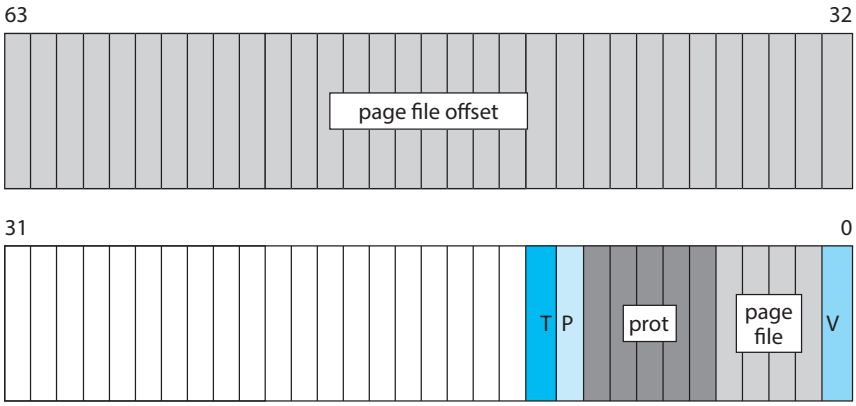


Figure B.5 Page-file page-table entry. The valid bit is zero.

been faulted in are marked zero-on-demand. Pages mapped through section objects encode a pointer to the appropriate section object. PTEs for pages that have been written to the page file contain enough information to locate the page on disk, and so forth. The structure of the page-file PTE is shown in Figure B.5. The T, P, and V bits are all zero for this type of PTE. The PTE includes 5 bits for page protection, 32 bits for page-file offset, and 4 bits to select the paging file. There are also 20 bits reserved for additional bookkeeping.

Windows uses a per-working-set, least-recently-used (LRU) replacement policy to take pages from processes as appropriate. When a process is started, it is assigned a default minimum working-set size. The working set of each process is allowed to grow until the amount of remaining physical memory starts to run low, at which point the VM manager starts to track the age of the pages in each working set. Eventually, when the available memory runs critically low, the VM manager trims the working set to remove older pages.

How old a page is depends not on how long it has been in memory but on when it was last referenced. This is determined by periodically making a pass through the working set of each process and incrementing the age for pages that have not been marked in the PTE as referenced since the last pass. When it becomes necessary to trim the working sets, the VM manager uses heuristics to decide how much to trim from each process and then removes the oldest pages first.

A process can have its working set trimmed even when plenty of memory is available, if it was given a ***hard limit*** on how much physical memory it could use. In Windows 7, the VM manager will also trim processes that are growing rapidly, even if memory is plentiful. This policy change significantly improves the responsiveness of the system for other processes.

Windows tracks working sets not only for user-mode processes but also for the system process, which includes all the pageable data structures and code that run in kernel mode. Windows 7 created additional working sets for the system process and associated them with particular categories of kernel memory; the file cache, kernel heap, and kernel code now have their own working sets. The distinct working sets allow the VM manager to use different policies to trim the different categories of kernel memory.

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. That is, 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 and allows reads to be clustered to improve I/O performance.

In addition to managing committed memory, the VM manager manages each process's reserved memory, or virtual address space. Each process has an associated tree that describes the ranges of virtual addresses in use and what the uses are. This allows the VM manager to fault in page-table pages as needed. If the PTE for a faulting address is uninitialized, 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 PTE and retrieve the page. In some cases, a PTE-table page itself may not exist; such a page must be transparently allocated and initialized by the VM manager. In other cases, the page may be shared as part of a section object, and the VAD will contain a pointer to that section object. The section object contains information on how to find the shared virtual page so that the PTE can be initialized to point at it directly.

B.3.3.3 Process Manager

The Windows 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 into larger units called **job objects**. The use of job objects allows limits to be placed on CPU usage, working-set size, and processor affinities that control multiple processes at once. Job objects are used to manage large data-center machines.

An example of process creation in the Win32 environment is as follows:

1. A Win32 application calls `CreateProcess()`.
2. A message is sent to the Win32 subsystem to notify it that the process is being created.
3. `CreateProcess()` in the original process then calls an API in the process manager of the NT executive to actually create the process.
4. The process manager calls the object manager to create a process object and returns the object handle to Win32.
5. Win32 calls the process manager again to create a thread for the process and returns handles to the new process and thread.

The Windows 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 in `ntdll.dll`, which is a link library automatically mapped into every newly created process. Windows also supports a UNIX `fork()` style of process creation in order to support the POSIX environmental subsystem. Although the Win32 environment calls the process manager directly from the client process, POSIX uses the cross-process nature of the Windows APIs to create the new process from within the subsystem process.

The process manager relies on the asynchronous procedure calls (APCs) implemented by the kernel layer. APCs are used to initiate thread execution, suspend and resume threads, access thread registers, terminate threads and processes, and support debuggers.

The debugger support in the process manager includes the APIs to suspend and resume threads and to create threads that begin in 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. The debugger makes use of thread injection to execute code within a process being debugged.

While running in the executive, a thread can temporarily attach to a different process. **Thread attach** is used by kernel worker threads that need to execute in the context of the process originating a work request. For example, the VM manager might use thread attach when it needs access to a process's working set or page tables, and the I/O manager might use it in updating the status variable in a process for asynchronous I/O operations.

The process manager also supports **impersonation**. Each thread has an associated **security token**. When the login process authenticates a user, the security token is attached to the user's process and inherited by its child processes. The token contains the **security identity** (**SID**) of the user, the SIDs of the groups the user belongs to, the privileges the user has, and the integrity level of the process. By default, all threads within a process share a common token, representing the user and the application that started the process. However, a thread running in a process with a security token belonging to one user can set a thread-specific token belonging to another user to impersonate that user.

The impersonation facility is fundamental to the client–server RPC model, where services must act on behalf of a variety of clients with different security IDs. The right to impersonate a user is most often delivered as part of an RPC connection from a client process to a server process. Impersonation allows the server to access system services as if it were the client in order to access or create objects and files on behalf of the client. The server process must be trustworthy and must be carefully written to be robust against attacks. Otherwise, one client could take over a server process and then impersonate any user who made a subsequent client request.

B.3.3.4 Facilities for Client–Server Computing

The implementation of Windows uses a client–server model throughout. The environmental subsystems are servers that implement particular operating-system personalities. Many other services, such as user authentication, net-

work facilities, printer spooling, web services, network file systems, and plug-and-play, are also implemented using this model. To reduce the memory footprint, multiple services are often collected into a few processes running the `svchost.exe` program. Each service is loaded as a dynamic-link library (DLL), which implements the service by relying on the user-mode thread-pool facilities to share threads and wait for messages (see Section B.3.3.3).

The normal implementation paradigm for client–server computing is to use RPCs to communicate requests. The Win32 API supports a standard RPC protocol, as described in Section B.6.2.7. RPC uses multiple transports (for example, named pipes and TCP/IP) and can be used to implement RPCs between systems. When an RPC always occurs between a client and server on the local system, the advanced local procedure call facility (ALPC) can be used as the transport. At the lowest level of the system, in the implementation of the environmental systems, and for services that must be available in the early stages of booting, RPC is not available. Instead, native Windows services use ALPC directly.

ALPC is a message-passing mechanism. The server process publishes a globally visible connection-port object. When a client wants services from a subsystem or service, it opens a handle to the server'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 ALPC channel is created, one of three message-passing techniques is chosen.

1. The first technique is suitable for small to medium messages (up to 63 KB). 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 APIs that read and write directly into a process's address space. ALPC provides functions and synchronization so that a server can access the data in a client. This technique is normally used by RPC to achieve higher performance for specific scenarios.

The Win32 window manager uses its own form of message passing, which is independent of the executive ALPC 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 shared section object, and (3) an event-pair object. An ***event-pair object*** is a synchronization object used by the Win32 subsystem to provide notification when the client thread has copied a message to the Win32 server, or vice versa. The section object is used to pass the messages, and the event-pair object provides 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.

B.3.3.5 I/O Manager

The **I/O manager** is responsible for managing 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 cache manager, which handles caching for the entire I/O system. The I/O manager is fundamentally asynchronous, providing synchronous I/O 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, updating of a status variable in the calling process, delivery of APCs to initiating threads, 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 in a list for each device (called a driver or I/O stack). A driver is represented in the system as a **driver object**. Because a single driver can operate on multiple devices, the drivers are represented in the I/O stack by a **device object**, which contains a link to the driver object. 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 targeted I/O stack for processing. After a driver processes the IRP, it calls the I/O manager either to forward the IRP to the next driver in the stack or, if all processing is finished, to complete the operation represented by the IRP.

The I/O request may be completed in a context different from the one in which it was made. 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. An IRP may also be processed in interrupt-service routines and completed in an arbitrary process context. Because some final processing may need to take place in the context that initiated the I/O, the I/O manager uses an APC to do final I/O-completion processing in the process context of the originating thread.

The I/O 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 functionalities 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 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 a great deal of work. Fortunately, the port/miniport model makes it unnecessary to do this. Within a class of similar devices, such as audio drivers, SATA 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. The TCP/IP network stack is implemented in this way, with the `ndis.sys` class driver implementing much of the network driver functionality and calling out to the network miniport drivers for specific hardware.

Recent versions of Windows, including Windows 7, provide additional simplifications for writing device drivers for hardware devices. Kernel-mode drivers can now be written using the **Kernel-Mode Driver Framework** (**KMDF**), which provides a simplified programming model for drivers on top of WDM. Another option is the **User-Mode Driver Framework** (**UMDF**). Many drivers do not need to operate in kernel mode, and it is easier to develop and deploy drivers in user mode. It also makes the system more reliable, because a failure in a user-mode driver does not cause a kernel-mode crash.

B.3.3.6 Cache Manager

In many operating systems, caching is done by the file system. Instead, Windows 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 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. The cache manager maintains a private working set rather than sharing the system process's working set. The cache manager memory-maps files into kernel memory and then uses special interfaces to the VM manager to fault pages into or trim them from this private working set.

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.

When the I/O manager receives a file's user-level read request, the I/O manager sends an IRP to the I/O stack for the volume on which the file resides. For files that are marked as cacheable, the file system calls the cache manager

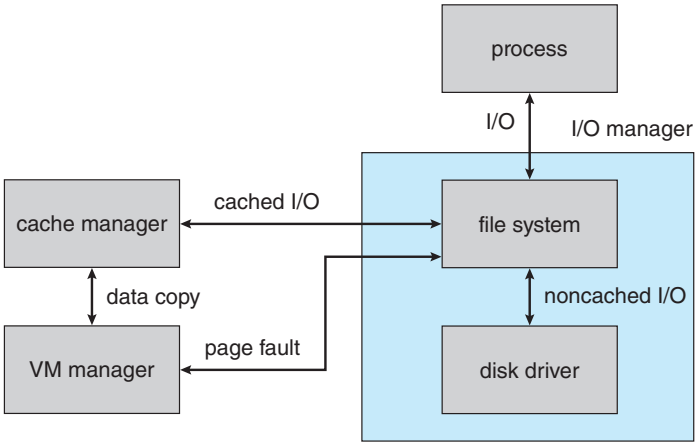


Figure B.6 File I/O.

to look up the requested data in its cached file views. 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 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 B.6 shows an overview of these operations.

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 the page or page it out. 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.

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 may find its data already cached and not need to wait for disk I/O.

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.

B.3.3.7 Security Reference Monitor

Centralizing management of system entities in the object manager enables Windows 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 access 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, 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 used primarily to limit the damage that can be done by execution of untrusted code.

The integrity level of the code executing in a process is also represented by a token. Integrity levels are a type of capability mechanism, as mentioned earlier. A process cannot modify an object with an integrity level higher than that of the code executing in the process, whatever other permissions have been granted. Integrity levels were introduced to make it harder for code that successfully attacks outward-facing software, like Internet Explorer, to take over a system.

Another responsibility of the SRM is logging security audit events. The Department of Defense's **Common Criteria** (the 2005 successor to the Orange Book) requires that a secure system have the ability to detect and log all attempts to access system resources so that it can more easily 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.

B.3.3.8 Plug-and-Play Manager

The operating system uses the **plug-and-play** (**PnP**) manager to recognize and adapt to changes in the hardware configuration. PnP devices use standard protocols to identify themselves to the system. The PnP manager automatically recognizes installed devices and detects changes in devices as the system

operates. The manager also keeps track of hardware 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 successfully.

The PnP manager handles dynamic reconfiguration as follows. First, it gets a list of devices from each bus driver (for example, PCI or USB). It loads the installed driver (after finding one, if necessary) and sends an `add-device` request to the appropriate driver for each device. The PnP manager then figures out the optimal resource assignments and sends a `start-device` request to each driver specifying the resource assignments 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. Finally, the PnP manager sends a `stop` request and can then reconfigure the device with a new `start-device` request.

The PnP manager also supports other requests. For example, `query-remove`, which operates similarly to `query-stop`, is employed when a user is getting ready to eject a removable device, such as a USB storage device. The `surprise-remove` request is used when a device fails or, more likely, when a user removes a device without telling the system to stop it first. Finally, the `remove` request tells the driver to stop using a device permanently.

Many programs in the system are interested in the addition or removal of devices, so the PnP manager supports notifications. Such a notification, for example, gives GUI file menus the information they need to update their list of disk volumes when a new storage device is attached or removed. Installing devices often results in adding new services to the `svchost.exe` processes in the system. These services frequently set themselves up to run whenever the system boots and continue to run even if the original device is never plugged into the system. Windows 7 introduced a **service-trigger** mechanism in the **service control manager** (**SCM**), which manages the system services. With this mechanism, services can register themselves to start only when SCM receives a notification from the PnP manager that the device of interest has been added to the system.

B.3.3.9 Power Manager

Windows works with the hardware to implement sophisticated strategies for energy efficiency, as described in Section B.2.8. The policies that drive these strategies are implemented by the **power manager**. The power manager detects current system conditions, such as the load on CPUs or I/O devices, and improves energy efficiency by reducing the performance and responsiveness of the system when need is low. The power manager can also put the entire system into a very efficient ***sleep*** mode and can even write all the contents of memory to disk and turn off the power to allow the system to go into ***hibernation***.

The primary advantage of sleep is that the system can enter fairly quickly, perhaps just a few seconds after the lid closes on a laptop. The return from sleep is also fairly quick. The power is turned down low on the CPUs and I/O devices, but the memory continues to be powered enough that its contents are not lost.

Hibernation takes considerably longer because the entire contents of memory must be transferred to disk before the system is turned off. However, the fact that the system is, in fact, turned off is a significant advantage. If there is a loss of power to the system, as when the battery is swapped on a laptop or a desktop system is unplugged, the saved system data will not be lost. Unlike shutdown, hibernation saves the currently running system so a user can resume where she left off, and because hibernation does not require power, a system can remain in hibernation indefinitely.

Like the PnP manager, the power manager provides notifications to the rest of the system about changes in the power state. Some applications want to know when the system is about to be shut down so they can start saving their states to disk.

B.3.3.10 Registry

Windows keeps much of its configuration information in internal databases, called **hives**, that are managed by the Windows configuration manager, which is commonly known as the **registry**. There are separate hives for system information, default user preferences, software installation, security, and boot options. Because the information in the **system hive** is required to boot the system, the registry manager is implemented as a component of the executive.

The registry represents the configuration state in each hive as a hierarchical namespace of keys (directories), each of which can contain a set of typed values, such as UNICODE string, ANSI string, integer, or untyped binary data. In theory, new keys and values are created and initialized as new software is installed; then they are modified to reflect changes in the configuration of that software. In practice, the registry is often used as a general-purpose database, as an interprocess-communication mechanism, and for many other such inventive purposes.

Restarting applications, or even the system, every time a configuration change was made would be a nuisance. Instead, programs rely on various kinds of notifications, such as those provided by the PnP and power managers, to learn about changes in the system configuration. The registry also supplies notifications; it allows threads to register to be notified when changes are made to some part of the registry. The threads can thus detect and adapt to configuration changes recorded in the registry itself.

Whenever significant changes are made to the system, such as when updates to the operating system or drivers are installed, there is a danger that the configuration data may be corrupted (for example, if a working driver is replaced by a nonworking driver or an application fails to install correctly and leaves partial information in the registry). Windows creates a **system restore point** before making such changes. The restore point contains a copy of the hives before the change and can be used to return to this version of the hives and thereby get a corrupted system working again.

To improve the stability of the registry configuration, Windows added a transaction mechanism beginning with Windows Vista that can be used to prevent the registry from being partially updated with a collection of related configuration changes. Registry transactions can be part of more general transactions administered by the **kernel transaction manager** (**KTM**), which can also

include file-system transactions. KTM transactions do not have the full semantics found in normal database transactions, and they have not supplanted the system restore facility for recovering from damage to the registry configuration caused by software installation.

B.3.3.11 Booting

The booting of a Windows PC begins when the hardware powers on and firmware begins executing from ROM. In older machines, this firmware was known as the BIOS, but more modern systems use UEFI (the Unified Extensible Firmware Interface), which is faster and more general and makes better use of the facilities in contemporary processors. The firmware runs **power-on self-test** (**POST**) diagnostics; identifies many of the devices attached to the system and initializes them to a clean, power-up state; and then builds the description used by the **advanced configuration and power interface** (**ACPI**). Next, the firmware finds the system disk, loads the Windows `bootmgr` program, and begins executing it.

In a machine that has been hibernating, the `winresume` program is loaded next. It restores the running system from disk, and the system continues execution at the point it had reached right before hibernating. In a machine that has been shut down, the `bootmgr` performs further initialization of the system and then loads `winload`. This program loads `hal.dll`, the kernel (`ntoskrnl.exe`), any drivers needed in booting, and the system hive. `winload` then transfers execution to the kernel.

The kernel initializes itself and creates two processes. The **system process** contains all the internal kernel worker threads and never executes in user mode. The first user-mode process created is SMSS, for ***session manager subsystem***, which is similar to the INIT (initialization) process in UNIX. SMSS performs further initialization of the system, including establishing the paging files, loading more device drivers, and managing the Windows sessions. Each session is used to represent a logged-on user, except for ***session 0***, which is used to run system-wide background services, such as LSASS and SERVICES. A session is anchored by an instance of the CSRSS process. Each session other than 0 initially runs the WINLOGON process. This process logs on a user and then launches the EXPLORER process, which implements the Windows GUI experience. The following list itemizes some of these aspects of booting:

- SMSS completes system initialization and then starts up session 0 and the first login session.
- WININIT runs in session 0 to initialize user mode and start LSASS, SERVICES, and the local session manager, LSM.
- LSASS, the security subsystem, implements facilities such as authentication of users.
- SERVICES contains the service control manager, or SCM, which supervises all background activities in the system, including user-mode services. A number of services will have registered to start when the system boots. Others will be started only on demand or when triggered by an event such as the arrival of a device.

- CSRSS is the Win32 environmental subsystem process. It is started in every session—unlike the POSIX subsystem, which is started only on demand when a POSIX process is created.
- WINLOGON is run in each Windows session other than session 0 to log on a user.

The system optimizes the boot process by prepaging from files on 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 necessary to start the system are reduced by grouping services into fewer processes. 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.

B.4 Terminal Services and Fast User Switching

Windows supports a GUI-based console that interfaces with the user via keyboard, mouse, and display. Most systems also support audio and video. Audio input is used by Windows voice-recognition software; voice recognition makes the system more convenient and increases its accessibility for users with disabilities. Windows 7 added support for **multi-touch hardware**, allowing users to input data by touching the screen and making gestures with one or more fingers. Eventually, the video-input capability, which is currently used for communication applications, is likely to be used for visually interpreting gestures, as Microsoft has demonstrated for its Xbox 360 Kinect product. Other future input experiences may evolve from Microsoft's **surface computer**. Most often installed at public venues, such as hotels and conference centers, the surface computer is a table surface with special cameras underneath. It can track the actions of multiple users at once and recognize objects that are placed on top.

The PC was, of course, envisioned as a ***personal computer***—an inherently single-user machine. Modern Windows, however, supports the sharing of a PC among multiple users. Each user that is logged on using the GUI has a **session** created to represent the GUI environment he will be using and to contain all the processes created to run his applications. Windows allows multiple sessions to exist at the same time on a single machine. However, Windows only supports a single console, consisting of all the monitors, keyboards, and mice connected to the PC. Only one session can be connected to the console at a time. From the logon screen displayed on the console, users can create new sessions or attach to an existing session that was previously created. This allows multiple users to share a single PC without having to log off and on between users. Microsoft calls this use of sessions ***fast user switching***.

Users can also create new sessions, or connect to existing sessions, on one PC from a session running on another Windows PC. The terminal server (TS) connects one of the GUI windows in a user's local session to the new or existing session, called a **remote desktop**, on the remote computer. The most common use of remote desktops is for users to connect to a session on their work PC from their home PC.

Many corporations use corporate terminal-server systems maintained in data centers to run all user sessions that access corporate resources, rather than

allowing users to access those resources from the PCs in each user's office. Each server computer may handle many dozens of remote-desktop sessions. This is a form of **thin-client** computing, in which individual computers rely on a server for many functions. Relying on data-center terminal servers improves reliability, manageability, and security of the corporate computing resources.

The TS is also used by Windows to implement **remote assistance**. A remote user can be invited to share a session with the user logged on to the session on the console. The remote user can watch the user's actions and even be given control of the desktop to help resolve computing problems.

B.5 File System

The native file system in Windows is NTFS. It is used for all local volumes. However, associated USB thumb drives, flash memory on cameras, and external disks may be formatted with the 32-bit FAT file system for portability. FAT is a much older file-system format that is understood by many systems besides Windows, such as the software running on cameras. A disadvantage is that the FAT file system does not restrict file access to authorized users. The only solution for securing data with FAT is to run an application to encrypt the data before storing it on the file system.

In contrast, NTFS uses ACLs to control access to individual files and supports implicit encryption of individual files or entire volumes (using Windows BitLocker feature). NTFS implements many other features as well, including data recovery, fault tolerance, very large files and file systems, multiple data streams, UNICODE names, sparse files, journaling, volume shadow copies, and file compression.

B.5.1 NTFS Internal Layout

The fundamental entity in NTFS is a volume. A volume is created by the Windows logical disk management utility and is based on a logical disk partition. A volume may occupy a portion of a disk or 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 based on the volume size—4 KB for volumes larger than 2 GB. Given the size of today's disks, it may make sense to use cluster sizes larger than the Windows defaults to achieve better performance, although these performance gains will come at the expense of more internal fragmentation.

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 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 short name), the creation time, and the

security descriptor that specifies the access control list. User data are 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 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 only the size of the unnamed attribute 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.

B.5.1.1 NTFS B+ Tree

As in 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. In a B+ tree, the length of every path from the root of the tree to a leaf is the same, and the cost of reorganizing the tree is eliminated. 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 that 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.

B.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 files described 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 using the `chkdsk` program.
- 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 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.

Keeping all the NTFS metadata in actual files has a useful property. As discussed in Section B.3.3.6, the cache manager caches file data. Since all the NTFS metadata reside in files, these data can be cached using the same mechanisms used for ordinary data.

B.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 UNIX file systems, including UFS but not ZFS, store redundant metadata on the disk, and they recover from crashes by 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 took some steps to do this in Windows Vista.

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 ***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 and then resets the log file and performs the queued transactions.

B.5.3 Security

The security of an NTFS volume is derived from the Windows object model. Each NTFS file references a security descriptor, which specifies the owner of the file, and an access-control list, which contains the access permissions granted or denied to each user or group listed. Early versions of NTFS used a separate security descriptor as an attribute of each file. Beginning with Windows 2000, the security-descriptors attribute points to a shared copy, with a significant savings in disk and caching space; many, many files have identical security descriptors.

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 directory-by-directory parsing of path names. Prefix matching is an algorithm that looks up strings in a cache and finds the entry with the longest match—for example, an entry for `\foo\bar\dir` would be a match for `\foo\bar\dir2\dir3\myfile`. The prefix-matching cache allows path-name traversal to begin much deeper in the tree, saving many steps. Enforcing traversal checks means that the user's access must be checked at each directory level. For instance, a user might lack permission to traverse `\foo\bar`, so starting at the access for `\foo\bar\dir` would be an error.

B.5.4 Volume Management and Fault Tolerance

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

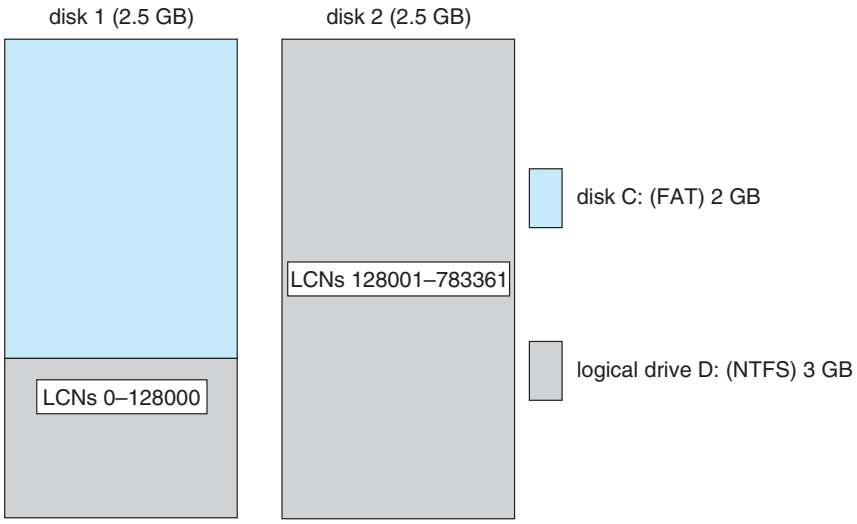


Figure B.7 Volume set on two drives.

B.5.4.1 Volume Sets and RAID Sets

One way to combine multiple disks is to concatenate them logically to form a large logical volume, as shown in Figure B.7. In Windows, 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.

Another way to combine multiple physical partitions is to interleave their blocks in round-robin fashion to form a **stripe set**. This scheme is also called RAID level 0, or **disk striping**. (For more on RAID (redundant arrays of inexpensive disks), see Section 11.8.) `FtDisk` 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. Windows also supports RAID level 5, stripe set with parity, and RAID level 1, mirroring.

B.5.4.2 Sector Sparing and Cluster Remapping

To deal with disk sectors that go bad, `FtDisk` uses a hardware technique called sector sparing, and NTFS uses a software technique called cluster remapping. **Sector sparing** 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 sparing 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 in a new location that is obtained by sector sparing or cluster remapping.

B.5.5 Compression

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 a 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 NTFS finds a gap in the virtual-cluster numbers, it just zero-fills that portion of the caller's buffer. This technique is also used by UNIX.

B.5.6 Mount Points, Symbolic Links, and Hard Links

Mount points are a form of symbolic link specific to directories on NTFS that were introduced in Windows 2000. They provide a mechanism for organizing disk volumes that is more flexible than the use of global names (like drive letters). A mount point is implemented as a symbolic link with associated data that contains 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.

Windows Vista introduced support for a more general form of symbolic links, similar to those found in UNIX. The links can be absolute or relative, can point to objects that do not exist, and can point to both files and directories even across volumes. NTFS also supports **hard links**, where a single file has an entry in more than one directory of the same volume.

B.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 by reading from the journal. The search indexer 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.

B.5.8 Volume Shadow Copies

Windows 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. This technique is known as ***snapshots*** in some other file systems. Making a shadow copy of a volume is a form of copy-on-write, where blocks modified after the shadow copy is created are stored in their original form 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 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 backup media.

B.6 Networking

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

B.6.1 Network Interfaces

To describe networking in Windows, 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 and network layers in the ISO model and enables many protocols to operate over many different network adapters. In terms of the ISO 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.

B.6.2 Protocols

Windows 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 comes with several networking protocols. Next, we discuss a number of these protocols.

B.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. `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. `Printer` messages are used to send data to a remote print queue and to receive status information from the queue, and `Message` messages are used to communicate with another workstation. A version of the SMB protocol was published as the **common Internet file system** (**CIFS**) and is supported on a number of operating systems.

B.6.2.2 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 uses TCP/IP to connect to a wide variety of operating systems and hardware platforms. The Windows TCP/IP package includes the simple network-management protocol (SNM), the dynamic host-configuration protocol (DHCP), and the older Windows Internet name service (WINS). Windows Vista introduced a new implementation of TCP/IP that supports both `IPv4` and `IPv6` in the same network stack. This new implementation also supports offloading of the network stack onto advanced hardware, to achieve very high performance for servers.

Windows provides a software firewall that limits the TCP ports that can be used by programs for network communication. Network firewalls are commonly implemented in routers and are a very important security measure. Having a firewall built into the operating system makes a hardware router unnecessary, and it also provides more integrated management and easier use.

B.6.2.3 Point-to-Point Tunneling Protocol

The **point-to-point tunneling protocol** (**PPTP**) is a protocol provided by Windows to communicate between remote-access server modules running on Windows 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 multiprotocol **virtual private networks** (**VPNs**) over the Internet.

B.6.2.4 HTTP Protocol

The HTTP protocol is used to `get/put` information using the World Wide Web. Windows implements HTTP using a kernel-mode driver, so web servers can operate with a low-overhead connection to the networking stack. HTTP is a fairly general protocol that Windows makes available as a transport option for implementing RPC.

B.6.2.5 Web-Distributed Authoring and Versioning Protocol

Web-distributed authoring and versioning (WebDAV) is an HTTP-based protocol for collaborative authoring across a network. Windows builds a WebDAV

redirector into the file system. Being built directly into the file system enables WebDAV to work with other file-system features, such as encryption. Personal files can then be stored securely in a public place. Because WebDAV uses HTTP, which is a `get/put` protocol, Windows has to cache the files locally so programs can use `read` and `write` operations on parts of the files.

B.6.2.6 Named Pipes

Named pipes are a connection-oriented messaging mechanism. A process can 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 SMB protocol supports named pipes, so they can also be used for communication between processes on different systems.

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

B.6.2.7 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 **marshaling**. The client stub code and the descriptors necessary to pack and unpack the arguments for an RPC are compiled from a specification written in the **Microsoft Interface Definition Language**.

The Windows RPC mechanism follows the widely used distributed-computing-environment standard for RPC messages, so programs written to use Windows 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.

B.6.2.8 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. Many Windows services provide COM interfaces. Windows has a distributed extension called

DCOM that can be used over a network utilizing RPC to provide a transparent method of developing distributed applications.

B.6.3 Redirectors and Servers

In Windows, an application can use the Windows 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 those provided by Windows. A **redirector** is the client-side object that forwards I/O requests to a remote system, 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 B.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.
7. 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 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 described in Section B.6.2. The list of redirectors is maintained in the system hive of the registry.

B.6.3.1 Distributed File System

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 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.

B.6.3.2 Folder Redirection and Client-Side Caching

To improve the PC experience for users who frequently switch among computers, Windows 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, as when a user takes a laptop onto an airplane. To give users off-line access to their redirected files, Windows uses **client-side caching** (CSC). CSC is also used when the computer is on-line 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.

B.6.4 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 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 domain is a group of Windows workstations and servers that share a common security policy and user database. Since Windows uses the Kerberos protocol for trust and authentication, a Windows domain is the same thing as a Kerberos realm. Windows uses a hierarchical approach for establishing trust relationships between related domains. The trust relationships are based on DNS and allow 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 (loaded into the user's access token by LSASS). All users retain the ability to restrict access to their own workstations, however, no matter what any domain controller may say.

B.6.5 Active Directory

Active Directory is the Windows 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 Windows features that need it, such as **Windows group policy**. Administrators use group policies to establish uniform standards for desktop preferences and software. For many

corporate information-technology groups, this uniformity drastically reduces the cost of computing.

B.7 Programmer Interface

The Win32 API is the fundamental interface to the capabilities of Windows. 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.

B.7.1 Access to Kernel Objects

The Windows 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 an instance of `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 handles referencing the object in all processes drops to zero.

B.7.2 Sharing Objects between Processes

Windows 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 `SECURITIES_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 B.8 shows a code sample that creates a semaphore handle inherited by a child process.

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

Figure B.8 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 B.9 Code for sharing an object by name lookup.

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 B.8, 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 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 and share a single object named "foo" when two distinct objects—possibly of different types—were 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 as a parameter. The second process gets a handle to share the object by calling `OpenXXX()` (or `CreateXXX`) with the same name, as shown in the example in Figure B.9.

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 B.10.

B.7.3 Process Management

In Windows, a **process** is a loaded instance of an application and a **thread** is an executable unit of code that can be scheduled by the kernel dispatcher. Thus, a process contains one or more threads. A process is created when a thread in some other process calls the `CreateProcess()` API. This routine loads any dynamic link libraries used by the process and creates an initial thread in the process. Additional threads can be created by the `CreateThread()` function. Each thread is created with its own stack, which defaults to 1 MB unless otherwise specified in an argument to `CreateThread()`.

```
// 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 B.10 Code for sharing an object by passing a handle.

B.7.3.1 Scheduling Rule

Priorities in the Win32 environment are based on the native kernel (NT) scheduling model, but not all priority values may be chosen. The Win32 API uses four priority classes:

1. `IDLE_PRIORITY_CLASS` (NT priority level 4)
2. `NORMAL_PRIORITY_CLASS` (NT priority level 8)
3. `HIGH_PRIORITY_CLASS` (NT priority level 13)
4. `REALTIME_PRIORITY_CLASS` (NT 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 is the default for all threads that execute in the process. It can be changed with the `SetPriorityClass()` function or by passing an argument to the `START` command. 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.

When a user is running an interactive process, the system needs to schedule the process's threads to provide good responsiveness. For this reason, Windows has a special scheduling rule for processes in the `NORMAL_PRIORITY_CLASS`. Windows distinguishes between the process associated with the foreground window on the screen and the other (background) processes. When a process moves into the foreground, Windows increases the scheduling quantum for all its threads by a factor of 3; CPU-bound threads

in the foreground process will run three times longer than similar threads in background processes.

B.7.3.2 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 B.3.2.2 that the kernel has two priority classes: 16–31 for the real-time class and 1–15 for the variable 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 discussed in Section B.3.2.2, the kernel adjusts the priority of a variable class 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.

B.7.3.3 Thread Suspend and Resume

A thread can be created in a ***suspended state*** or can be placed in a suspended state later by use of the `SuspendThread()` function. Before a suspended thread can be scheduled by the kernel dispatcher, it must be moved out of the suspended state by use of the `ResumeThread()` function. Both functions set a counter so that if a thread is suspended twice, it must be resumed twice before it can run.

B.7.3.4 Thread Synchronization

To synchronize concurrent access to shared objects by threads, the kernel provides synchronization objects, such as semaphores and mutexes. These are dispatcher objects, as discussed in Section B.3.2.2. Threads can also synchronize with kernel services operating on kernel objects—such as threads, processes, and files—because these are also dispatcher objects. Synchronization with kernel dispatcher objects can be achieved by use of the `WaitForSingleObject()` and `WaitForMultipleObjects()` functions; these functions wait for one or more dispatcher objects to be signaled.

Another method of synchronization is available to threads within the same process that want to execute code exclusively. The Win32 **critical section object** is a user-mode mutex object that can often be acquired and released without entering the kernel. On a multiprocessor, a Win32 critical section will attempt to spin while waiting for a critical section held by another thread to be released.

If the spinning takes too long, the acquiring thread will allocate a kernel mutex and yield its CPU. Critical sections are particularly efficient because the kernel mutex is allocated only when there is contention and then used only after attempting to spin. Most mutexes in programs are never actually contended, so the savings are significant.

Before using a critical section, some thread in the process must call `InitializeCriticalSection()`. Each thread that wants to acquire the mutex calls `EnterCriticalSection()` and then later calls `LeaveCriticalSection()` to release the mutex. There is also a `TryEnterCriticalSection()` function, which attempts to acquire the mutex without blocking.

For programs that want user-mode reader–writer locks rather than a mutex, Win32 supports **slim reader–writer** (**SRW**) **locks**. SRW locks have APIs similar to those for critical sections, such as `InitializeSRWLock`, `AcquireSRWLockXXX`, and `ReleaseSRWLockXXX`, where `XXX` is either `Exclusive` or `Shared`, depending on whether the thread wants write access or just read access to the object protected by the lock. The Win32 API also supports **condition variables**, which can be used with either critical sections or SRW locks.

B.7.3.5 Thread Pool

Repeatedly creating and deleting threads can be expensive for applications and services that perform small amounts of work in each instantiation. The Win32 thread pool provides user-mode programs with three services: a queue to which work requests may be submitted (via the `SubmitThreadpoolWork()` function), an API that can be used to bind callbacks to waitable handles (`RegisterWaitForSingleObject()`), and APIs to work with timers (`CreateThreadpoolTimer()` and `WaitForThreadpoolTimerCallbacks()`) and to bind callbacks to I/O completion queues (`BindIoCompletionCallback()`).

The goal of using a thread pool is to increase performance and reduce memory footprint. Threads are relatively expensive, and each processor can only be executing one thread at a time no matter how many threads are available. The thread pool attempts to reduce the number of runnable 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 I/O- 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 separate threads to servicing each waitable handle, timer, or completion port.

B.7.3.6 Fibers

A **fiber** is user-mode code that is scheduled according to a user-defined scheduling algorithm. Fibers are completely a user-mode facility; the kernel is not aware that they exist. The fiber mechanism uses Windows threads as if they were CPUs to execute the fibers. Fibers are cooperatively scheduled, meaning that they are never preempted but must explicitly yield the thread on which they are running. When a fiber yields a thread, another fiber can be scheduled on it by the run-time system (the programming language run-time code).

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()`.

Fibers are not recommended for threads that use Win32 APIs rather than standard C-library functions because of potential incompatibilities. Win32 user-mode threads have a **thread-environment block** (**TEB**) that contains numerous per-thread fields used by the Win32 APIs. Fibers must share the TEB of the thread on which they are running. This can lead to problems when a Win32 interface puts state information into the TEB for one fiber and then the information is overwritten by a different fiber. Fibers are included in the Win32 API to facilitate the porting of legacy UNIX applications that were written for a user-mode thread model such as Pthreads.

B.7.3.7 User-Mode Scheduling (UMS) and ConcRT

A new mechanism in Windows 7, user-mode scheduling (UMS), addresses several limitations of fibers. First, recall that fibers are unreliable for executing Win32 APIs because they do not have their own TEBs. When a thread running a fiber blocks in the kernel, the user scheduler loses control of the CPU for a time as the kernel dispatcher takes over scheduling. Problems may result when fibers change the kernel state of a thread, such as the priority or impersonation token, or when they start asynchronous I/O.

UMS provides an alternative model by recognizing that each Windows thread is actually two threads: a kernel thread (KT) and a user thread (UT). Each type of thread has its own stack and its own set of saved registers. The KT and UT appear as a single thread to the programmer because UTs can never block but must always enter the kernel, where an implicit switch to the corresponding KT takes place. UMS uses each UT's TEB to uniquely identify the UT. When a UT enters the kernel, an explicit switch is made to the KT that corresponds to the UT identified by the current TEB. The reason the kernel does not know which UT is running is that UTs can invoke a user-mode scheduler, as fibers do. But in UMS, the scheduler switches UTs, including switching the TEBs.

When a UT enters the kernel, its KT may block. When this happens, the kernel switches to a scheduling thread, which UMS calls a ***primary***, and uses this thread to reenter the user-mode scheduler so that it can pick another UT to run. Eventually, a blocked KT will complete its operation and be ready to return to user mode. Since UMS has already reentered the user-mode scheduler to run a different UT, UMS queues the UT corresponding to the completed KT to a completion list in user mode. When the user-mode scheduler is choosing a new UT to switch to, it can examine the completion list and treat any UT on the list as a candidate for scheduling.

Unlike fibers, UMS is not intended to be used directly by the programmer. The details of writing user-mode schedulers can be very challenging, and UMS does not include such a scheduler. Rather, the schedulers come from programming language libraries that build on top of UMS. Microsoft Visual Studio 2010 shipped with Concurrency Runtime (ConcRT), a concurrent programming framework for C++. ConcRT provides a user-mode scheduler together with facilities for decomposing programs into tasks, which can then be scheduled on the available CPUs. ConcRT provides support for `par_for` styles of con-

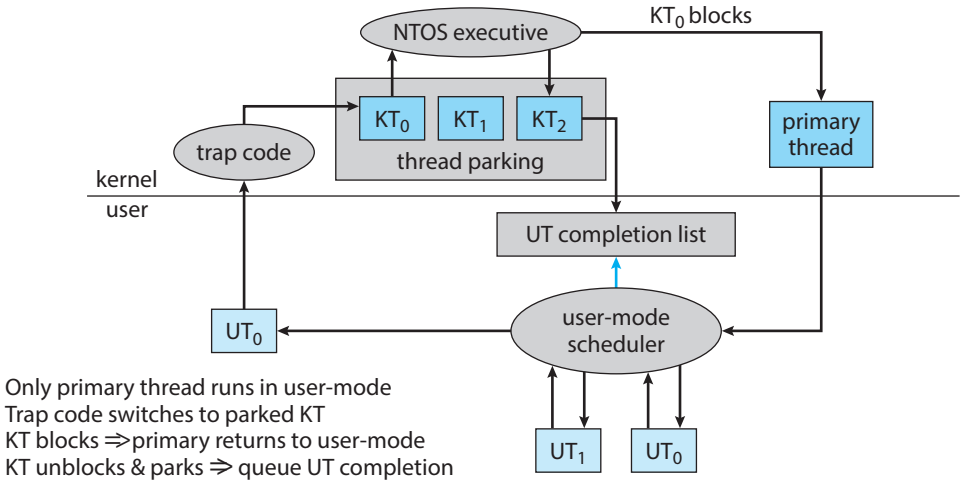


Figure B.11 User-mode scheduling.

structs, as well as rudimentary resource management and task synchronization primitives. The key features of UMS are depicted in Figure B.11.

B.7.3.8 Winsock

Winsock is the Windows sockets API. Winsock is a session-layer interface that is largely compatible with UNIX sockets but has some added Windows 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. Winsock underwent a major update in Windows Vista to add tracing, IPv6 support, impersonation, new security APIs and many other features.

Winsock follows the Windows Open System Architecture (WOSA) model, which provides a standard service provider interface (SPI) between applications and networking protocols. Applications can load and unload ***layered protocols*** that build additional functionality, such as additional security, on top of the transport protocol layers. Winsock supports asynchronous operations and notifications, reliable multicasting, secure sockets, and kernel mode sockets. There is also support for simpler usage models, like the `WSAConnectByName()` function, which accepts the target as strings specifying the name or IP address of the server and the service or port number of the destination port.

B.7.4 IPC Using Windows Messaging

Win32 applications handle interprocess communication in several ways. One way is by using shared kernel objects. Another is by using the Windows messaging facility, an approach that is particularly popular for Win32 GUI applications. One thread can send a message to another thread or to a window by calling `PostMessage()`, `PostThreadMessage()`, `SendMessage()`, `SendThreadMessage()`, or `SendMessageCallback()`. ***Posting*** a message and ***sending*** a message differ in this way: the post routines are asynchronous, they return immediately, and the calling thread does not know when the message

```
// 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 B.12 Code fragments for allocating virtual memory.

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 copies the data to a new block of memory and gives the virtual address of the new block to the receiving process.

Every Win32 thread has its own input queue from which it receives messages. If a Win32 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".

B.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.

B.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. Examples of these functions appear in Figure B.12.

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.

B.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

```
// 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 B.13 Code fragments for memory mapping of a file.

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 B.13.

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 handle duplication.

B.7.5.3 Heaps

Heaps provide a third way for applications to use memory, just as with `malloc()` and `free()` in standard C. A heap in the Win32 environment is a region of reserved address space. When a Win32 process is initialized, it is created with a **default heap**. Since most Win32 applications are multithreaded, 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 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.

The original Win32 heap was optimized for efficient use of space. This led to significant problems with fragmentation of the address space for larger server programs that ran for long periods of time. A new **low-fragmentation heap** (**LFH**) design introduced in Windows XP greatly reduced the fragmen-

```
// reserve a slot for a variable
DWORD var_index = T1sAlloc();
// set it to the value 10
T1sSetValue(var_index, 10);
// get the value
int var T1sGetValue(var_index);
// release the index
T1sFree(var_index);
```

Figure B.14 Code for dynamic thread-local storage.

tation problem. The Windows 7 heap manager automatically turns on LFH as appropriate.

B.7.5.4 Thread-Local Storage

A fourth way for applications to use memory is through a **thread-local storage** (**TLS**) mechanism. Functions that rely on global or static data typically fail to work properly in a multithreaded environment. For instance, the C run-time 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. TLS provides a way to maintain instances of variables that are global to the function being executed but not shared with any other thread.

TLS provides both dynamic and static methods of creating thread-local storage. The dynamic method is illustrated in Figure B.14. The TLS mechanism allocates global heap storage and attaches it to the thread environment block that Windows allocates to every user-mode thread. The TEB is readily accessible by each thread and is used not just for TLS but for all the per-thread state information in user mode.

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;
```

B.8 Summary

Microsoft designed Windows to be an extensible, portable operating system—one able to take advantage of new techniques and hardware. Windows 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 to support a wide variety of application environments. Windows provides virtual memory, integrated caching, and preemptive scheduling. It supports elaborate security mechanisms and includes internationalization features. Windows 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.

Practice Exercises

B.1 What type of operating system is Windows? Describe two of its major features.

B.2 List the design goals of Windows. Describe two in detail.

B.3 Describe the booting process for a Windows system.

B.4 Describe the three main architectural layers of the Windows kernel.

B.5 What is the job of the object manager?

B.6 What types of services does the process manager provide?

B.7 What is a local procedure call?

B.8 What are the responsibilities of the I/O manager?

B.9 What types of networking does Windows support? How does Windows implement transport protocols? Describe two networking protocols.

B.10 How is the NTFS namespace organized?

B.11 How does NTFS handle data structures? How does NTFS recover from a system crash? What is guaranteed after a recovery takes place?

B.12 How does Windows allocate user memory?

B.13 Describe some of the ways in which an application can use memory via the Win32 API.

Further Reading

[Russinovich et al. (2017)] provides an overview of Windows 7 and considerable technical detail about system internals and components. [Brown (2000)] presents details of the security architecture of Windows.

The Microsoft Developer Network Library (http://msdn.microsoft.com) supplies a wealth of information on Windows and other Microsoft products, including documentation of all the published APIs.

[Iseminger (2000)] provides a good reference on the Windows Active Directory. Detailed discussions of writing programs that use the Win32 API appear in [Richter (1997)].

The source code for a 2005 WRK version of the Windows kernel, together with a collection of slides and other CRK curriculum materials, is available from www.microsoft.com/WindowsAcademic for use by universities.

Bibliography

[Brown (2000)] K. Brown, *Programming Windows Security*, Addison-Wesley (2000).

[Iseminger (2000)] D. Iseminger, *Active Directory Services for Microsoft Windows 2000. Technical Reference*, Microsoft Press (2000).

[Richter (1997)] J. Richter, *Advanced Windows*, Microsoft Press (1997).

[Russinovich et al. (2017)] M. Russinovich, D. A. Solomon, and A. Ionescu, *Windows Internals–Part 1*, Seventh Edition, Microsoft Press (2017).

BSD UNIX

This chapter was first written in 1991 and has been updated over time.

In Chapter 20, we presented an in-depth examination of the Linux operating system. In this chapter, we examine another popular UNIX version—UnixBSD. We start by presenting a brief history of the UNIX operating system. We then describe the system's user and programmer interfaces. Finally, we discuss the internal data structures and algorithms used by the FreeBSD kernel to support the user–programmer interface.

C.1 UNIX History

The first version of UNIX was developed in 1969 by Ken Thompson of the Research Group at Bell Laboratories to use an otherwise idle PDP-7. Thompson was soon joined by Dennis Ritchie and they, with other members of the Research Group, produced the early versions of UNIX.

Ritchie had previously worked on the MULTICS project, and MULTICS had a strong influence on the newer operating system. Even the name ***UNIX*** is a pun on ***MULTICS***. The basic organization of the file system, the idea of the command interpreter (or the shell) as a user process, the use of a separate process for each command, the original line-editing characters (# to erase the last character and @ to erase the entire line), and numerous other features came directly from MULTICS. Ideas from other operating systems, such as MIT's CTSS and the XDS-940 system, were also used.

Ritchie and Thompson worked quietly on UNIX for many years. They moved it to a PDP-11/20 for a second version; for a third version, they rewrote most of the operating system in the systems-programming language C, instead of the previously used assembly language. C was developed at Bell Laboratories to support UNIX. UNIX was also moved to larger PDP-11 models, such as the 11/45 and 11/70. Multiprogramming and other enhancements were added when it was rewritten in C and moved to systems (such as the 11/45) that had hardware support for multiprogramming.

As UNIX developed, it became widely used within Bell Laboratories and gradually spread to a few universities. The first version widely available out-

side Bell Laboratories was Version 6, released in 1976. (The version number for early UNIX systems corresponds to the edition number of the *UNIX Programmer's Manual* that was current when the distribution was made; the code and the manual were revised independently.)

In 1978, Version 7 was distributed. This UNIX system ran on the PDP-11/70 and the Interdata 8/32 and is the ancestor of most modern UNIX systems. In particular, it was soon ported to other PDP-11 models and to the VAX computer line. The version available on the VAX was known as 32V. Research has continued since then.

C.1.1 UNIX Support Group

After the distribution of Version 7 in 1978, the UNIX Support Group (USG) assumed administrative control and responsibility from the Research Group for distributions of UNIX within AT&T, the parent organization for Bell Laboratories. UNIX was becoming a product, rather than simply a research tool. The Research Group continued to develop their own versions of UNIX, however, to support their internal computing. Version 8 included a facility called the **stream I/O system**, which allows flexible configuration of kernel IPC modules. It also contained RFS, a remote file system similar to Sun's NFS. The current version is Version 10, released in 1989 and available only within Bell Laboratories.

USG mainly provided support for UNIX within AT&T. The first external distribution from USG was System III, in 1982. System III incorporated features of Version 7 and 32V, as well as features of several UNIX systems developed by groups other than Research. For example, features of UNIX/RT, a real-time UNIX system, and numerous portions of the Programmer's Work Bench (PWB) software tools package were included in System III.

USG released System V in 1983; it is largely derived from System III. The divestiture of the various Bell operating companies from AT&T left AT&T in a position to market System V aggressively. USG was restructured as the UNIX System Development Laboratory (USDL), which released UNIX System V Release 2 (V.2) in 1984. UNIX System V Release 2, Version 4 (V.2.4) added a new implementation of virtual memory with copy-on-write paging and shared memory. USDL was in turn replaced by AT&T Information Systems (ATTIS), which distributed System V Release 3 (V.3) in 1987. V.3 adapts the V8 implementation of the stream I/O system and makes it available as ***STREAMS***. It also includes RFS, the NFS-like remote file system mentioned earlier.

C.1.2 Berkeley Begins Development

The small size, modularity, and clean design of early UNIX systems led to UNIX-based work at numerous other computer-science organizations, such as RAND, BBN, the University of Illinois, Harvard, Purdue, and DEC. The most influential UNIX development group outside of Bell Laboratories and AT&T, however, has been the University of California at Berkeley.

Bill Joy and Ozalp Babaoglu did the first Berkeley VAX UNIX work in 1978. They added virtual memory, demand paging, and page replacement to 32V to produce 3BSD UNIX. This version was the first to implement any of these facilities on a UNIX system. The large virtual memory space of 3BSD allowed the development of very large programs, such as Berkeley's own Franz LISP. The memory-management work convinced the Defense Advanced Research

Projects Agency (DARPA) to fund Berkeley for the development of a standard UNIX system for government use; 4BSD UNIX was the result.

The 4 BSD work for DARPA was guided by a steering committee that included many notable people from the UNIX and networking communities. One of the goals of this project was to provide support for the DARPA Internet networking protocols (TCP/IP). This support was provided in a general manner. It is possible in 4.2 BSD to communicate uniformly among diverse network facilities, including local-area networks (such as Ethernets and token rings) and wide-area networks (such as NSFNET). This implementation was the most important reason for the current popularity of these protocols. Many vendors of UNIX computer systems used it as the basis for their implementations, and it was even used in other operating systems. It permitted the Internet to grow from 60 connected networks in 1984 to more than 8,000 networks and an estimated 10 million users in 1993.

In addition, Berkeley adapted many features from contemporary operating systems to improve the design and implementation of UNIX. Many of the terminal line-editing functions of the TENEX (TOPS-20) operating system were provided by a new terminal driver. A new user interface (the C Shell), a new text editor (ex/vi), compilers for Pascal and LISP, and many new systems programs were written at Berkeley. For 4.2 BSD, certain efficiency improvements were inspired by the VMS operating system.

UNIX software from Berkeley was released in **Berkeley Software Distributions (BSD)**. It is convenient to refer to the Berkeley VAX UNIX systems following 3 BSD as 4 BSD, but there were actually several specific releases, most notably 4.1 BSD and 4.2 BSD; 4.2 BSD, first distributed in 1983, was the culmination of the original Berkeley DARPA UNIX project. The equivalent version for PDP-11 systems was 2.9 BSD.

In 1986, 4.3 BSD was released. It was very similar to 4.2 BSD but included numerous internal changes, such as bug fixes and performance improvements. Some new facilities were also added, including support for the Xerox Network System protocols.

The next version was 4.3 BSD Tahoe, released in 1988. It included improved networking congestion control and TCP/IP performance. Disk configurations were separated from the device drivers and read off the disks themselves. Expanded time-zone support was also included. 4.3 BSD Tahoe was actually developed on and for the CCI Tahoe system (Computer Console, Inc., Power 6 computer), rather than for the usual VAX base. The corresponding PDP-11 release was 2.10.1BSD; it was distributed by the USENIX association, which also published the 4.3 BSD manuals. The 4.3.2 BSD Reno release saw the inclusion of an implementation of ISO/OSI networking.

The last Berkeley release, 4.4 BSD, was finalized in June of 1993. It included new X.25 networking support and POSIX standard compliance. It also had a radically new file system organization, with a new virtual file system interface and support for ***stackable*** file systems, allowing file systems to be layered on top of each other for easy inclusion of new features. An implementation of NFS was included in the release (Section 15.8), along with a new log-based file system (see Chapter 11). The 4.4 BSD virtual memory system was derived from Mach (described in Section A.13). Several other changes, such as enhanced security and improved kernel structure, were also included. With the release of version 4.4, Berkeley halted its research efforts.

C.1.3 The Spread of UNIX

UNIX 4 BSD was the operating system of choice for the VAX from its initial release (in 1979) until the release of Ultrix, DEC's BSD implementation. Indeed, 4 BSD is still the best choice for many research and networking installations. The current set of UNIX operating systems is not limited to those from Bell Laboratories (which is currently owned by Lucent Technology) and Berkeley, however. Sun Microsystems helped popularize the BSD flavor of UNIX by shipping it on Sun workstations. As UNIX grew in popularity, it was moved to many computers and computer systems. A wide variety of UNIX and UNIX-like operating systems have been created. DEC supported its UNIX (Ultrix) on its workstations and is replacing Ultrix with another UNIX-derived operating system, OSF/1. Microsoft rewrote UNIX for the Intel 8088 family and called it XENIX, and its Windows NT operating system was heavily influenced by UNIX. IBM has UNIX (AIX) on its PCs, workstations, and mainframes. In fact, UNIX is available on almost all general-purpose computers. It runs on personal computers, workstations, minicomputers, mainframes, and supercomputers, from Apple Macintosh IIs to Cray IIs. Because of its wide availability, it is used in environments ranging from academic to military to manufacturing process control. Most of these systems are based on Version 7, System III, 4.2 BSD, or System V.

The wide popularity of UNIX with computer vendors has made UNIX the most portable of operating systems, and users can expect a UNIX environment independent of any specific computer manufacturer. But the large number of implementations of the system has led to remarkable variation in the programming and user interfaces distributed by the vendors. For true vendor independence, application-program developers need consistent interfaces. Such interfaces would allow all "UNIX" applications to run on all UNIX systems, which is certainly not the current situation. This issue has become important as UNIX has become the preferred program-development platform for applications ranging from databases to graphics and networking, and it has led to a strong market demand for UNIX standards.

Several standardization projects have been undertaken. The first was the ***/usr/group 1984 Standard***, sponsored by the UniForum industry user's group. Since then, many official standards bodies have continued the effort, including IEEE and ISO (the POSIX standard). The X/Open Group international consortium completed XPG3, a Common Application Environment, which subsumes the IEEE interface standard. Unfortunately, XPG3 is based on a draft of the ANSI C standard rather than the final specification, and therefore needed to be redone as XPG4. In 1989, the ANSI standards body standardized the C programming language, producing an ANSI C specification that vendors were quick to adopt.

As such projects continue, the flavors of UNIX will converge and lead to one programming interface to UNIX, allowing UNIX to become even more popular. In fact, two separate sets of powerful UNIX vendors are working on this problem: The AT&T-guided UNIX International (UI) and the Open Software Foundation (OSF) have both agreed to follow the POSIX standard. Recently, many of the vendors involved in those two groups have agreed on further standardization (the COSE agreement).

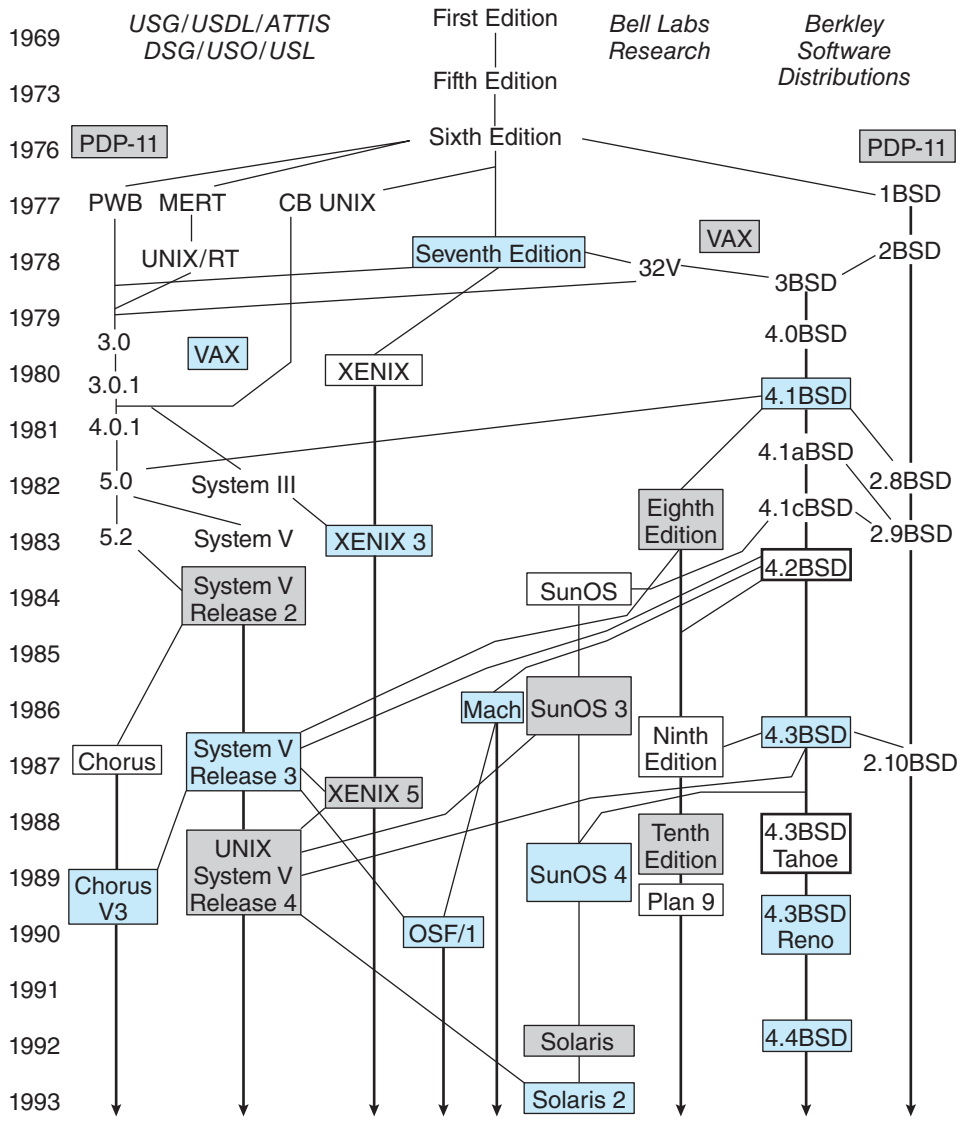


Figure C.1 History of UNIX versions up to 1993.

AT&T replaced its ATTIS group in 1989 with the UNIX Software Organization (USO), which shipped the first merged UNIX, System V Release 4. This system combines features from System V, 4.3 BSD, and Sun's SunOS, including long file names, the Berkeley file system, virtual memory management, symbolic links, multiple access groups, job control, and reliable signals; it also conforms to the published POSIX standard, POSIX.1. After USO produced SVR4, it became an independent AT&T subsidiary named Unix System Laboratories (USL); in 1993, it was purchased by Novell, Inc. Figure C.1 summarizes the relationships among the various versions of UNIX.

The UNIX system has grown from a personal project of two Bell Laboratories employees to an operating system defined by multinational standardization bodies. At the same time, UNIX is an excellent vehicle for academic study, and we believe it will remain an important part of operating-system theory and practice. For example, the Tunis operating system, the Xinu operating system, and the Minix operating system are based on the concepts of UNIX but were developed explicitly for classroom study. There is a plethora of ongoing UNIX-related research systems, including Mach, Chorus, Comandos, and Roisin. The original developers, Ritchie and Thompson, were honored in 1983 by the Association for Computing Machinery Turing Award for their work on UNIX.

C.1.4 History of FreeBSD

The specific UNIX version used in this chapter is the Intel version of FreeBSD. This system implements many interesting operating-system concepts, such as demand paging with clustering, as well as networking. The FreeBSD project began in early 1993 to produce a snapshot of 386 BSD to solve problems that could not be resolved using the existing patch mechanism. 386 BSD was derived from 4.3 BSD-Lite (Net/2) and was released in June 1992 by William Jolitz. FreeBSD (coined by David Greenman) 1.0 was released in December 1993, and FreeBSD 1.1 was released in May 1994. Both versions were based on 4.3 BSD-Lite. Legal issues between UCB and Novell required that 4.3 BSD-Lite code no longer be used, so the final 4.3 BSD-Lite release was made in July 1994 (FreeBSD 1.1.5.1).

FreeBSD was then reinvented based on 4.4BSD-Lite code, which was incomplete. FreeBSD 2.0 was released in November 1994. Later releases included 2.0.5 in June 1995, 2.1.5 in August 1996, 2.1.7.1 in February 1997, 2.2.1 in April 1997, 2.2.8 in November 1998, 3.0 in October 1998, 3.1 in February 1999, 3.2 in May 1999, 3.3 in September 1999, 3.4 in December 1999, 3.5 in June 2000, 4.0 in March 2000, 4.1 in July 2000, and 4.2 in November 2000.

The goal of the FreeBSD project is to provide software that can be used for any purpose with no strings attached. The idea is that the code will get the widest possible use and provide the most benefit. At present, it runs primarily on Intel platforms, although Alpha platforms are supported. Work is underway to port to other processor platforms as well.

C.2 Design Principles

UNIX was designed to be a time-sharing system. The standard user interface (the shell) is simple and can be replaced by another, if desired. The file system is a multilevel tree, which allows users to create their own subdirectories. Each user data file is simply a sequence of bytes.

Disk files and I/O devices are treated as similarly as possible. Thus, device dependencies and peculiarities are kept in the kernel as much as possible. Even in the kernel, most of them are confined to the device drivers.

UNIX supports multiple processes. A process can easily create new processes. CPU scheduling is a simple priority algorithm. FreeBSD uses demand

paging as a mechanism to support memory-management and CPU-scheduling decisions. Swapping is used if a system is suffering from excess paging.

Because UNIX was originated by Thompson and Ritchie as a system for their own convenience, it was small enough to understand. Most of the algorithms were selected for ***simplicity***, not for speed or sophistication. The intent was to have the kernel and libraries provide a small set of facilities that was sufficiently powerful to allow a person to build a more complex system if needed. UNIX's clean design has resulted in many imitations and modifications.

Although the designers of UNIX had a significant amount of knowledge about other operating systems, UNIX had no elaborate design spelled out before its implementation. This flexibility appears to have been one of the key factors in the development of the system. Some design principles were involved, however, even though they were not made explicit at the outset.

The UNIX system was designed by programmers for programmers. Thus, it has always been interactive, and facilities for program development have always been a high priority. Such facilities include the program ***make*** (which can be used to check which of a collection of source files for a program need to be compiled and then to do the compiling) and the ***Source Code Control System (SCCS)*** (which is used to keep successive versions of files available without having to store the entire contents of each step). The primary version-control system used by UNIX is the ***Concurrent Versions System*** (CVS) due to the large number of developers operating on and using the code.

The operating system is written mostly in C, which was developed to support UNIX, since neither Thompson nor Ritchie enjoyed programming in assembly language. The avoidance of assembly language was also necessary because of the uncertainty about the machines on which UNIX would be run. It has greatly simplified the problems of moving UNIX from one hardware system to another.

From the beginning, UNIX development systems have had all the UNIX sources available online, and the developers have used the systems under development as their primary systems. This pattern of development has greatly facilitated the discovery of deficiencies and their fixes, as well as of new possibilities and their implementations. It has also encouraged the plethora of UNIX variants existing today, but the benefits have outweighed the disadvantages. If something is broken, it can be fixed at a local site; there is no need to wait for the next release of the system. Such fixes, as well as new facilities, may be incorporated into later distributions.

The size constraints of the PDP-11 (and earlier computers used for UNIX) have forced a certain elegance. Where other systems have elaborate algorithms for dealing with pathological conditions, UNIX just does a controlled crash called ***panic***. Instead of attempting to cure such conditions, UNIX tries to prevent them. Where other systems would use brute force or macro-expansion, UNIX mostly has had to develop more subtle, or at least simpler, approaches.

These early strengths of UNIX produced much of its popularity, which in turn produced new demands that challenged those strengths. UNIX was used for tasks such as networking, graphics, and real-time operation, which did not always fit into its original text-oriented model. Thus, changes were made to certain internal facilities, and new programming interfaces were added. Supporting these new facilities and others—particularly window interfaces

—required large amounts of code, radically increasing the size of the system. For instance, both networking and windowing doubled the size of the system. This pattern in turn pointed out the continued strength of UNIX—whenever a new development occurred in the industry, UNIX could usually absorb it but remain UNIX.

C.3 Programmer Interface

Like most operating systems, UNIX consists of two separable parts: the kernel and the systems programs. We can view the UNIX operating system as being layered, as shown in Figure C.2. Everything below the system-call interface and above the physical hardware is the ***kernel***. The kernel provides the file system, CPU scheduling, memory management, and other operating-system functions through system calls. Systems programs use the kernel-supported system calls to provide useful functions, such as compilation and file manipulation.

System calls define the ***programmer interface*** to UNIX. The set of systems programs commonly available defines the ***user interface***. The programmer and user interface define the context that the kernel must support.

Most systems programs are written in C, and the ***UNIX Programmer's Manual*** presents all system calls as C functions. A system program written in C for FreeBSD on the Pentium can generally be moved to an Alpha FreeBSD system and simply recompiled, even though the two systems are quite different. The details of system calls are known only to the compiler. This feature is a major reason for the portability of UNIX programs.

System calls for UNIX can be roughly grouped into three categories: file manipulation, process control, and information manipulation. In Chapter 2, we listed a fourth category, device manipulation, but since devices in UNIX are treated as (special) files, the same system calls support both files and devices (although there is an extra system call for setting device parameters).

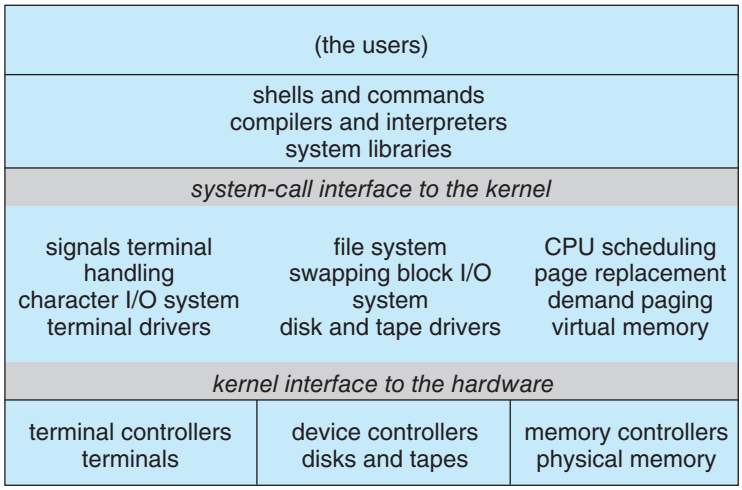


Figure C.2 4.4BSD layer structure.

C.3.1 File Manipulation

A ***file*** in UNIX is a sequence of bytes. Different programs expect various levels of structure, but the kernel does not impose a structure on files. For instance, the convention for text files is lines of ASCII characters separated by a single newline character (which is the linefeed character in ASCII), but the kernel knows nothing of this convention.

Files are organized in tree-structured ***directories***. Directories are themselves files that contain information on how to find other files. A ***path name*** to a file is a text string that identifies a file by specifying a path through the directory structure to the file. Syntactically, it consists of individual file-name elements separated by the slash character. For example, in */usr/local/font*, the first slash indicates the root of the directory tree, called the ***root directory***. The next element, *usr,* is a subdirectory of the root, *local* is a subdirectory of *usr*, and *font* is a file or directory in the directory ***local***. Whether *font* is an ordinary file or a directory cannot be determined from the path-name syntax.

The UNIX file system has both ***absolute path names*** and ***relative path names***. Absolute path names start at the root of the file system and are distinguished by a slash at the beginning of the path name; */usr/local/font* is an absolute path name. Relative path names start at the ***current directory***, which is an attribute of the process accessing the path name. Thus, *local/font* indicates a file or directory named *font* in the directory *local* in the current directory, which might or might not be */usr.*

A file may be known by more than one name in one or more directories. Such multiple names are known as ***links***, and all links are treated equally by the operating system. FreeBSD also supports ***symbolic links***, which are files containing the path name of another file. The two kinds of links are also known as ***hard links*** and ***soft links***. Soft (symbolic) links, unlike hard links, may point to directories and may cross file-system boundaries.

The file name "." in a directory is a hard link to the directory itself. The file name ".." is a hard link to the parent directory. Thus, if the current directory is */user/jlp/programs,* then *../bin/wdf* refers to */user/jlp/bin/wdf.*

Hardware devices have names in the file system. These ***device special files*** or ***special files*** are known to the kernel as device interfaces, but they are nonetheless accessed by the user by much the same system calls as are other files.

Figure C.3 shows a typical UNIX file system. The root (/) normally contains a small number of directories as well as */kernel,* the binary boot image of the operating system; */dev* contains the device special files, such as */dev/console, /dev/lp0, /dev/mt0,* and so on; and */bin* contains the binaries of the essential UNIX systems programs. Other binaries may be in */usr/bin* (for applications systems programs, such as text formatters), */usr/compat* (for programs from other operating systems, such as Linux), or */usr/local/bin* (for systems programs written at the local site). Library files—such as the C, Pascal, and FORTRAN subroutine libraries—are kept in */lib* (or */usr/lib* or */usr/local/lib*).

The files of users themselves are stored in a separate directory for each user, typically in */usr.* Thus, the user directory for *carol* would normally be in */usr/carol.* For a large system, these directories may be further grouped to ease administration, creating a file structure with */usr/prof/avi* and */usr/staff/carol.* Administrative files and programs, such as the password file, are kept in */etc.*

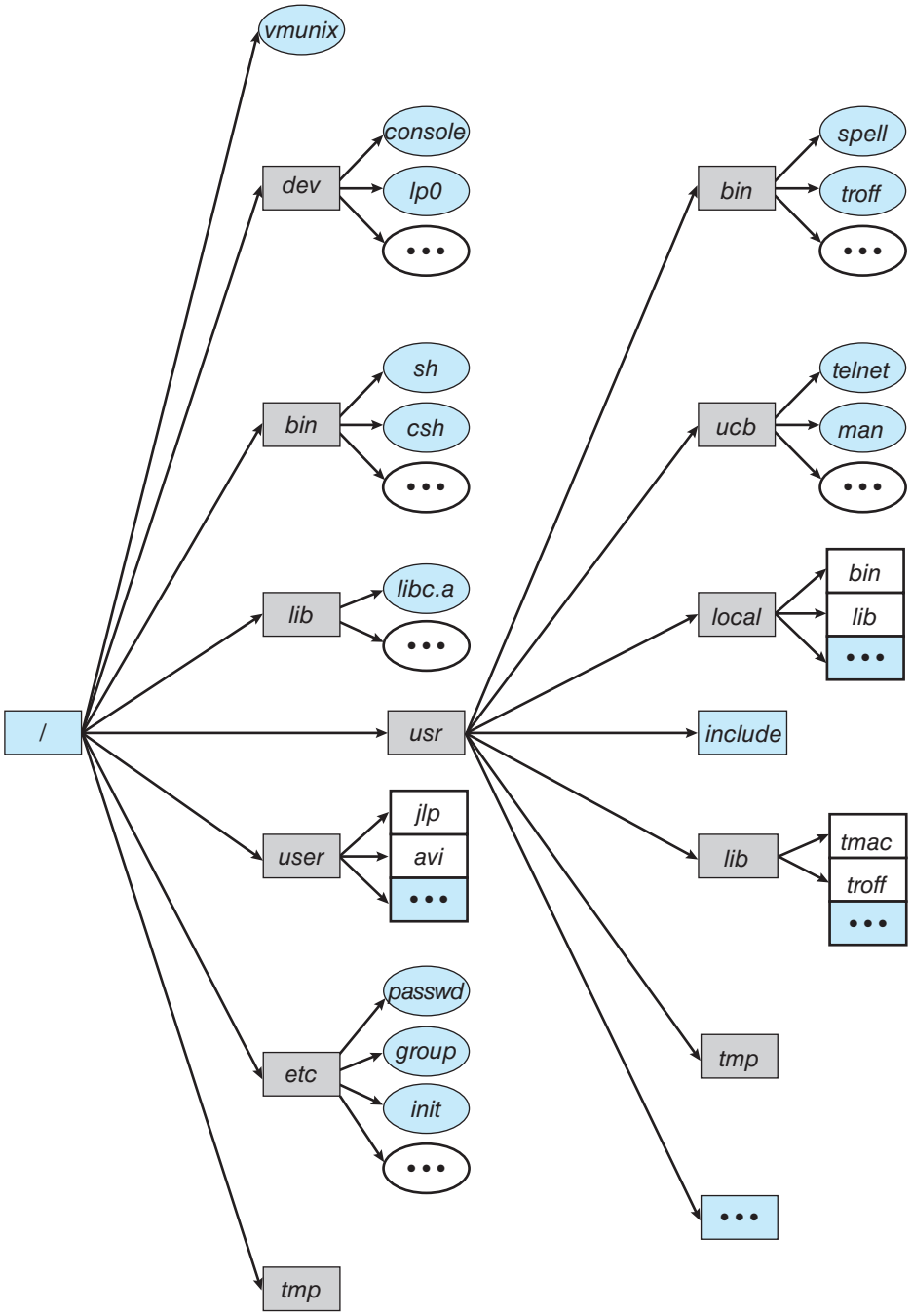


Figure C.3 Typical UNIX directory structure.

Temporary files can be put in */tmp*, which is normally erased during system boot, or in */usr/tmp*.

Each of these directories may have considerably more structure. For example, the font-description tables for the troff formatter for the Merganthaler 202

typesetter are kept in */usr/lib/troff/dev202*. All the conventions concerning the location of specific files and directories have been defined by programmers and their programs. The operating-system kernel needs only */etc/init*, which is used to initialize terminal processes, to be operable.

System calls for basic file manipulation are `creat()`, `open()`, `read()`, `write()`, `close()`, `unlink()`, and `trunc()`. The `creat()` system call, given a path name, creates an empty file (or truncates an existing one). An existing file is opened by the `open()` system call, which takes a path name and a mode (such as read, write, or read–write) and returns a small integer, called a ***file descriptor***. The file descriptor may then be passed to a `read()` or `write()` system call (along with a buffer address and the number of bytes to transfer) to perform data transfers to or from the file. A file is closed when its file descriptor is passed to the `close()` system call. The `trunc()` call reduces the length of a file to 0.

A file descriptor is an index into a small table of open files for this process. Descriptors start at 0 and seldom get higher than 6 or 7 for typical programs, depending on the maximum number of simultaneously open files.

Each `read()` or `write()` updates the current offset into the file, which is associated with the file-table entry and is used to determine the position in the file for the next `read()` or `write()`. The `lseek()` system call allows the position to be reset explicitly. It also allows the creation of sparse files (files with "holes" in them). The `dup()` and `dup2()` system calls can be used to produce a new file descriptor that is a copy of an existing one. The `fcntl()` system call can also do that and in addition can examine or set various parameters of an open file. For example, it can make each succeeding `write()` to an open file append to the end of that file. There is an additional system call, `ioctl()`, for manipulating device parameters. It can set the baud rate of a serial port or rewind a tape, for instance.

Information about the file (such as its size, protection modes, owner, and so on) can be obtained by the `stat()` system call. Several system calls allow some of this information to be changed: `rename()` (change file name), `chmod()` (change the protection mode), and `chown()` (change the owner and group). Many of these system calls have variants that apply to file descriptors instead of file names. The `link()` system call makes a hard link for an existing file, creating a new name for an existing file. A link is removed by the `unlink(())` system call; if it is the last link, the file is deleted. The `symlink()` system call makes a symbolic link.

Directories are made by the `mkdir()` system call and are deleted by `rmdir()`. The current directory is changed by `cd()`.

Although the standard file calls (`open()` and others) can be used on directories, it is inadvisable to do so, since directories have an internal structure that must be preserved. Instead, another set of system calls is provided to open a directory, to step through each file entry within the directory, to close the directory, and to perform other functions; these are `opendir()`, `readdir()`, `closedir()`, and others.

C.3.2 Process Control

A ***process*** is a program in execution. Processes are identified by their ***process identifier***, which is an integer. A new process is created by the `fork()` system

call. The new process consists of a copy of the address space of the original process (the same program and the same variables with the same values). Both processes (the parent and the child) continue execution at the instruction after the `fork()` with one difference: the return code for the `fork()` is zero for the new (child) process, whereas the (nonzero) process identifier of the child is returned to the parent.

Typically, the `execve()` system call is used after a fork by one of the two processes to replace that process's virtual memory space with a new program. The `execve()` system call loads a binary file into memory (destroying the memory image of the program containing the `execve()` system call) and starts its execution.

A process may terminate by using the `exit()` system call, and its parent process may wait for that event by using the `wait()` system call. If the child process crashes, the system simulates the `exit()` call. The `wait()` system call provides the process ID of a terminated child so that the parent can tell which of possibly many children terminated. A second system call, `wait3()`, is similar to `wait()` but also allows the parent to collect performance statistics about the child. Between the time the child exits and the time the parent completes one of the `wait()` system calls, the child is ***defunct***. A defunct process can do nothing but exists merely so that the parent can collect its status information. If the parent process of a defunct process exits before a child, the defunct process is inherited by the `init` process (which in turn waits on it) and becomes a ***zombie*** process. A typical use of these facilities is shown in Figure C.4.

The simplest form of communication between processes is by ***pipes***. A pipe may be created before the `fork()`, and its endpoints are then set up between the `fork()` and the `execve()`. A pipe is essentially a queue of bytes between two processes. The pipe is accessed by a file descriptor, like an ordinary file. One process writes into the pipe, and the other reads from the pipe. The size of the original pipe system was fixed by the system. With FreeBSD pipes are implemented on top of the socket system, which has variable-sized buffers. Reading from an empty pipe or writing into a full pipe causes the process to be blocked until the state of the pipe changes. Special arrangements are needed for a pipe to be placed between a parent and child (so only one is reading and one is writing).

All user processes are descendants of one original process, called `init` (which has process identifier 1). Each terminal port available for interactive use has a `getty` process forked for it by `init`. The `getty` process initializes terminal line parameters and waits for a user's login name, which it passes through

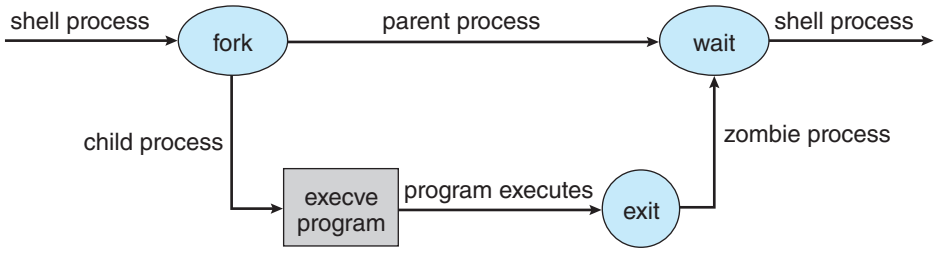


Figure C.4 A shell forks a subprocess to execute a program.

an `execve()` as an argument to a `login` process. The `login` process collects the user's password, encrypts it, and compares the result to an encrypted string taken from the file */etc/passwd*. If the comparison is successful, the user is allowed to log in. The `login` process executes a ***shell***, or command interpreter, after setting the numeric ***user identifier*** of the process to that of the user logging in. (The shell and the user identifier are found in */etc/passwd* by the user's login name.) It is with this shell that the user ordinarily communicates for the rest of the login session. The shell itself forks subprocesses for the commands the user tells it to execute.

The user identifier is used by the kernel to determine the user's permissions for certain system calls, especially those involving file accesses. There is also a ***group identifier***, which is used to provide similar privileges to a collection of users. In FreeBSD a process may be in several groups simultaneously. The `login` process puts the shell in all the groups permitted to the user by the files */etc/passwd* and */etc/group*.

Two user identifiers are used by the kernel: the effective user identifier and the real user identifier. The ***effective user identifier*** is used to determine file access permissions. If the file of a program being loaded by an `execve()` has the `setuid` bit set in its inode, the effective user identifier of the process is set to the user identifier of the owner of the file, whereas the ***real user identifier*** is left as it was. This scheme allows certain processes to have more than ordinary privileges while still being executable by ordinary users. The setuid idea was patented by Dennis Ritchie (U.S. Patent 4,135,240) and is one of the distinctive features of UNIX. A similar `setgid` bit exists for groups. A process may determine its real and effective user identifier with the `getuid()` and `geteuid()` calls, respectively. The `getgid()` and `getegid()` calls determine the process's real and effective group identifier, respectively. The rest of a process's groups may be found with the `getgroups()` system call.

C.3.3 Signals

Signals are a facility for handling exceptional conditions similar to software interrupts. There are 20 different signals, each corresponding to a distinct condition. A signal may be generated by a keyboard interrupt, by an error in a process (such as a bad memory reference), or by a number of asynchronous events (such as timers or job-control signals from the shell). Almost any signal may also be generated by the `kill()` system call.

The `interrupt` signal, SIGINT, is used to stop a command before that command completes. It is usually produced by the ^C character (ASCII 3). As of 4.2BSD, the important keyboard characters are defined by a table for each terminal and can be redefined easily. The `quit` signal, SIGQUIT, is usually produced by the ^bs character (ASCII 28). The `quit` signal both stops the currently executing program and dumps its current memory image to a file named *core* in the current directory. The core file can be used by debuggers. SIGILL is produced by an illegal instruction and SIGSEGV by an attempt to address memory outside of the legal virtual memory space of a process.

Arrangements can be made either for most signals to be ignored (to have no effect) or for a routine in the user process (a signal handler) to be called. A signal handler may safely do one of two things before returning from catching a signal: call the `exit()` system call or modify a global variable. One signal

(the `kill` signal, number 9, SIGKILL) cannot be ignored or caught by a signal handler. SIGKILL is used, for example, to kill a runaway process that is ignoring other signals such as SIGINT and SIGQUIT.

Signals can be lost. If another signal of the same kind is sent before a previous signal has been accepted by the process to which it is directed, the first signal will be overwritten, and only the last signal will be seen by the process. In other words, a call to the signal handler tells a process that there has been at least one occurrence of the signal. Also, there is no relative priority among UNIX signals. If two different signals are sent to the same process at the same time, we cannot know which one the process will receive first.

Signals were originally intended to deal with exceptional events. As is true of most UNIX features, however, signal use has steadily expanded. 4.1BSD introduced job control, which uses signals to start and stop subprocesses on demand. This facility allows one shell to control multiple processes—starting, stopping, and backgrounding them as the user wishes. The SIGWINCH signal, invented by Sun Microsystems, is used for informing a process that the window in which output is being displayed has changed size. Signals are also used to deliver urgent data from network connections.

Users wanted more reliable signals and a bug fix in an inherent race condition in the old signal implementation. Thus, 4.2BSD brought with it a race-free, reliable, separately implemented signal capability. It allows individual signals to be blocked during critical sections, and it has a new system call to let a process sleep until interrupted. It is similar to hardware-interrupt functionality. This capability is now part of the POSIX standard.

C.3.4 Process Groups

Groups of related processes frequently cooperate to accomplish a common task. For instance, processes may create, and communicate over, pipes. Such a set of processes is termed a ***process group***, or a ***job***. Signals may be sent to all processes in a group. A process usually inherits its process group from its parent, but the `setpgrp()` system call allows a process to change its group.

Process groups are used by the C shell to control the operation of multiple jobs. Only one process group may use a terminal device for I/O at any time. This ***foreground*** job has the attention of the user on that terminal, while all other nonattached jobs (***background*** jobs) perform their functions without user interaction. Access to the terminal is controlled by process group signals. Each job has a ***controlling terminal*** (again, inherited from its parent). If the process group of the controlling terminal matches the group of a process, that process is in the foreground and is allowed to perform I/O. If a nonmatching (background) process attempts the same, a SIGTTIN or SIGTTOU signal is sent to its process group. This signal usually causes the process group to freeze until it is foregrounded by the user, at which point it receives a SIGCONT signal, indicating that the process can perform the I/O. Similarly, a SIGSTOP may be sent to the foreground process group to freeze it.

C.3.5 Information Manipulation

System calls exist to set and return both an interval timer (`getitimer()`/`setitimer()`) and the current time (`gettimeofday()`/`settimeofday()`) in

microseconds. In addition, processes can ask for their process identifier (`getpid()`), their group identifier (`getgid()`), the name of the machine on which they are executing (`gethostname()`), and many other values.

C.3.6 Library Routines

The system-call interface to UNIX is supported and augmented by a large collection of library routines and header files. The header files provide the definition of complex data structures used in system calls. In addition, a large library of functions provides additional program support.

For example, the UNIX I/O system calls provide for the reading and writing of blocks of bytes. Some applications may want to read and write only 1 byte at a time. Although possible, that would require a system call for each byte—a very high overhead. Instead, a set of standard library routines (the standard I/O package accessed through the header file <*stdio.h*>) provides another interface, which reads and writes several thousand bytes at a time using local buffers and transfers between these buffers (in user memory) when I/O is desired. Formatted I/O is also supported by the standard I/O package.

Additional library support is provided for mathematical functions, network access, data conversion, and so on. The FreeBSD kernel supports over 300 system calls; the C program library has over 300 library functions. The library functions eventually result in system calls where necessary (for example, the `getchar()` library routine will result in a `read()` system call if the file buffer is empty). However, the programmer generally does not need to distinguish between the basic set of kernel system calls and the additional functions provided by library functions.

C.4 User Interface

Both the programmer and the user of a UNIX system deal mainly with the set of systems programs that have been written and are available for execution. These programs make the necessary system calls to support their function, but the system calls themselves are contained within the program and do not need to be obvious to the user.

The common systems programs can be grouped into several categories; most of them are file or directory oriented. For example, the systems programs to manipulate directories are `mkdir` to create a new directory, `rmdir` to remove a directory, `cd` to change the current directory to another, and `pwd` to print the absolute path name of the current (working) directory.

The `ls` program lists the names of the files in the current directory. Any of 28 options can ask that properties of the files be displayed also. For example, the `-l` option asks for a long listing showing the file name, owner, protection, date and time of creation, and size. The `cp` program creates a new file that is a copy of an existing file. The `mv` program moves a file from one place to another in the directory tree. In most cases, this move simply requires a renaming of the file. If necessary, however, the file is copied to the new location, and the old copy is deleted. A file is deleted by the `rm` program (which makes an `unlink()` system call).

To display a file on the terminal, a user can run `cat`. The `cat` program takes a list of files and concatenates them, copying the result to the standard output, commonly the terminal. On a high-speed cathode-ray tube (CRT) display, of course, the file may speed by too fast to be read. The `more` program displays the file one screen at a time, pausing until the user types a character to continue to the next screen. The `head` program displays just the first few lines of a file; `tail` shows the last few lines.

These are the basic systems programs widely used in UNIX. In addition, there are a number of editors (`ed`, `sed`, `emacs`, `vi`, and so on), compilers (C, python, FORTRAN, and so on), and text formatters (troff, TEX, scribe, and so on). There are also programs for sorting (`sort`) and comparing files (`cmp`, `diff`), looking for patterns (`grep`, `awk`), sending mail to other users (`mail`), and many other activities.

C.4.1 Shells and Commands

Both user-written and systems programs are normally executed by a command interpreter. The command interpreter in UNIX is a user process like any other. As noted earlier, it is called a shell—because it surrounds the kernel of the operating system. Users can write their own shells, and, in fact, several shells are in general use. The ***Bourne shell***, written by Steve Bourne, is probably the most widely used—or, at least, the most widely available. The ***C shell***, mostly the work of Bill Joy, a founder of Sun Microsystems, is the most popular on BSD systems. The Korn shell, by Dave Korn, has become popular because it combines the features of the Bourne shell and the C shell.

The common shells share much of their command-language syntax. UNIX is normally an interactive system. The shell indicates its readiness to accept another command by typing a prompt, and the user types a command on a single line. For instance, in the line

% ***ls -l***

the percent sign is the usual C shell prompt, and the `ls -l` (typed by the user) is the (long) list-directory command. Commands can take arguments, which the user types after the command name on the same line, separated by white space (spaces or tabs).

Although a few commands are built into the shells (such as `cd`), a typical command is an executable binary object file. A list of several directories, the ***search path***, is kept by the shell. For each command, each of the directories in the search path is searched, in order, for a file of the same name. If a file is found, it is loaded and executed. The search path can be set by the user. The directories */bin* and */usr/bin* are almost always in the search path, and a typical search path on a FreeBSD system might be

(. /usr/avi/bin /usr/local/bin /bin /usr/bin)

The `ls` command's object file is */bin/ls*, and the shell itself is */bin/sh* (the Bourne shell) or */bin/csh* (the C shell).

Execution of a command is done by a `fork()` system call followed by an `execve()` of the object file. The shell usually then does a `wait()` to suspend

its own execution until the command completes (Figure C.4). There is a simple syntax (an ampersand [&] at the end of the command line) to indicate that the shell should *not* wait for the completion of the command. A command left running in this manner while the shell continues to interpret further commands is said to be a ***background*** command, or to be running in the background. Processes for which the shell *does* wait are said to run in the ***foreground***.

The C shell in FreeBSD systems provides a facility called ***job control*** (partially implemented in the kernel), as mentioned previously. Job control allows processes to be moved between the foreground and the background. The processes can be stopped and restarted on various conditions, such as a background job wanting input from the user's terminal. This scheme allows most of the control of processes provided by windowing or layering interfaces but requires no special hardware. Job control is also useful in window systems, such as the X Window System developed at MIT. Each window is treated as a terminal, allowing multiple processes to be in the foreground (one per window) at any one time. Of course, background processes may exist on any of the windows. The Korn shell also supports job control, and job control (and process groups) will likely be standard in future versions of UNIX.

C.4.2 Standard I/O

Processes can open files as they like, but most processes expect three file descriptors (numbers 0, 1, and 2) to be open when they start. These file descriptors are inherited across the `fork()` (and possibly the `execve()`) that created the process. They are known as ***standard input*** (0), ***standard output*** (1), and ***standard error*** (2). All three are frequently open to the user's terminal. Thus, the program can read what the user types by reading standard input, and the program can send output to the user's screen by writing to standard output. The standard-error file descriptor is also open for writing and is used for error output; standard output is used for ordinary output. Most programs can also accept a file (rather than a terminal) for standard input and standard output. The program does not care where its input is coming from and where its output is going. This is one of the elegant design features of UNIX.

The common shells have a simple syntax for changing what files are open for the standard I/O streams of a process. Changing a standard file is called ***I/O redirection***. The syntax for I/O redirection is shown in Figure C.5. In this

command	meaning of command
% *ls* > *filea*	direct output of *ls* to file *filea*
% *pr* < *filea* > *fileb*	input from *filea* and output to *fileb*
% *lpr* < *fileb*	input from *fileb*
% % make program > & errs	save both standard output and standard error in a file

Figure C.5 Standard /io/ redirection.

example, the `ls` command produces a listing of the names of files in the current directory, the `pr` command formats that list into pages suitable for a printer, and the `lpr` command spools the formatted output to a printer, such as */dev/lp0*. The subsequent command forces all output and all error messages to be redirected to a file. Without the ampersand, error messages appear on the terminal.

C.4.3 Pipelines, Filters, and Shell Scripts

The first three commands of Figure C.5 could have been coalesced into the one command

% ***ls | pr | lpr***

Each vertical bar tells the shell to arrange for the output of the preceding command to be passed as input to the following command. A **pipe** is used to carry the data from one process to the other. One process writes into one end of the pipe, and another process reads from the other end. In the example, the write end of one pipe would be set up by the shell to be the standard output of `ls`, and the read end of the pipe would be the standard input of `pr`. Another pipe would be between `pr` and `lpr`.

A command such as `pr` that passes its standard input to its standard output, performing some processing on it, is called a ***filter***. Many UNIX commands can be used as filters. Complicated functions can be pieced together as pipelines of common commands. Also, common functions, such as output formatting, do not need to be built into numerous commands, because the output of almost any program can be piped through `pr` (or some other appropriate filter).

Both of the common UNIX shells are also programming languages, with shell variables and the usual higher-level programming-language control constructs (loops, conditionals). The execution of a command is analogous to a subroutine call. A file of shell commands, a ***shell script***, can be executed like any other command, with the appropriate shell being invoked automatically to read it. ***Shell programming*** thus can be used to combine ordinary programs conveniently for sophisticated applications without the need for any programming in conventional languages.

This external user view is commonly thought of as the definition of UNIX, yet it is the most easily changed definition. Writing a new shell with a quite different syntax and semantics would greatly change the user view while not changing the kernel or even the programmer interface. Several menu-driven and iconic interfaces for UNIX exist, and the X Window System is rapidly becoming a standard. The heart of UNIX is, of course, the kernel. This kernel is much more difficult to change than is the user interface, because all programs depend on the system calls that it provides to remain consistent. Of course, new system calls can be added to increase functionality, but programs must then be modified to use the new calls.

C.5 Process Management

A major design problem for operating systems is the representation of processes. One substantial difference between UNIX and many other systems is the ease with which multiple processes can be created and manipulated. These

processes are represented in UNIX by various control blocks. No system control blocks are accessible in the virtual address space of a user process; control blocks associated with a process are stored in the kernel. The kernel uses the information in these control blocks for process control and CPU scheduling.

C.5.1 Process Control Blocks

The most basic data structure associated with processes is the ***process structure***. A process structure contains everything that the system needs to know about a process when the process is swapped out, such as its unique process identifier, scheduling information (for example, the priority of the process), and pointers to other control blocks. There is an array of process structures whose length is defined at system-linking time. The process structures of ready processes are kept linked together by the scheduler in a doubly linked list (the ready queue), and there are pointers from each process structure to the process's parent, to its youngest living child, and to various other relatives of interest, such as a list of processes sharing the same program code (text).

The ***virtual address space*** of a user process is divided into text (program code), data, and stack segments. The data and stack segments are always in the same address space, but they may grow separately, and usually in opposite directions. Most frequently, the stack grows down as the data grow up toward it. The text segment is sometimes (as on an Intel 8086 with separate instruction and data space) in an address space different from the data and stack, and it is usually read-only. The debugger puts a text segment in read–write mode to allow insertion of breakpoints.

Every process with sharable text (almost all, under FreeBSD) has a pointer from its process structure to a ***text structure***. The text structure records how many processes are using the text segment, including a pointer into a list of their process structures, and where the page table for the text segment can be found on disk when it is swapped. The text structure itself is always resident in main memory. An array of such structures is allocated at system link time. The text, data, and stack segments for the processes may be swapped. When the segments are swapped in, they are paged.

The ***page tables*** record information on the mapping from the process's virtual memory to physical memory. The process structure contains pointers to the page table, for use when the process is resident in main memory, or the address of the process on the swap device, when the process is swapped. There is no special separate page table for a shared text segment; every process sharing the text segment has entries for its pages in the process's page table.

Information about the process needed only when the process is resident (that is, not swapped out) is kept in the ***user structure*** (or ***u structure***), rather than in the process structure. This structure is mapped read-only into user virtual address space, so user processes can read its contents. It is writable by the kernel. The user structure contains a copy of the process control block, or PCB, which is kept here for saving the process's general registers, stack pointer, program counter, and page-table base registers when the process is not running. There is space to keep system-call parameters and return values. All user and group identifiers associated with the process (not just the effective user identifier kept in the process structure) are kept here. Signals, timers, and quotas have data structures here. Of more obvious relevance to the ordinary

user, the current directory and the table of open files are maintained in the user structure.

Every process has both a user and a system mode. Most ordinary work is done in ***user mode***, but when a system call is made, it is performed in ***system mode***. The system and user phases of a process never execute simultaneously. When a process is executing in system mode, a ***kernel stack*** for that process is used, rather than the user stack belonging to that process. The kernel stack for the process immediately follows the user structure. The kernel stack and the user structure together compose the ***system data segment*** for the process. The kernel has its own stack for use when it is not doing work on behalf of a process (for instance, for interrupt handling).

Figure C.6 illustrates how the process structure is used to find the various parts of a process.

The `fork()` system call allocates a new process structure (with a new process identifier) for the child process and copies the user structure. There is ordinarily no need for a new text structure, as the processes share their text. The appropriate counters and lists are merely updated. A new page table is constructed, and new main memory is allocated for the data and stack segments of the child process. The copying of the user structure preserves open file descriptors, user and group identifiers, signal handling, and most similar properties of a process.

The `vfork()` system call does *not* copy the data and stack to the new process; rather, the new process simply shares the page table of the old one. A new user structure and a new process structure are still created. A common use of this system call occurs when a shell executes a command and waits for its completion. The parent process uses `vfork()` to produce the child process. Because the child process wishes to use an `execve()` immediately to change its virtual address space completely, there is no need for a complete copy of the

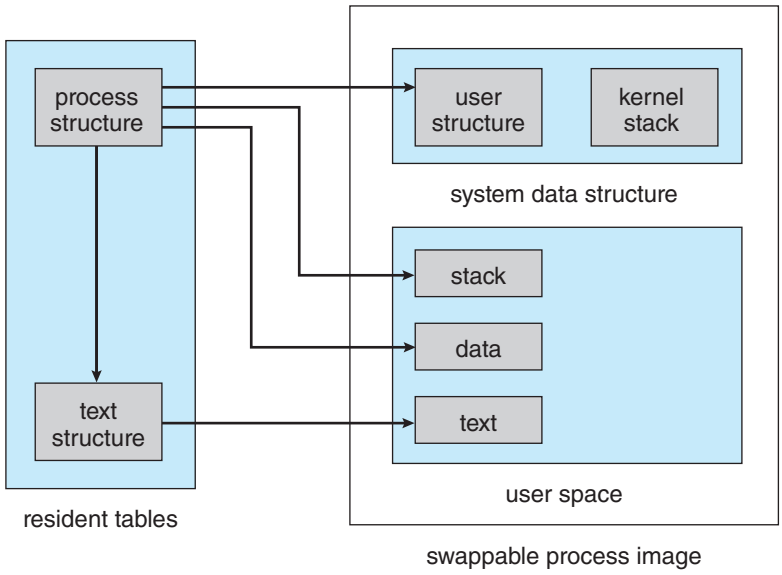


Figure C.6 Finding parts of a process using the process structure.

parent process. Such data structures as are necessary for manipulating pipes may be kept in registers between the `vfork()` and the `execve()`. Files may be closed in one process without affecting the other process, since the kernel data structures involved depend on the user structure, which is not shared. The parent is suspended when it calls `vfork()` until the child either calls `execve()` or terminates, so that the parent will not change memory that the child needs.

When the parent process is large, `vfork()` can produce substantial savings in system CPU time. However, it is a fairly dangerous system call, since any memory change occurs in both processes until the `execve()` occurs. An alternative is to share all pages by duplicating the page table but to mark the entries of both page tables as ***copy-on-write***. The hardware protection bits are set to trap any attempt to write in these shared pages. If such a trap occurs, a new frame is allocated, and the shared page is copied to the new frame. The page tables are adjusted to show that this page is no longer shared (and therefore no longer needs to be write-protected), and execution can resume.

An `execve()` system call creates no new process or user structure. Rather, the text and data of the process are replaced. Open files are preserved (although there is a way to specify that certain file descriptors are to be closed on an `execve()`). Most signal-handling properties are preserved, but arrangements to call a specific user routine on a signal are canceled, for obvious reasons. The process identifier and most other properties of the process are unchanged.

C.5.2 CPU Scheduling

CPU scheduling in UNIX is designed to benefit interactive processes. Processes are given small CPU time slices by a priority algorithm that reduces to round-robin scheduling for CPU-bound jobs.

Every process has a ***scheduling priority*** associated with it; larger numbers indicate lower priority. Processes doing disk I/O or other important tasks have priorities less than "pzero" and cannot be killed by signals. Ordinary user processes have positive priorities and thus are less likely to be run than is any system process, although user processes can set precedence over one another through the `nice` command.

The more CPU time a process accumulates, the lower (more positive) its priority becomes, and vice versa. This negative feedback in CPU scheduling makes it difficult for a single process to take all the CPU time. Process aging is employed to prevent starvation.

Older UNIX systems used a 1-second quantum for the round-robin scheduling. FreeBSD reschedules processes every 0.1 second and recomputes priorities every second. The round-robin scheduling is accomplished by the ***timeout*** mechanism, which tells the clock interrupt driver to call a kernel subroutine after a specified interval. The subroutine to be called in this case causes the rescheduling and then resubmits a timeout to call itself again. The priority recomputation is also timed by a subroutine that resubmits a timeout for itself.

There is no preemption of one process by another in the kernel. A process may relinquish the CPU because it is waiting for I/O or because its time slice has expired. When a process chooses to relinquish the CPU, it goes to ***sleep*** on an ***event***. The kernel primitive used for this purpose is called `sleep()` (not to be confused with the user-level library routine of the same name). `Sleep()` takes an argument that is, by convention, the address of a kernel data

structure related to an event for which a process is waiting. When the event occurs, the system process that knows about it calls `wakeup()` with the address corresponding to the event, and *all* processes that had done a sleep on the same address are put in the ready queue to be run.

For example, a process waiting for disk I/O to complete will sleep on the address of the buffer header corresponding to the data being transferred. When the interrupt routine for the disk driver notes that the transfer is complete, it calls `wakeup()` on the buffer header. The interrupt uses the kernel stack for whatever process happened to be running at the time, and the `wakeup()` is done from that system process.

The process that actually does run is chosen by the scheduler. `Sleep()` takes a second argument, which is the scheduling priority to be used for this purpose. This priority argument, if less than "pzero," also prevents the process from being awakened prematurely by some exceptional event, such as a signal.

When a signal is generated, it is left pending until the system half of the affected process next runs. This event usually happens soon, since the signal normally causes the process to be awakened if the process has been waiting for some other condition.

No memory is associated with events. The caller of the routine that does a `sleep()` on an event must be prepared to deal with a premature return, including the possibility that the reason for waiting has vanished.

Race conditions are involved in the event mechanism. If a process decides (because of checking a flag in memory, for instance) to sleep on an event, and the event occurs before the process can execute the primitive that does the `sleep()` on the event, the process sleeping may then sleep forever. We prevent this situation by raising the hardware processor priority during the critical section so that no interrupts can occur. Thus, only the process desiring the event can run until it is sleeping. Hardware processor priority is used in this manner to protect critical regions throughout the kernel and is the greatest obstacle to porting UNIX to multiple-processor machines. However, this problem has not prevented such porting from being done repeatedly.

Many processes, such as text editors, are I/O bound and are, in general, scheduled mainly on the basis of waiting for I/O. Experience suggests that the UNIX scheduler performs best with I/O-bound jobs, as can be observed when several CPU-bound jobs, such as text formatters or language interpreters, are running.

What has been referred to here as ***CPU scheduling*** corresponds closely to the ***short-term scheduling*** of Chapter 3. However, the negative-feedback property of the priority scheme provides some long-term scheduling in that it largely determines the long-term ***job mix***. Medium-term scheduling is done by the swapping mechanism described in Section C.6.

C.6 Memory Management

Much of UNIX's early development was done on a PDP-11. The PDP-11 has only eight segments in its virtual address space, and the size of each is at most 8,192 bytes. The larger machines, such as the PDP-11/70, allow separate instruction and address spaces, effectively doubling the address space and number of segments, but this address space is still relatively small. In addition, the kernel

was even more severely constrained due to dedication of one data segment to interrupt vectors, another to point at the per-process system data segment, and yet another for the UNIBUS (system I/O bus) registers. Further, on the smaller PDP-11s, total physical memory was limited to 256 KB. The total memory resources were insufficient to justify or support complex memory-management algorithms. Thus, UNIX swapped entire process memory images.

Berkeley introduced paging to UNIX with 3BSD. VAX 4.2BSD is a demand-paged virtual memory system. Paging eliminates external fragmentation of memory. (Internal fragmentation still occurs, but it is negligible with a reasonably small page size.) Because paging allows execution with only parts of each process in memory, more jobs can be kept in main memory, and swapping can be kept to a minimum. ***Demand paging*** is done in a straightforward manner. When a process needs a page and the page is not there, a page fault to the kernel occurs, a frame of main memory is allocated, and the proper disk page is read into the frame.

There are a few optimizations. If the page needed is still in the page table for the process but has been marked invalid by the page-replacement process, it can be marked valid and used without any I/O transfer. Pages can similarly be retrieved from the list of free frames. When most processes are started, many of their pages are prepaged and are put on the free list for recovery by this mechanism. Arrangements can also be made for a process to have no prepaging on startup. That is seldom done, however, because it results in more page-fault overhead, being closer to pure demand paging. FreeBSD implements page coloring with paging queues. The queues are arranged according to the size of the processor's L1 and L2 caches. When a new page needs to be allocated, FreeBSD tries to get one that is optimally aligned for the cache. If the page has to be fetched from disk, it must be locked in memory for the duration of the transfer. This locking ensures that the page will not be selected for page replacement. Once the page is fetched and mapped properly, it must remain locked if raw physical I/O is being done on it.

The page-replacement algorithm is more interesting. 4.2BSD uses a modification of the ***second-chance (clock) algorithm*** described in Section 10.4.5. The map of all nonkernel main memory (the ***core map*** or ***cmap***) is swept linearly and repeatedly by a software ***clock hand***. When the clock hand reaches a given frame, if the frame is marked as being in use by some software condition (for example, if physical I/O is in progress using it) or if the frame is already free, the frame is left untouched, and the clock hand sweeps to the next frame. Otherwise, the corresponding text or process page-table entry for this frame is located. If the entry is already invalid, the frame is added to the free list. Otherwise, the page-table entry is made invalid but reclaimable (that is, if it has not been paged out by the next time it is wanted, it can just be made valid again).

BSD Tahoe added support for systems that implement the reference bit. On such systems, one pass of the clock hand turns the reference bit off, and a second pass places those pages whose reference bits remain off onto the free list for replacement. Of course, if the page is dirty, it must first be written to disk before being added to the free list. Pageouts are done in clusters to improve performance.

There are checks to make sure that the number of valid data pages for a process does not fall too low and to keep the paging device from being flooded

with requests. There is also a mechanism by which a process can limit the amount of main memory it uses.

The LRU clock-hand scheme is implemented in the `pagedaemon`, which is process 2. (Remember that the `swapper` is process 0 and `init` is process 1.) This process spends most of its time sleeping, but a check is done several times per second (scheduled by a timeout) to see if action is necessary. If it is, process 2 is awakened. Whenever the number of free frames falls below a threshold, `lotsfree`, the `pagedaemon` is awakened. Thus, if there is always a large amount of free memory, the `pagedaemon` imposes no load on the system, because it never runs.

The sweep of the clock hand each time the `pagedaemon` process is awakened (that is, the number of frames scanned, which is usually more than the number paged out) is determined both by the number of frames lacking to reach `lotsfree` and by the number of frames that the scheduler has determined are needed for various reasons (the more frames needed, the longer the sweep). If the number of frames free rises to `lotsfree` before the expected sweep is completed, the hand stops, and the `pagedaemon` process sleeps. The parameters that control the range of the clock-hand sweep are determined at system startup according to the amount of main memory, such that `pagedaemon` does not use more than 10 percent of all CPU time.

If the scheduler decides that the paging system is overloaded, processes will be swapped out whole until the overload is relieved. This swapping usually happens only if several conditions are met: load average is high; free memory has fallen below a low limit, `minfree`; and the average memory available over recent time is less than a desirable amount, `desfree`, where `lotsfree` > `desfree` > `minfree`. In other words, only a chronic shortage of memory with several processes trying to run will cause swapping, and even then free memory has to be extremely low at the moment. (An excessive paging rate or a need for memory by the kernel itself may also enter into the calculations, in rare cases.) Processes may be swapped by the scheduler, of course, for other reasons (such as simply because they have not run for a long time).

The parameter `lotsfree` is usually one-quarter of the memory in the map that the clock hand sweeps, and `desfree` and `minfree` are usually the same across different systems but are limited to fractions of available memory. FreeBSD dynamically adjusts its paging queues so these virtual memory parameters will rarely need to be adjusted. `Minfree` pages must be kept free in order to supply any pages that might be needed at interrupt time.

Every process's text segment is, by default, shared and read-only. This scheme is practical with paging, because there is no external fragmentation, and the swap space gained by sharing more than offsets the negligible amount of overhead involved, as the kernel virtual space is large.

CPU scheduling, memory swapping, and paging interact. The lower the priority of a process, the more likely that its pages will be paged out and the more likely that it will be swapped in its entirety. The age preferences in choosing processes to swap guard against thrashing, but paging does so more effectively. Ideally, processes will not be swapped out unless they are idle, because each process will need only a small working set of pages in main memory at any one time, and the `pagedaemon` will reclaim unused pages for use by other processes.

The amount of memory the process will need is some fraction of that process's total virtual size—up to one-half if that process has been swapped out for a long time.

C.7 File System

The UNIX file system supports two main objects: files and directories. Directories are just files with a special format, so the representation of a file is the basic UNIX concept.

C.7.1 Blocks and Fragments

Most of the file system is taken up by ***data blocks***, which contain whatever the users have put in their files. Let's consider how these data blocks are stored on the disk.

The hardware disk sector is usually 512 bytes. A block size larger than 512 bytes is desirable for speed. However, because UNIX file systems usually contain a very large number of small files, much larger blocks would cause excessive internal fragmentation. That is why the earlier 4.1BSD file system was limited to a 1,024-byte (1-KB) block. The 4.2BSD solution is to use *two* block sizes for files that have no indirect blocks. All the blocks of a file are of a large size (such as 8 KB) except the last. The last block is an appropriate multiple of a smaller fragment size (for example, 1,024 KB) to fill out the file. Thus, a file of size 18,000 bytes would have two 8-KB blocks and one 2-KB fragment (which would not be filled completely).

The block and fragment sizes are set during file-system creation according to the intended use of the file system. If many small files are expected, the fragment size should be small; if repeated transfers of large files are expected, the basic block size should be large. Implementation details force a maximum block-to-fragment ratio of 8:1 and a minimum block size of 4 KB, so typical choices are 4,096:512 for the former case and 8,192:1,024 for the latter.

Suppose data are written to a file in transfer sizes of 1-KB bytes, and the block and fragment sizes of the file system are 4 KB and 512 bytes. The file system will allocate a 1-KB fragment to contain the data from the first transfer. The next transfer will cause a new 2-KB fragment to be allocated. The data from the original fragment must be copied into this new fragment, followed by the second 1-KB transfer. The allocation routines attempt to find the required space on the disk immediately following the existing fragment so that no copying is necessary. If they cannot do so, up to seven copies may be required before the fragment becomes a block. Provisions have been made for programs to discover the block size for a file so that transfers of that size can be made, to avoid fragment recopying.

C.7.2 Inodes

A file is represented by an ***inode***, which is a record that stores most of the information about a specific file on the disk. (See Figure C.7.) The name *inode* (pronounced *EYE node*) is derived from "index node" and was originally spelled "i-node"; the hyphen fell out of use over the years. The term is sometimes spelled "I node."

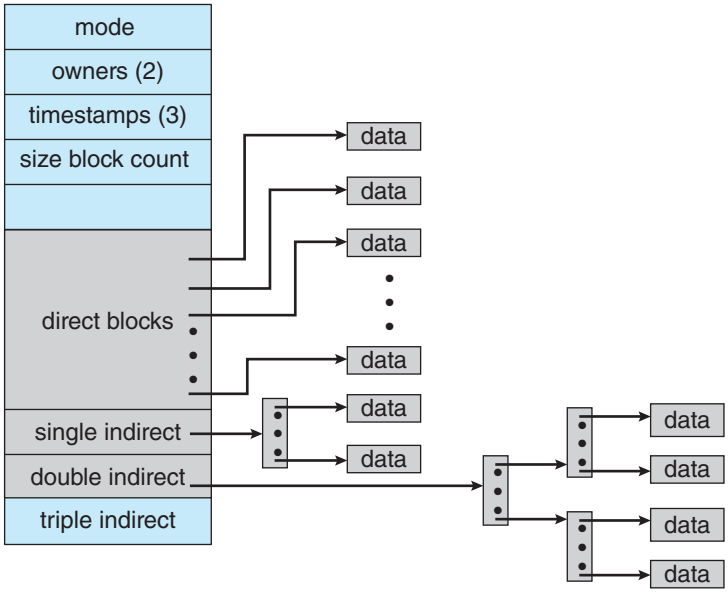


Figure C.7 The UNIX inode.

The inode contains the user and group identifiers of the file, the times of the last file modification and access, a count of the number of hard links (directory entries) to the file, and the type of the file (plain file, directory, symbolic link, character device, block device, or socket). In addition, the inode contains 15 pointers to the disk blocks containing the data contents of the file. The first 12 of these pointers point to ***direct blocks***. That is, they contain addresses of blocks that contain data of the file. Thus, the data for small files (no more than 12 blocks) can be referenced immediately, because a copy of the inode is kept in main memory while a file is open. If the block size is 4 KB, then up to 48 KB of data can be accessed directly from the inode.

The next three pointers in the inode point to ***indirect blocks***. If the file is large enough to use indirect blocks, each of the indirect blocks is of the major block size; the fragment size applies only to data blocks. The first indirect block pointer is the address of a ***single indirect block***. The single indirect block is an index block containing not data but the addresses of blocks that do contain data. Then, there is a ***double-indirect-block pointer***, the address of a block that contains the addresses of blocks that contain pointers to the actual data blocks. The last pointer would contain the address of a ***triple indirect block***; however, there is no need for it.

The minimum block size for a file system in 4.2BSD is 4 KB, so files with as many as 2^{32} bytes will use only double, not triple, indirection. That is, since each block pointer takes 4 bytes, we have 49,152 bytes accessible in direct blocks, 4,194,304 bytes accessible by a single indirection, and 4,294,967,296 bytes reachable through double indirection, for a total of 4,299,210,752 bytes, which is larger than 2^{32} bytes. The number 2^{32} is significant because the file offset in the file structure in main memory is kept in a 32-bit word. Files therefore cannot be larger than 2^{32} bytes. Since file pointers are signed integers

(for seeking backward and forward in a file), the actual maximum file size is 2^{32-1} bytes. Two gigabytes is large enough for most purposes.

C.7.3 Directories

Plain files are not distinguished from directories at this level of implementation. Directory contents are kept in data blocks, and directories are represented by an inode in the same way as plain files. Only the inode type field distinguishes between plain files and directories. Plain files are not assumed to have a structure, whereas directories have a specific structure. In Version 7, file names were limited to 14 characters, so directories were a list of 16-byte entries: 2 bytes for an inode number and 14 bytes for a file name.

In FreeBSD file names are of variable length, up to 255 bytes, so directory entries are also of variable length. Each entry contains first the length of the entry, then the file name and the inode number. This variable-length entry makes the directory management and search routines more complex, but it allows users to choose much more meaningful names for their files and directories. The first two names in every directory are "." and "..". New directory entries are added to the directory in the first space available, generally after the existing files. A linear search is used.

The user refers to a file by a path name, whereas the file system uses the inode as its definition of a file. Thus, the kernel has to map the supplied user path name to an inode. The directories are used for this mapping.

First, a starting directory is determined. As mentioned earlier, if the first character of the path name is "/," the starting directory is the root directory. If the path name starts with any character other than a slash, the starting directory is the current directory of the current process. The starting directory is checked for proper file type and access permissions, and an error is returned if necessary. The inode of the starting directory is always available.

The next element of the path name, up to the next "/" or to the end of the path name, is a file name. The starting directory is searched for this name, and an error is returned if the name is not found. If the path name has yet another element, the current inode must refer to a directory; an error is returned if it does not or if access is denied. This directory is searched in the same way as the previous one. This process continues until the end of the path name is reached and the desired inode is returned. This step-by-step process is needed because at any directory a mount point (or symbolic link, as discussed below) may be encountered, causing the translation to move to a different directory structure for continuation.

Hard links are simply directory entries like any other. We handle symbolic links for the most part by starting the search over with the path name taken from the contents of the symbolic link. We prevent infinite loops by counting the number of symbolic links encountered during a path-name search and returning an error when a limit (eight) is exceeded.

Nondisk files (such as devices) do not have data blocks allocated on the disk. The kernel notices these file types (as indicated in the inode) and calls appropriate drivers to handle I/O for them.

Once the inode is found by, for instance, the `open()` system call, a ***file structure*** is allocated to point to the inode. The file descriptor given to the user refers to this file structure. FreeBSD has a ***directory name cache*** to hold

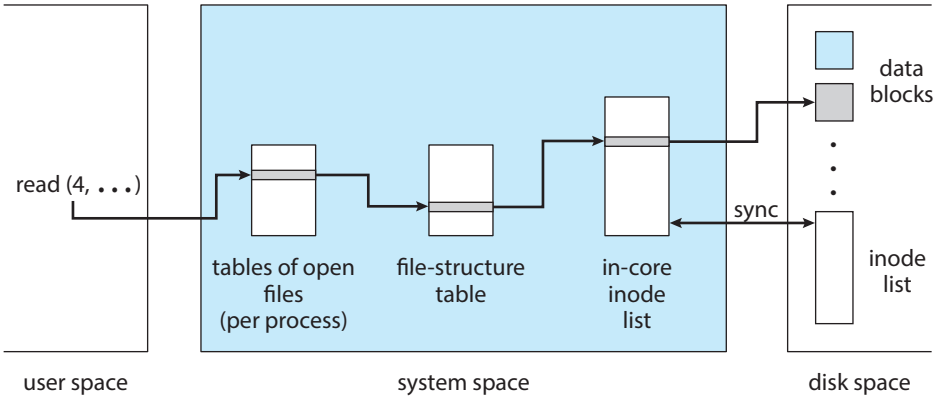


Figure C.8 File-system control blocks.

recent directory-to-inode translations, which greatly increases file-system performance.

C.7.4 Mapping a File Descriptor to an Inode

A system call that refers to an open file indicates the file by passing a file descriptor as an argument. The file descriptor is used by the kernel to index a table of open files for the current process. Each entry in the table contains a pointer to a file structure. This file structure in turn points to the inode; see Figure C.8. The open file table has a fixed length, which is settable only at boot time. Therefore, there is a fixed limit on the number of concurrently open files in a system.

The `read()` and `write()` system calls do not take a position in the file as an argument. Rather, the kernel keeps a ***file offset***, which is updated by an appropriate amount after each `read()` or `write()` according to the number of data actually transferred. The offset can be set directly by the `lseek()` system call. If the file descriptor indexed an array of inode pointers instead of file pointers, this offset would have to be kept in the inode. Because more than one process may open the same file, and each such process needs its own offset for the file, keeping the offset in the inode is inappropriate. Thus, the file structure is used to contain the offset. File structures are inherited by the child process after a `fork()`, so several processes may share the same offset location for a file.

The ***inode structure*** pointed to by the file structure is an in-core copy of the inode on the disk. The in-core inode has a few extra fields, such as a reference count of how many file structures are pointing at it, and the file structure has a similar reference count for how many file descriptors refer to it. When a count becomes zero, the entry is no longer needed and may be reclaimed and reused.

C.7.5 Disk Structures

The file system that the user sees is supported by data on a mass storage device —usually, a disk. The user ordinarily knows of only one file system, but this one logical file system may actually consist of several *physical* file systems, each on a different device. Because device characteristics differ, each separate

hardware device defines its own physical file system. In fact, we generally want to partition large physical devices, such as disks, into multiple *logical* devices. Each logical device defines a physical file system. Figure C.9 illustrates how a directory structure is partitioned into file systems, which are mapped onto logical devices, which are partitions of physical devices. The sizes and locations of these partitions were coded into device drivers in earlier systems, but they are maintained on the disk by FreeBSD.

Partitioning a physical device into multiple file systems has several benefits. Different file systems can support different uses. Although most partitions will be used by the file system, at least one will be needed as a swap area for the virtual memory software. Reliability is improved, because software damage is generally limited to only one file system. We can improve efficiency by varying the file-system parameters (such as the block and fragment sizes) for each partition. Also, having separate file systems prevents one program from using all available space for a large file, because files cannot be split across file systems. Finally, disk backups are done per partition, and it is faster to search a backup tape for a file if the partition is smaller. Restoring the full partition from tape is also faster.

The number of file systems on a drive varies according to the size of the disk and the purpose of the computer system as a whole. One file system, the

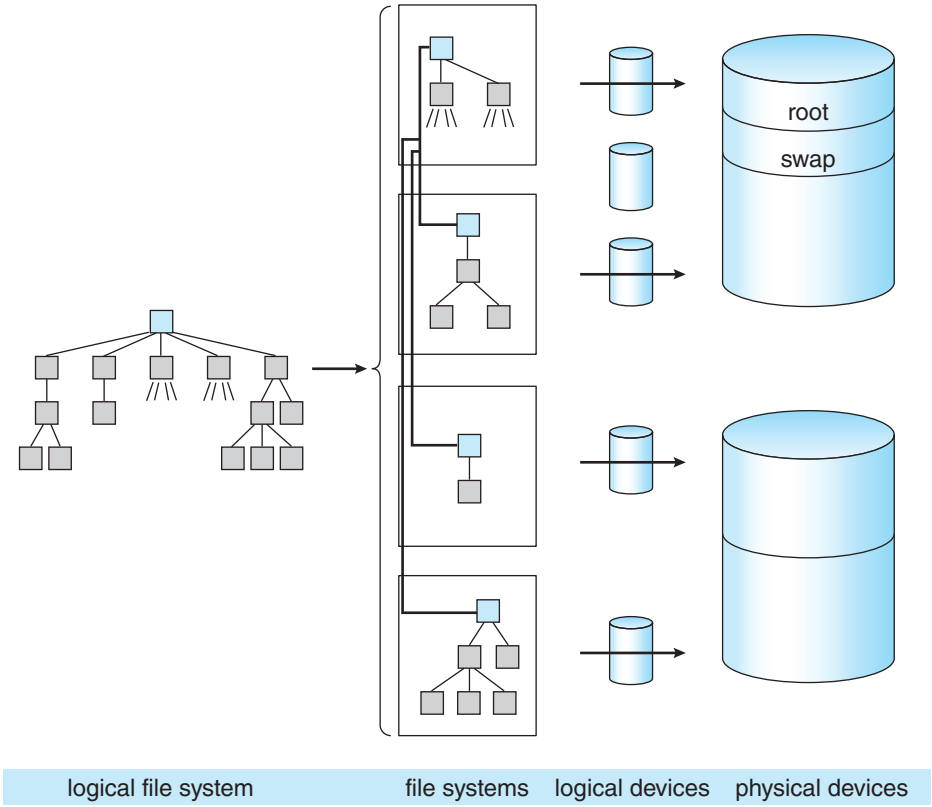


Figure C.9 Mapping of a logical file system to physical devices.

root file system, is always available. Other file systems may be ***mounted***—that is, integrated into the directory hierarchy of the root file system.

A bit in the inode structure indicates that the inode has a file system mounted on it. A reference to this file causes the ***mount table*** to be searched to find the device number of the mounted device. The device number is used to find the inode of the root directory of the mounted file system, and that inode is used. Conversely, if a path-name element is ".." and the directory being searched is the root directory of a file system that is mounted, the mount table is searched to find the inode it is mounted on, and that inode is used.

Each file system is a separate system resource and represents a set of files. The first sector on the logical device is the ***boot block***, possibly containing a primary bootstrap program, which may be used to call a secondary bootstrap program residing in the next 7.5 KB. A system needs only one partition containing boot-block data, but the system manager may install duplicates via privileged programs, to allow booting when the primary copy is damaged. The ***superblock*** contains static parameters of the file system. These parameters include the total size of the file system, the block and fragment sizes of the data blocks, and assorted parameters that affect allocation policies.

C.7.6 Implementations

The user interface to the file system is simple and well defined, allowing the implementation of the file system itself to be changed without significant effect on the user. The file system was changed between Version 6 and Version 7 of 3BSD, and again between Version 7 and 4BSD. For Version 7, the size of inodes doubled, the maximum file and file-system sizes increased, and the details of free-list handling and superblock information changed. At that time also, `seek()` (with a 16-bit offset) became `lseek()` (with a 32-bit offset), to allow specification of offsets in larger files, but few other changes were visible outside the kernel.

In 4.0BSD, the size of blocks used in the file system was increased from 512 bytes to 1,024 bytes. Although this increased size produced increased internal fragmentation on the disk, it doubled throughput, due mainly to the greater number of data accessed on each disk transfer. This idea was later adopted by System V, along with a number of other ideas, device drivers, and programs.

4.2BSD added the Berkeley Fast File System, which increased speed and was accompanied by new features. Symbolic links required new system calls. Long file names necessitated new directory system calls to traverse the now-complex internal directory structure. Finally, `truncate()` calls were added. The Fast File System was a success and is now found in most implementations of UNIX. Its performance is made possible by its layout and allocation policies, which we discuss next. In Section 14.4.4, we discussed changes made in SunOS to increase disk throughput further.

C.7.7 Layout and Allocation Policies

The kernel uses a <***logical device number, inode number***> pair to identify a file. The logical device number defines the file system involved. The inodes in the file system are numbered in sequence. In the Version 7 file system, all inodes are in an array immediately following a single superblock at the beginning of

the logical device, with the data blocks following the inodes. The inode number is effectively just an index into this array.

With the Version 7 file system, a block of a file can be anywhere on the disk between the end of the inode array and the end of the file system. Free blocks are kept in a linked list in the superblock. Blocks are pushed onto the front of the free list and are removed from the front as needed to serve new files or to extend existing files. Thus, the blocks of a file may be arbitrarily far from both the inode and one another. Furthermore, the more a file system of this kind is used, the more disorganized the blocks in a file become. We can reverse this process only by reinitializing and restoring the entire file system, which is not a convenient task to perform. This process was described in Section 14.7.4.

Another difficulty is that the reliability of the file system is suspect. For speed, the superblock of each mounted file system is kept in memory. Keeping the superblock in memory allows the kernel to access a superblock quickly, especially for using the free list. Every 30 seconds, the superblock is written to the disk, to keep the in-core and disk copies synchronized (by the update program, using the `sync()` system call). However, system bugs or hardware failures may cause a system crash, which destroys the in-core superblock between updates to the disk. Then, the free list on disk does not accurately reflect the state of the disk. To reconstruct it, we must perform a lengthy examination of all blocks in the file system. (This problem remains in the new file system.)

The 4.2BSD file-system implementation is radically different from that of Version 7. This reimplementation was done primarily to improve efficiency and robustness, and most such changes are invisible outside the kernel. Other changes introduced at the same time are visible at both the system-call and the user levels; examples include symbolic links and long file names (up to 255 characters). Most of the changes required for these features were not in the kernel, however, but in the programs that use them.

Space allocation is especially different. The major new concept in FreeBSD is the ***cylinder group***. The cylinder group was introduced to allow localization of the blocks in a file. Each cylinder group occupies one or more consecutive cylinders of the disk, so that disk accesses within the cylinder group require minimal disk head movement. Every cylinder group has a superblock, a cylinder block, an array of inodes, and some data blocks (Figure C.10).

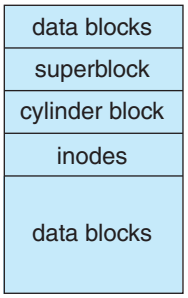


Figure C.10 4.3 BSD cylinder group.

The superblocks in all cylinder groups are identical, so that a superblock can be recovered from any one of them in the event of disk corruption. The ***cylinder block*** contains dynamic parameters of the particular cylinder group. These include a bit map of free data blocks and fragments and a bitmap of free inodes. Statistics on recent progress of the allocation strategies are also kept here.

The header information in a cylinder group (the superblock, the cylinder block, and the inodes) is not always at the beginning of the group. If it were, the header information for every cylinder group might be on the same disk platter, and a single disk head crash could wipe out all of them. Therefore, each cylinder group has its header information at a different offset from the beginning of the group.

The directory-listing command `ls` commonly reads all the inodes of every file in a directory, making it desirable for all such inodes to be close together on the disk. For this reason, the inode for a file is usually allocated from the cylinder group containing the inode of the file's parent directory. Not everything can be localized, however, so an inode for a new directory is put in a *different* cylinder group from that of its parent directory. The cylinder group chosen for such a new directory inode is that with the greatest number of unused inodes.

To reduce disk head seeks involved in accessing the data blocks of a file, we allocate blocks from the same cylinder group as often as possible. Because a single file cannot be allowed to take up all the blocks in a cylinder group, a file exceeding a certain size (such as 2 MB) has further block allocation redirected to a different cylinder group; the new group is chosen from among those having more than average free space. If the file continues to grow, allocation is again redirected (at each megabyte) to yet another cylinder group. Thus, all the blocks of a small file are likely to be in the same cylinder group, and the number of long head seeks involved in accessing a large file is kept small.

There are two levels of disk-block-allocation routines. The global policy routines select a desired disk block according to the considerations already discussed. The local policy routines use the specific information recorded in the cylinder blocks to choose a block near the one requested. If the requested block is not in use, it is returned. Otherwise, the routine returns either the block rotationally closest to the one requested in the same cylinder or a block in a different cylinder but in the same cylinder group. If no more blocks are in the cylinder group, a quadratic rehash is done among all the other cylinder groups to find a block. If that fails, an exhaustive search is done. If enough free space (typically 10 percent) is left in the file system, blocks are usually found where desired, the quadratic rehash and exhaustive search are not used, and performance of the file system does not degrade with use.

Because of the increased efficiency of the Fast File System, typical disks are now utilized at 30 percent of their raw transfer capacity. This percentage is a marked improvement over that realized with the Version 7 file system, which used about 3 percent of the bandwidth.

BSD Tahoe introduced the Fat Fast File System, which allows the number of inodes per cylinder group, the number of cylinders per cylinder group, and the number of distinguished rotational positions to be set when the file system is created. FreeBSD previously set these parameters according to the disk hardware type.

C.8 I/O System

One of the purposes of an operating system is to hide the peculiarities of specific hardware devices from the user. For example, the file system presents a simple, consistent storage facility (the file) independent of the underlying disk hardware. In UNIX, the peculiarities of I/O devices are also hidden from the bulk of the kernel itself by the ***I/O system***. The I/O system consists of a buffer caching system, general device-driver code, and drivers for specific hardware devices. Only the device driver knows the peculiarities of a specific device. The major parts of the I/O system are diagrammed in Figure C.11.

There are three main kinds of I/O in FreeBSD: block devices, character devices, and the socket interface. The socket interface, together with its protocols and network interfaces, will be described in Section C.9.1.

Block devices include disks and tapes. Their distinguishing characteristic is that they are directly addressable in a fixed block size—usually 512 bytes. A block-device driver is required to isolate details of tracks, cylinders, and so on from the rest of the kernel. Block devices are accessible directly through appropriate device special files (such as */dev/rp0*), but they are more commonly accessed indirectly through the file system. In either case, transfers are buffered through the ***block buffer cache***, which has a profound effect on efficiency.

Character devices include terminals and line printers but also include almost everything else (except network interfaces) that does not use the block buffer cache. For instance, */dev/mem* is an interface to physical main memory, and */dev/null* is a bottomless sink for data and an endless source of end-of-file markers. Some devices, such as high-speed graphics interfaces, may have their own buffers or may always do I/O directly into the user's data space; because they do not use the block buffer cache, they are classed as character devices. Terminals and terminal-like devices use ***C-lists***, which are buffers smaller than those of the block buffer cache.

Block devices and character devices are the two main device classes. Device drivers are accessed by one of two arrays of entry points. One array is for block devices; the other is for character devices. A device is distinguished by a class (block or character) and a ***device number***. The device number consists of two parts. The ***major device number*** is used to index the array for character or block devices to find entries into the appropriate device driver. The ***minor***

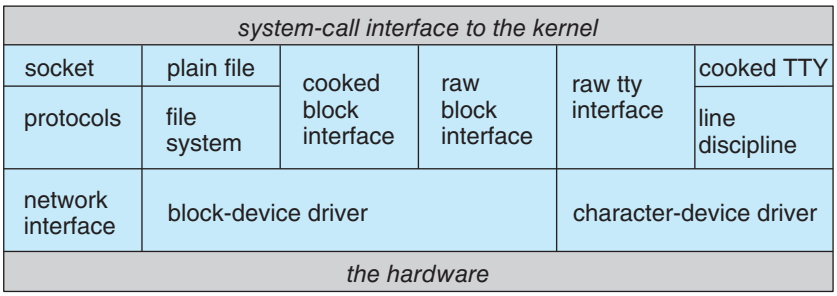


Figure C.11 4.3 BSD kernel I/O structure.

device number is interpreted by the device driver as, for example, a logical disk partition or a terminal line.

A device driver is connected to the rest of the kernel only by the entry points recorded in the array for its class and by its use of common buffering systems. This segregation is important for portability and for system configuration.

C.8.1 Block Buffer Cache

The block devices, as mentioned, use a block buffer cache. The buffer cache consists of a number of buffer headers, each of which can point to a piece of physical memory as well as to a device number and a block number on the device. The buffer headers for blocks not currently in use are kept in several linked lists, one for each of the following:

- Buffers recently used, linked in LRU order (the LRU list)
- Buffers not recently used or without valid contents (the AGE list)
- EMPTY buffers with no physical memory associated with them

The buffers in these lists are also hashed by device and block number for search efficiency.

When a block is wanted from a device (a read), the cache is searched. If the block is found, it is used, and no I/O transfer is necessary. If it is not found, a buffer is chosen from the AGE list or, if that list is empty, the LRU list. Then the device number and block number associated with it are updated, memory is found for it if necessary, and the new data are transferred into it from the device. If there are no empty buffers, the LRU buffer is written to its device (if it is modified), and the buffer is reused.

On a write, if the block in question is already in the buffer cache, the new data are put in the buffer (overwriting any previous data), the buffer header is marked to indicate that the buffer has been modified, and no I/O is immediately necessary. The data will be written when the buffer is needed for other data. If the block is not found in the buffer cache, an empty buffer is chosen (as with a read), and a transfer is done to this buffer. Writes are periodically forced for dirty buffer blocks to minimize potential file-system inconsistencies after a crash.

The number of data in a buffer in FreeBSD is variable, up to a maximum over all file systems, usually 8 KB. The minimum size is the paging-cluster size, usually 1,024 bytes. Buffers are page-cluster aligned, and any page cluster may be mapped into only one buffer at a time, just as any disk block may be mapped into only one buffer at a time. The EMPTY list holds buffer headers, which are used if a physical memory block of 8 KB is split to hold multiple, smaller blocks. Headers are needed for these blocks and are retrieved from EMPTY.

The number of data in a buffer may grow as a user process writes more data following those already in the buffer. When this increase occurs, a new buffer large enough to hold all the data is allocated, and the original data are copied into it, followed by the new data. If a buffer shrinks, a buffer is taken

off the empty queue, excess pages are put in it, and that buffer is released to be written to disk.

Some devices, such as magnetic tapes, require that blocks be written in a certain order. Facilities are therefore provided to force a synchronous write of buffers to these devices in the correct order. Directory blocks are also written synchronously, to forestall crash inconsistencies. Consider the chaos that could occur if many changes were made to a directory but the directory entries themselves were not updated!

The size of the buffer cache can have a profound effect on the performance of a system, because, if it is large enough, the percentage of cache hits can be high and the number of actual I/O transfers low. FreeBSD optimizes the buffer cache by continually adjusting the amount of memory used by programs and the disk cache.

Some interesting interactions occur among the buffer cache, the file system, and the disk drivers. When data are written to a disk file, they are buffered in the cache, and the disk driver sorts its output queue according to disk address. These two actions allow the disk driver to minimize disk head seeks and to write data at times optimized for disk rotation. Unless synchronous writes are required, a process writing to disk simply writes into the buffer cache, and the system asynchronously writes the data to disk when convenient. The user process sees very fast writes. When data are read from a disk file, the block I/O system does some read-ahead; however, writes are much nearer to asynchronous than are reads. Thus, output to the disk through the file system is often faster than is input for large transfers, counter to intuition.

C.8.2 Raw Device Interfaces

Almost every block device also has a character interface, and these are called ***raw device interfaces***. Such an interface differs from the ***block interface*** in that the block buffer cache is bypassed.

Each disk driver maintains a queue of pending transfers. Each record in the queue specifies whether it is a read or a write and gives a main memory address for the transfer (usually in 512-byte increments), a device address for the transfer (usually the address of a disk sector), and a transfer size (in sectors). It is simple to map the information from a block buffer to what is required for this queue.

It is almost as simple to map a piece of main memory corresponding to part of a user process's virtual address space. This mapping is what a raw disk interface, for instance, does. Unbuffered transfers directly to or from a user's virtual address space are thus allowed. The size of the transfer is limited by the physical devices, some of which require an even number of bytes.

The kernel accomplishes transfers for swapping and paging simply by putting the appropriate request on the queue for the appropriate device. No special swapping or paging device driver is needed.

The 4.2BSD file-system implementation was actually written and largely tested as a user process that used a raw disk interface, before the code was moved into the kernel. In an interesting about-face, the Mach operating system

has no file system per se. File systems can be implemented as user-level tasks (see Appendix D).

C.8.3 C-Lists

As mentioned, terminals and terminal-like devices use a character-buffering system that keeps small blocks of characters (usually 28 bytes) in linked lists called C-lists. Although all free character buffers are kept in a single free list, most device drivers that use them limit the number of characters that may be queued at one time for any given terminal line.

There are routines to enqueue and dequeue characters for such lists. A `write()` system call to a terminal enqueues characters on a list for the device. An initial transfer is started, and interrupts cause dequeuing of characters and further transfers.

Input is similarly interrupt driven. Terminal drivers typically support *two* input queues, however, and conversion from the first (raw queue) to the other (canonical queue) is triggered when the interrupt routine puts an end-of-line character on the raw queue. The process doing a read on the device is then awakened, and its system phase does the conversion. The characters thus put on the canonical queue are then available to be returned to the user process by the read.

The device driver can bypass the canonical queue and return characters directly from the raw queue. This mode of operation is known as ***raw mode***. Full-screen editors, as well as other programs that need to react to every keystroke, use this mode.

C.9 Interprocess Communication

Although many tasks can be accomplished in isolated processes, many others require interprocess communication. Isolated computing systems have long served for many applications, but networking is increasingly important. Furthermore, with the increasing use of personal workstations, resource sharing is becoming more common. Interprocess communication has not traditionally been one of UNIX's strong points.

C.9.1 Sockets

The pipe (discussed in Section C.4.3) is the IPC mechanism most characteristic of UNIX. A pipe permits a reliable unidirectional byte stream between two processes. It is traditionally implemented as an ordinary file, with a few exceptions. It has no name in the file system, being created instead by the `pipe()` system call. Its size is fixed, and when a process attempts to write to a full pipe, the process is suspended. Once all data previously written into the pipe have been read out, writing continues at the beginning of the file (pipes are not true circular buffers). One benefit of the small size of pipes (usually 4,096 bytes) is that pipe data are seldom actually written to disk; they usually are kept in memory by the normal block buffer cache.

In FreeBSD pipes are implemented as a special case of the ***socket*** mechanism. The socket mechanism provides a general interface not only to facilities such as pipes, which are local to one machine, but also to networking facilities.

Even on the same machine, a pipe can be used only by two processes related through use of the `fork()` system call. The socket mechanism can be used by unrelated processes.

A socket is an endpoint of communication. A socket in use usually has an ***address*** bound to it. The nature of the address depends on the ***communication domain*** of the socket. A characteristic property of a domain is that processes communicating in the same domain use the same ***address format***. A single socket can communicate in only one domain.

The three domains currently implemented in FreeBSD are the UNIX domain (AF_UNIX), the Internet domain (AF_INET), and the XEROX Network Services (NS) domain (AF_NS). The address format of the UNIX domain is that of an ordinary file-system path name, such as */alpha/beta/gamma*. Processes communicating in the Internet domain use DARPA Internet communications protocols (such as TCP/IP) and Internet addresses, which consist of a 32-bit host number and a 32-bit port number (representing a rendezvous point on the host).

There are several **socket types**, which represent classes of services. Each type may or may not be implemented in any communication domain. If a type is implemented in a given domain, it may be implemented by one or more protocols, which may be selected by the user:

- **Stream sockets.** These sockets provide reliable, duplex, sequenced data streams. No data are lost or duplicated in delivery, and there are no record boundaries. This type is supported in the Internet domain by TCP. In the UNIX domain, pipes are implemented as a pair of communicating stream sockets.
- **Sequenced packet sockets.** These sockets provide data streams like those of stream sockets, except that record boundaries are provided. This type is used in the XEROX AF_NS protocol.
- **Datagram sockets.** These sockets transfer messages of variable size in either direction. There is no guarantee that such messages will arrive in the same order they were sent, or that they will be unduplicated, or that they will arrive at all, but the original message (or record) size is preserved in any datagram that does arrive. This type is supported in the Internet domain by UDP.
- **Reliably delivered message sockets.** These sockets transfer messages that are guaranteed to arrive and that otherwise are like the messages transferred using datagram sockets. This type is currently unsupported.
- **Raw sockets.** These sockets allow direct access by processes to the protocols that support the other socket types. The protocols accessible include not only the uppermost ones but also lower-level protocols. For example, in the Internet domain, it is possible to reach TCP, IP beneath that, or an Ethernet protocol beneath that. This capability is useful for developing new protocols.

A set of system calls is specific to the socket facility. The `socket()` system call creates a socket. It takes as arguments specifications of the communication domain, the socket type, and the protocol to be used to support that type. The value returned by the call is a small integer called a **socket descriptor**, which

occupies the same name space as file descriptors. The socket descriptor indexes the array of open files in the *u* structure in the kernel and has a file structure allocated for it. The FreeBSD file structure may point to a socket structure instead of to an inode. In this case, certain socket information (such as the socket's type, its message count, and the data in its input and output queues) is kept directly in the socket structure.

For another process to address a socket, the socket must have a name. A name is bound to a socket by the `bind()` system call, which takes the socket descriptor, a pointer to the name, and the length of the name as a byte string. The contents and length of the byte string depend on the address format. The `connect()` system call is used to initiate a connection. The arguments are syntactically the same as those for `bind()`; the socket descriptor represents the local socket, and the address is that of the foreign socket to which the attempt to connect is made.

Many processes that communicate using the socket IPC follow the client–server model. In this model, the *server* process provides a *service* to the *client* process. When the service is available, the server process listens on a well-known address, and the client process uses `connect()` to reach the server.

A server process uses `socket()` to create a socket and `bind()` to bind the well-known address of its service to that socket. Then, it uses the `listen()` system call to tell the kernel that it is ready to accept connections from clients and to specify how many pending connections the kernel should queue until the server can service them. Finally, the server uses the `accept()` system call to accept individual connections. Both `listen()` and `accept()` take as an argument the socket descriptor of the original socket. The system call `accept()` returns a new socket descriptor corresponding to the new connection; the original socket descriptor is still open for further connections. The server usually uses `fork()` to produce a new process after the `accept()` to service the client while the original server process continues to listen for more connections. There are also system calls for setting parameters of a connection and for returning the address of the foreign socket after an `accept()`.

When a connection for a socket type, such as a stream socket, is established, the addresses of both endpoints are known, and no further addressing information is needed to transfer data. The ordinary `read()` and `write()` system calls may then be used to transfer data.

The simplest way to terminate a connection, and to destroy the associated socket, is to use the `close()` system call on its socket descriptor. We may also wish to terminate only one direction of communication of a duplex connection; the `shutdown()` system call can be used for this purpose.

Some socket types, such as datagram sockets, do not support connections. Instead, their sockets exchange datagrams that must be addressed individually. The system calls `sendto()` and `recvfrom()` are used for such connections. Both take as arguments a socket descriptor, a buffer pointer and length, and an address-buffer pointer and length. The address buffer contains the appropriate address for `sendto()` and is filled in with the address of the datagram just received by `recvfrom()`. The number of data actually transferred is returned by both system calls.

The `select()` system call can be used to multiplex data transfers on several file descriptors and/or socket descriptors. It can even be used to allow one server process to listen for client connections for many services and to `fork()`

a process for each connection as the connection is made. The server does a `socket()`, `bind()`, and `listen()` for each service and then does a `select()` on all the socket descriptors. When `select()` indicates activity on a descriptor, the server does an `accept()` on it and forks a process on the new descriptor returned by `accept()`, leaving the parent process to do a `select()` again.

C.9.2 Network Support

Almost all current UNIX systems support the UUCP network facilities, which are mostly used over dial-up telephone lines to support the UUCP mail network and the USENET news network. These are, however, rudimentary networking facilities; they do not support even remote login, much less remote procedure calls or distributed file systems. These facilities are almost completely implemented as user processes and are not part of the operating system itself.

FreeBSD supports the DARPA Internet protocols UDP, TCP, IP, and ICMP on a wide range of Ethernet, token-ring, and ARPANET interfaces. The framework in the kernel to support these protocols is intended to facilitate the implementation of further protocols, and all protocols are accessible via the socket interface. Rob Gurwitz of BBN wrote the first version of the code as an add-on package for 4.1BSD.

The International Standards Organization's (ISO) Open System Interconnection (OSI) Reference Model for networking prescribes seven layers of network protocols and strict methods of communication between them. An implementation of a protocol may communicate only with a peer entity speaking the same protocol at the same layer or with the protocol–protocol interface of a protocol in the layer immediately above or below in the same system. The ISO networking model is implemented in FreeBSD Reno and 4.4BSD.

The FreeBSD networking implementation, and to a certain extent the socket facility, is more oriented toward the ARPANET Reference Model (ARM). The ARPANET in its original form served as a proof of concept for many networking ideas, such as packet switching and protocol layering. The ARPANET was retired in 1988 because the hardware that supported it was no longer state of the art. Its successors, such as the NSFNET and the Internet, are even larger and serve as communications utilities for researchers and test-beds for Internet gateway research. The ARM predates the ISO model; the ISO model was in large part inspired by the ARPANET research.

Although the ISO model is often interpreted as setting a limit of one protocol communicating per layer, the ARM allows several protocols in the same layer. There are only four protocol layers in the ARM:

- **Process/applications.** This layer subsumes the application, presentation, and session layers of the ISO model. Such user-level programs as the file-transfer protocol (FTP) and Telnet (remote login) exist at this level.
- **Host–host.** This layer corresponds to ISO's transport and the top part of its network layers. Both the Transmission Control Protocol (TCP) and the Internet Protocol (IP) are in this layer, with TCP on top of IP. TCP corresponds to an ISO transport protocol, and IP performs the addressing functions of the ISO network layer.
- **Network interface.** This layer spans the lower part of the ISO network layer and the entire data-link layer. The protocols involved here depend on the

physical network type. The ARPANET uses the IMP-Host protocols, whereas an Ethernet uses Ethernet protocols.

- **Network hardware.** The ARM is primarily concerned with software, so there is no explicit network hardware layer. However, any actual network will have hardware corresponding to the ISO physical layer.

The networking framework in FreeBSD is more generalized than either the ISO model or the ARM, although it is most closely related to the ARM (Figure C.12).

User processes communicate with network protocols (and thus with other processes on other machines) via the socket facility. This facility corresponds to the ISO session layer, as it is responsible for setting up and controlling communications.

Sockets are supported by protocols—possibly by several, layered one on another. A protocol may provide services such as reliable delivery, suppression of duplicate transmissions, flow control, and addressing, depending on the socket type being supported and the services required by any higher protocols.

A protocol may communicate with another protocol or with the network interface that is appropriate for the network hardware. There is little restriction in the general framework on what protocols may communicate with what other protocols or on how many protocols may be layered on top of one another. The user process may, by means of the raw socket type, directly access any layer of protocol from the uppermost used to support one of the other socket types, such as streams, down to a raw network interface. This capability is used by routing processes and also for new protocol development.

Most often, there is one **network-interface driver** per network controller type. The network interface is responsible for handling characteristics specific to the local network being addressed. This arrangement ensures that the protocols using the interface do not need to be concerned with these characteristics.

The functions of the network interface depend largely on the **network hardware**, which is whatever is necessary for the network. Some networks may

<table>
<tr><th>ISO reference model</th><th>ARPANET reference model</th><th>4.2BSD layers</th><th>example layering</th></tr>
<tr><td>application</td><td rowspan="3">process applications</td><td rowspan="2">user programs and libraries</td><td rowspan="2">telnet</td></tr>
<tr><td>presentation</td></tr>
<tr><td>session transport</td><td>sockets</td><td>sock_stream</td></tr>
<tr><td></td><td rowspan="2">host–host</td><td rowspan="2">protocol</td><td>TCP</td></tr>
<tr><td rowspan="2">network data link</td><td>IP</td></tr>
<tr><td>hardware</td><td>network interface</td><td>network interfaces</td><td>Ethernet driver</td></tr>
<tr><td></td><td>network hardware</td><td>network hardware</td><td>interlan controller</td></tr>
</table>

Figure C.12 Network reference models and layering.

support reliable transmission at this level, but most do not. Some networks provide broadcast addressing, but many do not.

The socket facility and the networking framework use a common set of memory buffers, or *mbufs*. These are intermediate in size between the large buffers used by the block I/O system and the C-lists used by character devices. An *mbuf* is 128 bytes long; 112 bytes may be used for data, and the rest is used for pointers to link the *mbuf* into queues and for indicators of how much of the data area is actually in use.

Data are ordinarily passed between layers—socket–protocol, protocol–protocol, or protocol–network interface—in *mbufs*. The ability to pass the buffers containing the data eliminates some data copying, but there is still frequently a need to remove or add protocol headers. It is also convenient and efficient for many purposes to be able to hold data that occupy an area the size of the memory-management page. Thus, the data of an *mbuf* may reside not in the *mbuf* itself but elsewhere in memory. There is an *mbuf* page table for this purpose, as well as a pool of pages dedicated to *mbuf* use.

C.10 Summary

The early advantages of UNIX were that it was written in a high-level language, was distributed in source form, and provided powerful operating-system primitives on an inexpensive platform. These advantages led to UNIX's popularity at educational, research, and government institutions and eventually in the commercial world. This popularity produced many strains of UNIX with varying and improved facilities.

UNIX provides a file system with tree-structured directories. The kernel supports files as unstructured sequences of bytes. Direct access and sequential access are supported through system calls and library routines.

Files are stored as an array of fixed-size data blocks with perhaps a trailing fragment. The data blocks are found by pointers in the inode. Directory entries point to inodes. Disk space is allocated from cylinder groups to minimize head movement and to improve performance.

UNIX is a multiprogrammed system. Processes can easily create new processes with the `fork()` system call. Processes can communicate with pipes or, more generally, sockets. They may be grouped into jobs that may be controlled with signals.

Processes are represented by two structures: the process structure and the user structure. CPU scheduling is a priority algorithm with dynamically computed priorities that reduces to round-robin scheduling in the extreme case.

FreeBSD memory management uses swapping supported by paging. A `pagedaemon` process uses a modified second-chance page-replacement algorithm to keep enough free frames to support the executing processes.

Page and file I/O uses a block buffer cache to minimize the amount of actual I/O. Terminal devices use a separate character-buffering system.

Networking support is one of the most important features in FreeBSD. The socket concept provides the programming mechanism to access other processes, even across a network. Sockets provide an interface to several sets of protocols.

Further Reading

[McKusick et al. (2015)] provides a good general discussion of FreeBSD. A modern scheduler for FreeBSD is described in [Roberson (2003)]. Locking in the Multithreaded FreeBSD Kernel is described in [Baldwin (2002)].

FreeBSD is described in *The FreeBSD Handbook*, which can be downloaded from http://www.freebsd.org.

Bibliography

[Baldwin (2002)] J. Baldwin, "Locking in the Multithreaded FreeBSD Kernel", *USENIX BSD* (2002).

[McKusick et al. (2015)] M. K. McKusick, G. V. Neville-Neil, and R. N. M. Watson, *The Design and Implementation of the FreeBSD UNIX Operating System–Second Edition*, Pearson (2015).

[Roberson (2003)] J. Roberson, "ULE: A Modern Scheduler For FreeBSD", *Proceedings of the USENIX BSDCon Conference* (2003), pages 17–28.

The Mach System

This chapter was first written in 1991 and has been updated over time but is no longer modified.

In this appendix we examine the Mach operating system. Mach is designed to incorporate the many recent innovations in operating-system research to produce a fully functional, technically advanced system. Unlike UNIX, which was developed without regard for multiprocessing, Mach incorporates multiprocessing support throughout. This support is exceedingly flexible, accommodating shared-memory systems as well as systems with no memory shared between processors. Mach is designed to run on computer systems ranging from one processor to thousands of processors. In addition, it is easily ported to many varied computer architectures. A key goal of Mach is to be a distributed system capable of functioning on heterogeneous hardware.

Although many experimental operating systems are being designed, built, and used, Mach satisfies the needs of most users better than the others because it offers full compatibility with UNIX 4.3 BSD. This compatibility also gives us a unique opportunity to compare two functionally similar, but internally dissimilar, operating systems. Mach and UNIX differ in their emphases, so our Mach discussion does not exactly parallel our UNIX discussion. In addition, we do not include a section on the user interface, because that component is similar to the user interface in 4.3 BSD. As you will see, Mach provides the ability to layer emulation of other operating systems as well; other operating systems can even run concurrently with Mach.

D.1 History of the Mach System

Mach traces its ancestry to the Accent operating system developed at Carnegie Mellon University (CMU). Although Accent pioneered a number of novel operating-system concepts, its utility was limited by its inability to execute UNIX applications and its strong ties to a single hardware architecture, which made it difficult to port. Mach's communication system and philosophy are derived from Accent, but many other significant portions of the system (for example, the virtual memory system and the management of tasks and threads) were developed from scratch. An important goal of the Mach effort was support for multiprocessors.

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 4.3 BSD 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; 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 (Figure D.1) 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 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 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 to 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 release of OSF/1 occurred a year later, and it now competes with UNIX System V, Release 4, the operating system of choice among ***UNIX International (UI)*** members. OSF members include key technological companies such as IBM, DEC, and HP. Mach 2.5 is also the basis for the operating system on the NeXT workstation, the brainchild of Steve Jobs, of Apple Computer fame. OSF is evaluating Mach 3 as the basis for a future

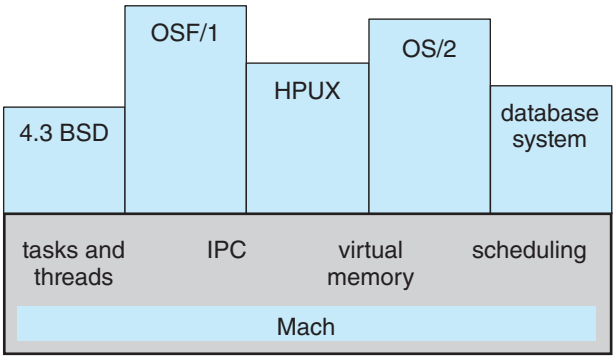


Figure D.1 Mach 3 structure.

operating-system release, and research on Mach continues at CMU, OSF, and elsewhere.

D.2 Design Principles

The Mach operating system was designed to provide basic mechanisms that most current operating systems lack. The goal is to design an operating system that is BSD-compatible and, in addition, excels in the following areas:

- Support for diverse architectures, including multiprocessors with varying degrees of shared memory access: uniform memory access (UMA), non-uniform memory access (NUMA), and no remote memory access (NORMA)
- Ability to function with varying intercomputer network speeds, from wide-area networks to high-speed local-area networks and tightly coupled multiprocessors
- Simplified kernel structure, with a small number of abstractions (in turn, these abstractions are sufficiently general to allow other operating systems to be implemented on top of Mach.)
- Distributed operation, providing network transparency to clients and an object-oriented organization both internally and externally
- Integrated memory management and interprocess communication, to provide efficient communication of large numbers of data as well as communication-based memory management
- Heterogeneous system support, to make Mach widely available and inter-operable among computer systems from multiple vendors

The designers of Mach have been heavily influenced by BSD (and by UNIX in general), whose benefits include

- A simple programmer interface, with a good set of primitives and a consistent set of interfaces to system facilities
- Easy portability to a wide class of single processors
- An extensive library of utilities and applications
- The ability to combine utilities easily via pipes

Of course, the designers also wanted to redress what they saw as the drawbacks of BSD:

- A kernel that has become the repository of many redundant features—and that consequently is difficult to manage and modify
- Original design goals that made it difficult to provide support for multiprocessors, distributed systems, and shared program libraries (for instance, because the kernel was designed for single processors, it has no provisions for locking code or data that other processors might be using.)

- Too many fundamental abstractions, providing too many similar, competing means with which to accomplish the same tasks

The development of Mach continues to be a huge undertaking. The benefits of such a system are equally large, however. The operating system runs on many existing single-processor and multiprocessor architectures, and it can be easily ported to future ones. It makes research easier, because computer scientists can add features via user-level code, instead of having to write their own tailor-made operating system. Areas of experimentation include operating systems, databases, reliable distributed systems, multiprocessor languages, security, and distributed artificial intelligence. In its current version, the Mach system is usually as efficient as other major versions of UNIX when performing similar tasks.

D.3 System Components

To achieve the design goals of Mach, the developers reduced the operating-system functionality to a small set of basic abstractions, out of which all other functionality can be derived. The Mach approach is to place as little as possible within the kernel but to make what is there powerful enough that all other features can be implemented at the user level.

Mach's design philosophy is to have a simple, extensible kernel, concentrating on communication facilities. For instance, all requests to the kernel, and all data movement among processes, are handled through one communication mechanism. Mach is therefore able to provide system-wide protection to its users by protecting the communication mechanism. Optimizing this one communication path can result in significant performance gains, and it is simpler than trying to optimize several paths. Mach is extensible, because many traditionally kernel-based functions can be implemented as user-level servers. For instance, all pagers (including the default pager) can be implemented externally and called by the kernel for the user.

Mach is an example of an object-oriented system where the data and the operations that manipulate that data are encapsulated into an abstract object. Only the operations of the object are able to act on the entities defined in it. The details of how these operations are implemented are hidden, as are the internal data structures. Thus, a programmer can use an object only by invoking its defined, exported operations. A programmer can change the internal operations without changing the interface definition, so changes and optimizations do not affect other aspects of system operation. The object-oriented approach supported by Mach allows objects to reside anywhere in a network of Mach systems, transparent to the user. The `port` mechanism, discussed later in this section, makes all of this possible.

Mach's primitive abstractions are the heart of the system and are as follows:

- A **task** is an execution environment that provides the basic unit of resource allocation. It consists of a virtual address space and protected access to system resources via ports, and it may contain one or more threads.
- A **thread** is the basic unit of execution and must run in the context of a task (which provides the address space). All threads within a task share the

task's resources (ports, memory, and so on). There is no notion of a ***process*** in Mach. Rather, a traditional process is implemented as a task with a single thread of control.

- A **port** is the basic object-reference mechanism in Mach and is implemented as a kernel-protected communication channel. Communication is accomplished by sending messages to ports; messages are queued at the destination port if no thread is immediately ready to receive them. Ports are protected by kernel-managed capabilities, or **port rights**. A task must have a port right to send a message to a port. The programmer invokes an operation on an object by ***sending*** a message to a port associated with that object. The object being represented by a port ***receives*** the messages.
- A **port set** is a group of ports sharing a common message queue. A thread can receive messages for a port set and thus service multiple ports. Each received message identifies the individual port (within the set) from which it was received. The receiver can use this to identify the object referred to by the message.
- A **message** is the basic method of communication between threads in Mach. It is a typed collection of data objects. For each object, it may contain the actual data or a pointer to out-of-line data. Port rights are passed in messages; this is the only way to move them among tasks. (Passing a port right in shared memory does not work, because the Mach kernel will not permit the new task to use a right obtained in this manner.)
- A **memory object** is a source of memory. Tasks can access it by mapping portions of an object (or the entire object) into their address spaces. The object can be managed by a user-mode external memory manager. One example is a file managed by a file server; however, a memory object can be any object for which memory-mapped access makes sense. A mapped buffer implementation of a UNIX pipe is another example.

Figure D.2 illustrates these abstractions, which we explain further in the remainder of this chapter.

An unusual feature of Mach, and a key to the system's efficiency, is the blending of memory and interprocess-communication (IPC) features. Whereas some other distributed systems (such as Solaris, with its NFS features) have special-purpose extensions to the file system to extend it over a network, Mach provides a general-purpose, extensible merger of memory and messages at the heart of its kernel. This feature not only allows Mach to be used for distributed and parallel programming but also helps in the implementation of the kernel itself.

Mach connects memory management and IPC by allowing each to be used in the implementation of the other. Memory management is based on the use of memory objects. A memory object is represented by a port (or ports), and IPC messages are sent to this port to request operations (for example, `pagein`, `pageout`) on the object. Because IPC is used, memory objects can reside on remote systems and be accessed transparently. The kernel caches the contents of memory objects in local memory. Conversely, memory-management techniques are used in the implementation of message passing. Where possible,

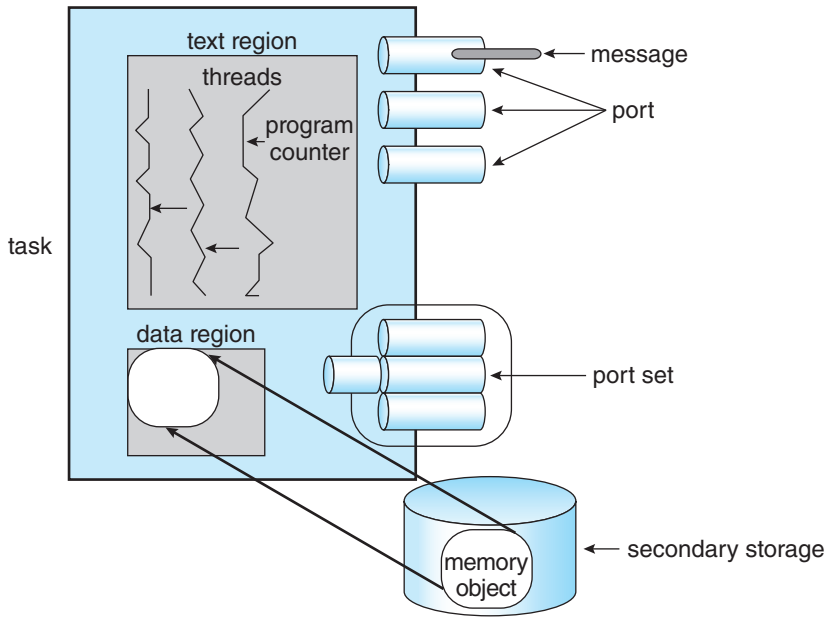


Figure D.2 Mach's basic abstractions.

Mach passes messages by moving pointers to shared memory objects, rather than by copying the objects themselves.

IPC tends to involve considerable system overhead. For intrasystem messages, it is generally less efficient than communication accomplished through shared memory. Because Mach is a message-based kernel, message handling must be carried out efficiently. Most of the inefficiency of message handling in traditional operating systems is due to either the copying of messages from one task to another (for intracomputer messages) or low network-transfer speed (for intercomputer messages). To solve these problems, Mach uses virtual memory remapping to transfer the contents of large messages. In other words, the message transfer modifies the receiving task's address space to include a copy of the message contents. Virtual copy (or copy-on-write) techniques are used to avoid or delay the actual copying of the data. This approach has several advantages:

- Increased flexibility in memory management for user programs
- Greater generality, allowing the virtual copy approach to be used in tightly and loosely coupled computers
- Improved performance over UNIX message passing
- Easier task migration (because ports are location independent, a task and all its ports can be moved from one machine to another. All tasks that previously communicated with the moved task can continue to do so because they reference the task only by its ports and communicate via messages to these ports.)

In the sections that follow, we detail the operation of process management, IPC, and memory management. Then, we discuss Mach's chameleonlike ability to support multiple operating-system interfaces.

D.4 Process Management

A task can be thought of as a traditional process that does not have an instruction pointer or a register set. A task contains a virtual address space, a set of port rights, and accounting information. A task is a passive entity that does nothing unless it has one or more threads executing in it.

D.4.1 Basic Structure

A task containing one thread is similar to a UNIX process. Just as a `fork()` system call produces a new UNIX process, Mach creates a new task by using `fork()`. The new task's memory is a duplicate of the parent's address space, as dictated by the **inheritance attributes** of the parent's memory. The new task contains one thread, which is started at the same point as the creating `fork()` call in the parent. Threads and tasks can also be suspended and resumed.

Threads are especially useful in server applications, which are common in UNIX, since one task can have multiple threads to service multiple requests to the task. Threads also allow efficient use of parallel computing resources. Rather than having one process on each processor, with the corresponding performance penalty and operating-system overhead, a task can have its threads spread among parallel processors. Threads add efficiency to user-level programs as well. For instance, in UNIX, an entire process must wait when a page fault occurs or when a system call is executed. In a task with multiple threads, only the thread that causes the page fault or executes a system call is delayed; all other threads continue executing. Because Mach has kernel-supported threads (Chapter 4), the threads have some cost associated with them. They must have supporting data structures in the kernel, and more complex kernel-scheduling algorithms must be provided. These algorithms and thread states are discussed in Chapter 4.

At the user level, threads may be in one of two states:

- **Running.** The thread is either executing or waiting to be allocated a processor. A thread is considered to be running even if it is blocked within the kernel (waiting for a page fault to be satisfied, for instance).
- **Suspended.** The thread is neither executing on a processor nor waiting to be allocated a processor. A thread can resume its execution only if it is returned to the running state.

These two states are also associated with a task. An operation on a task affects all threads in a task, so suspending a task involves suspending all the threads in it. Task and thread suspensions are separate, independent mechanisms, however, so resuming a thread in a suspended task does not resume the task.

Mach provides primitives from which thread-synchronization tools can be built. This practice is consistent with Mach's philosophy of providing mini-

mum yet sufficient functionality in the kernel. The Mach IPC facility can be used for synchronization, with processes exchanging messages at rendezvous points. Thread-level synchronization is provided by calls to start and stop threads at appropriate times. A ***suspend count*** is kept for each thread. This count allows multiple `suspend()` calls to be executed on a thread, and only when an equal number of `resume()` calls occur is the thread resumed. Unfortunately, this feature has its own limitation. Because it is an error for a `start()` call to be executed before a `stop()` call (the suspend count would become negative), these routines cannot be used to synchronize shared data access. However, `wait()` and `signal()` operations associated with semaphores, and used for synchronization, can be implemented via the IPC calls. We discuss this method in Section D.5.

D.4.2 The C Threads Package

Mach provides low-level but flexible routines instead of polished, large, and more restrictive functions. Rather than making programmers work at this low level, Mach provides many higher-level interfaces for programming in C and other languages. For instance, the C threads package provides multiple threads of control, shared variables, mutual exclusion for critical sections, and condition variables for synchronization. In fact, C threads is one of the major influences on the POSIX Pthreads standard, which many operating systems support. As a result, there are strong similarities between the C threads and Pthreads programming interfaces. The thread-control routines include calls to perform these tasks:

- Create a new thread within a task, given a function to execute and parameters as input. The thread then executes concurrently with the creating thread, which receives a thread identifier when the call returns.
- Destroy the calling thread, and return a value to the creating thread.
- Wait for a specific thread to terminate before allowing the calling thread to continue. This call is a synchronization tool, much like the UNIX `wait()` system calls.
- Yield use of a processor, signaling that the scheduler can run another thread at this point. This call is also useful in the presence of a preemptive scheduler, as it can be used to relinquish the CPU voluntarily before the time quantum (or scheduling interval) expires if a thread has no use for the CPU.

Mutual exclusion is achieved through the use of spinlocks, as discussed in Chapter 6. The routines associated with mutual exclusion are these:

- The routine `mutex_alloc()` dynamically creates a mutex variable.
- The routine `mutex_free()` deallocates a dynamically created mutex variable.
- The routine `mutex_lock()` locks a mutex variable. The executing thread loops in a spinlock until the lock is attained. A deadlock results if a thread with a lock tries to lock the same mutex variable. Bounded waiting is

not guaranteed by the C threads package. Rather, it is dependent on the hardware instructions used to implement the mutex routines.

- The routine `mutex_unlock()` unlocks a mutex variable, much like the typical `signal()` operation of a semaphore.

General synchronization without busy waiting can be achieved through the use of condition variables, which can be used to implement a monitor, as described in Chapter 6. A condition variable is associated with a mutex variable and reflects a Boolean state of that variable. The routines associated with general synchronization are these:

- The routine `condition_alloc()` dynamically allocates a condition variable.
- The routine `condition_free()` deletes a dynamically created condition variable allocated as a result of `condition_alloc()`.
- The routine `condition_wait()` unlocks the associated mutex variable and blocks the thread until a `condition_signal()` is executed on the condition variable, indicating that the event being waited for may have occurred. The mutex variable is then locked, and the thread continues. A `condition_signal()` does not guarantee that the condition still holds when the unblocked thread finally returns from its `condition_wait()` call, so the awakened thread must loop, executing the `condition_wait()` routine until it is unblocked and the condition holds.

As an example of the C threads routines, consider the bounded-buffer synchronization problem of Section 7.1.1. The producer and consumer are represented as threads that access the common bounded-buffer pool. We use a mutex variable to protect the buffer while it is being updated. Once we have exclusive access to the buffer, we use condition variables to block the producer thread if the buffer is full and to block the consumer thread if the buffer is empty. As in Chapter 6, we assume that the buffer consists of `n` slots, each capable of holding one item. The `mutex` semaphore provides mutual exclusion for accesses to the buffer pool and is initialized to the value 1. The `empty` and `full` semaphores count the number of empty and full buffers, respectively. The semaphore `empty` is initialized to the value `n`; the semaphore `full` is initialized to the value 0. The condition variable `nonempty` is true while the buffer has items in it, and `nonfull` is true if the buffer has an empty slot. The first step includes the allocation of the mutex and condition variables:

```
mutex_alloc(mutex);
condition_alloc(nonempty, nonfull);
```

The code for the producer thread is shown in Figure D.3, and the code for the consumer thread is shown in Figure D.4. When the program terminates, the mutex and condition variables need to be deallocated:

```
mutex_free(mutex);
condition_free(nonempty, nonfull);
```

```
do {
   . . .
   // produce an item into nextp
   . . .
   mutex_lock(mutex);
   while(full)
      condition_wait(nonfull, mutex);
   . . .
   // add nextp to buffer
   . . .
   condition_signal(nonempty);
   mutex_unlock(mutex);
} while(TRUE);
```

Figure D.3 The structure of the producer process.

D.4.3 The CPU Scheduler

The CPU scheduler for a thread-based multiprocessor operating system is more complex than its process-based relatives. There are generally more threads in a multithreaded system than there are processes in a multitasking system. Keeping track of multiple processors is also difficult and is a relatively new area of research. Mach uses a simple policy to keep the scheduler manageable. Only threads are scheduled, so no knowledge of tasks is needed in the scheduler. All threads compete equally for resources, including time quanta.

Each thread has an associated priority number ranging from 0 through 127, which is based on the exponential average of its usage of the CPU. That is, a thread that recently used the CPU for a large amount of time has the lowest

```
do {
   mutex_lock(mutex);
   while(empty)
      condition_wait(nonempty, mutex);
   . . .
   // remove an item from the buffe to nextc
   . . .
   condition_signal(nonfull);
   mutex_unlock(mutex);
   . . .
   // consume the item in nextc
   . . .
} until(FALSE);
```

Figure D.4 The structure of the consumer process.

priority. Mach uses the priority to place the thread in one of 32 global run queues. These queues are searched in priority order for waiting threads when a processor becomes idle. Mach also keeps per-processor, or local, run queues. A local run queue is used for threads that are bound to an individual processor. For instance, a device driver for a device connected to an individual CPU must run on only that CPU.

Instead of a central dispatcher that assigns threads to processors, each processor consults the local and global run queues to select the appropriate next thread to run. Threads in the local run queue have absolute priority over those in the global queues, because it is assumed that they are performing some chore for the kernel. The run queues—like most other objects in Mach—are locked when they are modified to avoid simultaneous changes by multiple processors. To speed dispatching of threads on the global run queue, Mach maintains a list of idle processors.

Additional scheduling difficulties arise from the multiprocessor nature of Mach. A fixed time quantum is not appropriate because, for instance, there may be fewer runnable threads than there are available processors. It would be wasteful to interrupt a thread with a context switch to the kernel when that thread's quantum runs out, only to have the thread be placed right back in the running state. Thus, instead of using a fixed-length quantum, Mach varies the size of the time quantum inversely with the total number of threads in the system. It keeps the time quantum over the entire system constant, however. For example, in a system with 10 processors, 11 threads, and a 100-millisecond quantum, a context switch needs to occur on each processor only once per second to maintain the desired quantum.

Of course, complications still exist. Even relinquishing the CPU while waiting for a resource is more difficult than it is on traditional operating systems. First, a thread must issue a call to alert the scheduler that the thread is about to block. This alert avoids race conditions and deadlocks, which could occur when the execution takes place in a multiprocessor environment. A second call actually causes the thread to be moved off the run queue until the appropriate event occurs. The scheduler uses many other internal thread states to control thread execution.

D.4.4 Exception Handling

Mach was designed to provide a single, simple, consistent exception-handling system, with support for standard as well as user-defined exceptions. To avoid redundancy in the kernel, Mach uses kernel primitives whenever possible. For instance, an exception handler is just another thread in the task in which the exception occurs. Remote procedure call (RPC) messages are used to synchronize the execution of the thread causing the exception (the ***victim***) and that of the handler and to communicate information about the exception between the victim and handler. Mach exceptions are also used to emulate the BSD `signal` package.

Disruptions to normal program execution come in two varieties: internally generated exceptions and external interrupts. Interrupts are asynchronously generated disruptions of a thread or task, whereas exceptions are caused by the occurrence of unusual conditions during a thread's execution. Mach's general-purpose exception facility is used for error detection and debugger support.

This facility is also useful for other functions, such as taking a core dump of a bad task, allowing tasks to handle their own errors (mostly arithmetic), and emulating instructions not implemented in hardware.

Mach supports two different granularities of exception handling. Error handling is supported by per-thread exception handling, whereas debuggers use per-task handling. It makes little sense to try to debug only one thread or to have exceptions from multiple threads invoke multiple debuggers. Aside from this distinction, the only difference between the two types of exceptions lies in their inheritance from a parent task. Task-wide exception-handling facilities are passed from the parent to child tasks, so debuggers are able to manipulate an entire tree of tasks. Error handlers are not inherited and default to no handler at thread- and task-creation time. Finally, error handlers take precedence over debuggers if the exceptions occur simultaneously. The reason for this approach is that error handlers are normally part of the task and therefore should execute normally even in the presence of a debugger.

Exception handling proceeds as follows:

- The victim thread causes notification of an exception's occurrence via a `raise()` RPC message sent to the handler.
- The victim then calls a routine to wait until the exception is handled.
- The handler receives notification of the exception, usually including information about the exception, the thread, and the task causing the exception.
- The handler performs its function according to the type of exception. The handler's action involves ***clearing*** the exception, causing the victim to ***resume***, or ***terminating*** the victim thread.

To support the execution of BSD programs, Mach needs to support BSD-style signals. Signals provide software-generated interrupts and exceptions. Unfortunately, signals are of limited functionality in multithreaded operating systems. The first problem is that, in UNIX, a signal's handler must be a routine in the process receiving the signal. If the signal is caused by a problem in the process itself (for example, a division by 0), the problem cannot be remedied, because a process has limited access to its own context. A second, more troublesome aspect of signals is that they were designed for only single-threaded programs. For instance, it makes no sense for all threads in a task to get a signal, but how can a signal be seen by only one thread?

Because the signal system must work correctly with multithreaded applications for Mach to run 4.3 BSD programs, signals could not be abandoned. Producing a functionally correct signal package required several rewrites of the code, however. A final problem with UNIX signals is that they can be lost. This loss occurs when another signal of the same type occurs before the first is handled. Mach exceptions are queued as a result of their RPC implementation.

Externally generated signals, including those sent from one BSD process to another, are processed by the BSD server section of the Mach 2.5 kernel. Their behavior is therefore the same as it is under BSD. Hardware exceptions are a different matter, because BSD programs expect to receive hardware exceptions as signals. Therefore, a hardware exception caused by a thread must arrive at the thread as a signal. So that this result is produced, hardware exceptions are

converted to exception RPCs. For tasks and threads that do not make explicit use of the Mach exception-handling facility, the destination of this RPC defaults to an in-kernel task. This task has only one purpose: Its thread runs in a continuous loop, receiving the exception RPCs. For each RPC, it converts the exception into the appropriate signal, which is sent to the thread that caused the hardware exception. It then completes the RPC, clearing the original exception condition. With the completion of the RPC, the initiating thread reenters the run state. It immediately sees the signal and executes its signal-handling code. In this manner, all hardware exceptions begin in a uniform way—as exception RPCs. Threads not designed to handle such exceptions, however, receive the exceptions as they would on a standard BSD system—as signals. In Mach 3.0, the signal-handling code is moved entirely into a server, but the overall structure and flow of control is similar to those of Mach 2.5.

D.5 Interprocess Communication

Most commercial operating systems, such as UNIX, provide communication between processes and between hosts with fixed, global names (or Internet addresses). There is no location independence of facilities, because any remote system needing to use a facility must know the name of the system providing that facility. Usually, data in the messages are untyped streams of bytes. Mach simplifies this picture by sending messages between location-independent ports. The messages contain typed data for ease of interpretation. All BSD communication methods can be implemented with this simplified system.

The two components of Mach IPC are ports and messages. Almost everything in Mach is an object, and all objects are addressed via their communication ports. Messages are sent to these ports to initiate operations on the objects by the routines that implement the objects. By depending on only ports and messages for all communication, Mach delivers location independence of objects and security of communication. Data independence is provided by the NetMsgServer task (Section D.5.3).

Mach ensures security by requiring that message senders and receivers have ***rights***. A right consists of a port name and a capability—send or receive—on that port, and is much like a capability in object-oriented systems. Only one task may have receive rights to any given port, but many tasks may have send rights. When an object is created, its creator also allocates a port to represent the object and obtains the access rights to that port. Rights can be given out by the creator of the object, including the kernel, and are passed in messages. If the holder of a receive right sends that right in a message, the receiver of the message gains the right, and the sender loses it. A task may allocate ports to allow access to any objects it owns or for communication. The destruction of either a port or the holder of the receive right causes the revocation of all rights to that port, and the tasks holding send rights can be notified if desired.

D.5.1 Ports

A port is implemented as a protected, bounded queue within the kernel of the system on which the object resides. If a queue is full, a sender may abort the

send, wait for a slot to become available in the queue, or have the kernel deliver the message.

Several system calls provide the port with the following functionalities:

- Allocate a new port in a specified task and give the caller's task all access rights to the new port. The port name is returned.
- Deallocate a task's access rights to a port. If the task holds the receive right, the port is destroyed, and all other tasks with send rights are, potentially, notified.
- Get the current status of a task's port.
- Create a backup port, which is given the receive right for a port if the task containing the receive right requests its deallocation or terminates.

When a task is created, the kernel creates several ports for it. The function `task_self()` returns the name of the port that represents the task in calls to the kernel. For instance, to allocate a new port, a task calls `port_allocate()` with `task_self()` as the name of the task that will own the port. Thread creation results in a similar `thread_self()` thread kernel port. This scheme is similar to the standard process-ID concept found in UNIX. Another port is returned by `task_notify()`; this is the port to which the kernel will send event-notification messages (such as notifications of port terminations).

Ports can also be collected into ***port sets***. This facility is useful if one thread is to service requests coming in on multiple ports—for example, for multiple objects. A port may be a member of no more than one port set at a time. Furthermore, if a port is in a set, it may not be used directly to receive messages. Instead, messages will be routed to the port set's queue. A port set may not be passed in messages, unlike a port. Port sets are objects that serve a purpose similar to the 4.3 BSD `select()` system call, but they are more efficient.

D.5.2 Messages

A message consists of a fixed-length header and a variable number of typed data objects. The header contains the destination's port name, the name of the reply port to which return messages should be sent, and the length of the message (Figure D.5). The data in the message (or in-line data) were limited to less than 8 KB in Mach 2.5 systems, but Mach 3.0 has no limit. Each data section may be a simple type (numbers or characters), port rights, or pointers to out-of-line data. Each section has an associated type, so that the receiver can unpack the data correctly even if it uses a byte ordering different from that used by the sender. The kernel also inspects the message for certain types of data. For instance, the kernel must process port information within a message, either by translating the port name into an internal port data structure address or by forwarding it for processing to the NetMsgServer (Section D.5.3).

The use of pointers in a message provides the means to transfer the entire address space of a task in one single message. The kernel also must process pointers to out-of-line data, since a pointer to data in the sender's address space would be invalid in the receiver's—especially if the sender and receiver reside on different systems. Generally, systems send messages by copying the data from the sender to the receiver. Because this technique can be inefficient, especially for large messages, Mach takes a different approach. The data refer-

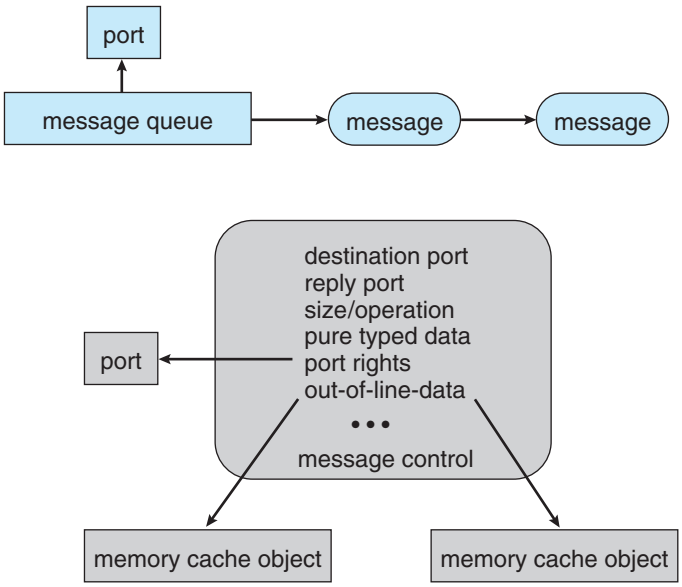


Figure D.5 Mach messages.

enced by a pointer in a message being sent to a port on the same system are not copied between the sender and the receiver. Instead, the address map of the receiving task is modified to include a copy-on-write copy of the pages of the message. This operation is *much* faster than a data copy and makes message passing more efficient. In essence, message passing is implemented via virtual memory management.

In Version 2.5, this operation was implemented in two phases. A pointer to a region of memory caused the kernel to map that region into its own space temporarily, setting the sender's memory map to copy-on-write mode to ensure that any modifications did not affect the original version of the data. When a message was received at its destination, the kernel moved its mapping to the receiver's address space, using a newly allocated region of virtual memory within that task.

In Version 3, this process was simplified. The kernel creates a data structure that would be a copy of the region if it were part of an address map. On receipt, this data structure is added to the receiver's map and becomes a copy accessible to the receiver.

The newly allocated regions in a task do not need to be contiguous with previous allocations, so Mach virtual memory is said to be ***sparse***, consisting of regions of data separated by unallocated addresses. A full message transfer is shown in Figure D.6.

D.5.3 The NetMsgServer

For a message to be sent between computers, the message's destination must be located, and the message must be transmitted to the destination. UNIX traditionally leaves these mechanisms to the low-level network protocols, which require the use of statically assigned communication endpoints (for example,

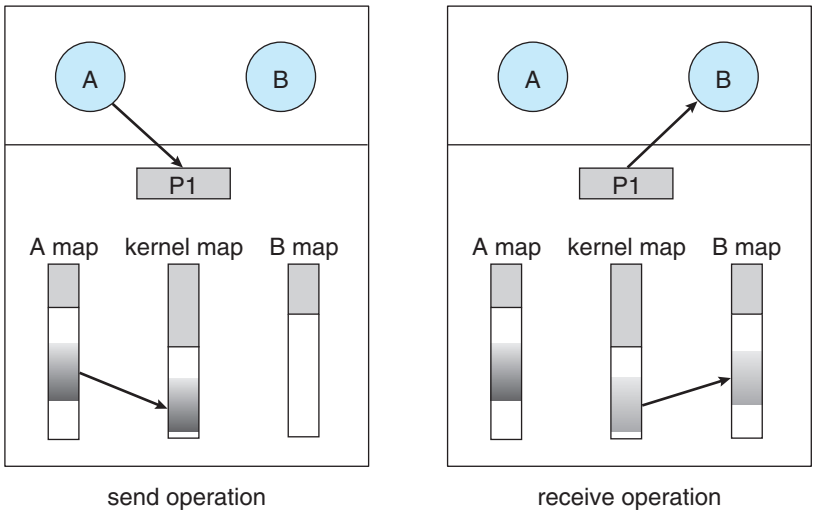


Figure D.6 Mach message transfer.

the port number for services based on TCP or UDP). One of Mach's tenets is that all objects within the system are location independent and that the location is transparent to the user. This tenet requires Mach to provide location-transparent naming and transport to extend IPC across multiple computers.

This naming and transport are performed by the **Network Message Server** (**NetMsgServer**), a user-level, capability-based networking daemon that forwards messages between hosts. It also provides a primitive network-wide name service that allows tasks to register ports for lookup by tasks on any other computer in the network. Mach ports can be transferred only in messages, and messages must be sent to ports. The primitive name service solves the problem of transferring the first port. Subsequent IPC interactions are fully transparent, because the NetMsgServer tracks all rights and out-of-line memory passed in intercomputer messages and arranges for the appropriate transfers. The NetMsgServers maintain among themselves a distributed database of port rights that have been transferred between computers and of the ports to which these rights correspond.

The kernel uses the NetMsgServer when a message needs to be sent to a port that is not on the kernel's computer. Mach's kernel IPC is used to transfer the message to the local NetMsgServer. The NetMsgServer then uses whatever network protocols are appropriate to transfer the message to its peer on the other computer. The notion of a NetMsgServer is protocol independent, and NetMsgServers have been built to use various protocols. Of course, the NetMsgServers involved in a transfer must agree on the protocol used. Finally, the NetMsgServer on the destination computer uses that kernel's IPC to send the message to the correct destination task.

The ability to extend local IPC transparently across nodes is supported by the use of proxy ports. When a send right is transferred from one computer to another, the NetMsgServer on the destination computer creates a new port, or proxy, to represent the original port at the destination. Messages sent to this proxy are received by the NetMsgServer and are forwarded transparently

to the original port. This procedure is one example of how NetMsgServers cooperate to make a proxy indistinguishable from the original port.

Because Mach is designed to function in a network of heterogeneous systems, it must provide a way for systems to send data formatted in a way that is understandable by both the sender and the receiver. Unfortunately, computers differ in the formats they use to store various types of data. For instance, an integer on one system might take 2 bytes to store, and the most significant byte might be stored before the least significant one. Another system might reverse this ordering. The NetMsgServer therefore uses the type information stored in a message to translate the data from the sender's to the receiver's format. In this way, all data are represented correctly when they reach their destination.

The NetMsgServer on a given computer accepts RPCs that add, look up, and remove network ports from the NetMsgServer's name service. As a security precaution, a port value provided in an add request for a port must match that in the remove request for a thread to ask for a port name to be removed from the database.

As an example of the NetMsgServer's operation, consider a thread on node A sending a message to a port that happens to be in a task on node B. The program simply sends a message to a port to which it has a send right. The message is first passed to the kernel, which delivers it to its first recipient, the NetMsgServer on node A. The NetMsgServer then contacts (through its database information) the NetMsgServer on node B and sends the message. The NetMsgServer on node B presents the message to the kernel with the appropriate local port for node B. The kernel finally provides the message to the receiving task when a thread in that task executes a `msg_receive()` call. This sequence of events is shown in Figure D.7.

Mach 3.0 provides an alternative to the NetMsgServer as part of its improved support for NORMA multiprocessors. The NORMA IPC subsystem of Mach 3.0 implements functionality similar to the NetMsgServer directly in the

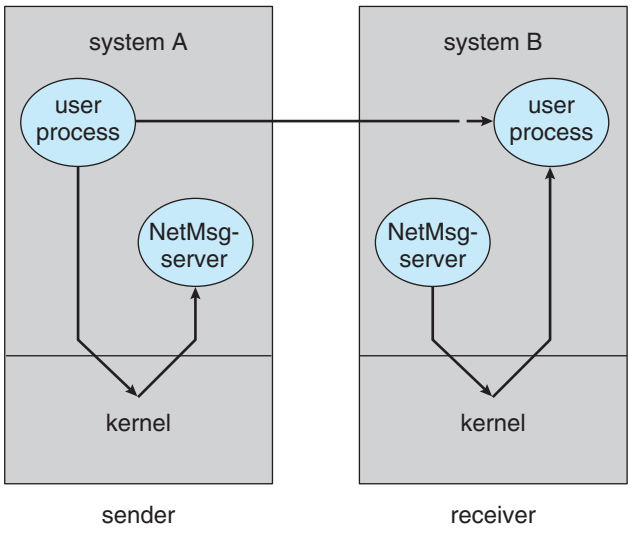


Figure D.7 Network IPC forwarding by NetMsgServer.

Mach kernel, providing much more efficient internode IPC for multicomputers with fast interconnection hardware. For example, the time-consuming copying of messages between the NetMsgServer and the kernel is eliminated. Use of the NORMA IPC does not preclude use of the NetMsgServer; the NetMsgServer can still be used to provide Mach IPC service over networks that link a NORMA multiprocessor to other computers. In addition to the NORMA IPC, Mach 3.0 also provides support for memory management across a NORMA system and enables a task in such a system to create child tasks on nodes other than its own. These features support the implementation of a single-system-image operating system on a NORMA multiprocessor. The multiprocessor behaves like one large system rather than an assemblage of smaller systems (for both users and applications).

D.5.4 Synchronization Through IPC

The IPC mechanism is extremely flexible and is used throughout Mach. For example, it may be used for thread synchronization. A port may be used as a synchronization variable and may have n messages sent to it for n resources. Any thread wishing to use a resource executes a receive call on that port. The thread will receive a message if the resource is available. Otherwise, it will wait on the port until a message is available there. To return a resource after use, the thread can send a message to the port. In this regard, receiving is equivalent to the semaphore operation `wait()`, and sending is equivalent to `signal()`. This method can be used for synchronizing semaphore operations among threads in the same task, but it cannot be used for synchronization among tasks, because only one task may have receive rights to a port. For more general-purpose semaphores, a simple daemon can be written to implement the same method.

D.6 Memory Management

Given the object-oriented nature of Mach, it is not surprising that a principal abstraction in Mach is the memory object. Memory objects are used to manage secondary storage and generally represent files, pipes, or other data that are mapped into virtual memory for reading and writing (Figure D.8). Memory objects may be backed by user-level memory managers, which take the place of the more traditional kernel-incorporated virtual memory pager found in other operating systems. In contrast to the traditional approach of having the kernel manage secondary storage, Mach treats secondary-storage objects (usually files) as it does all other objects in the system. Each object has a port associated with it and may be manipulated by messages sent to its port. Memory objects —unlike the memory-management routines in monolithic, traditional kernels —allow easy experimentation with new memory-manipulation algorithms.

D.6.1 Basic Structure

The virtual address space of a task is generally sparse, consisting of many holes of unallocated space. For instance, a memory-mapped file is placed in some set of addresses. Large messages are also transferred as shared memory segments. For each of these segments, a section of virtual memory address is used to provide the threads with access to the message. As new items are mapped or

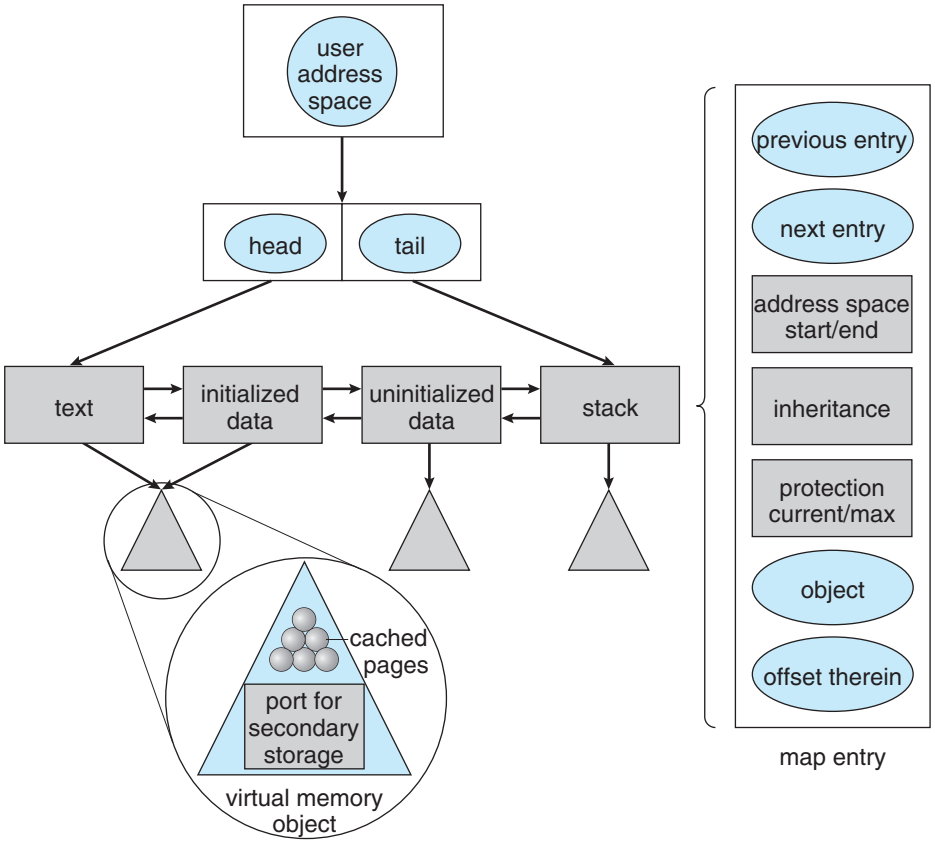


Figure D.8 Mach virtual memory task address map.

removed from the address space, holes of unallocated memory appear in the address space.

Mach makes no attempt to compress the address space, although a task may fail (or crash) if it has no room for a requested region in its address space. Given that address spaces are 4 GB or more, this limitation is not currently a problem. However, maintaining a regular page table for a 4-GB address space for each task, especially one with holes in it, would use excessive amounts of memory (1 MB or more). The key to sparse address spaces is that page-table space is used only for currently allocated regions. When a page fault occurs, the kernel must check to see whether the page is in a valid region, rather than simply indexing into the page table and checking the entry. Although the resulting lookup is more complex, the benefits of reduced memory-storage requirements and simpler address-space maintenance make the approach worthwhile.

Mach also has system calls to support standard virtual memory functionality, including the allocation, deallocation, and copying of virtual memory. When allocating a new virtual memory object, the thread may provide an address for the object or may let the kernel choose the address. Physical memory is not allocated until pages in this object are accessed. The object's backing store is managed by the default pager (Section D.6.2). Virtual memory objects

are also allocated automatically when a task receives a message containing out-of-line data.

Associated system calls return information about a memory object in a task's address space, change the access protection of the object, and specify how an object is to be passed to child tasks at the time of their creation (shared, copy-on-write, or not present).

D.6.2 User-Level Memory Managers

A secondary-storage object is usually mapped into the virtual address space of a task. Mach maintains a cache of memory-resident pages of all mapped objects, as in other virtual memory implementations. However, a page fault occurring when a thread accesses a nonresident page is executed as a message to the object's port. The concept that a memory object can be created and serviced by nonkernel tasks (unlike threads, for instance, which are created and maintained only by the kernel) is important. The end result is that, in the traditional sense, memory can be paged by user-written memory managers. When the object is destroyed, it is up to the memory manager to write back any changed pages to secondary storage. No assumptions are made by Mach about the content or importance of memory objects, so the memory objects are independent of the kernel.

In several circumstances, user-level memory managers are insufficient. For instance, a task allocating a new region of virtual memory might not have a memory manager assigned to that region, since it does not represent a secondary-storage object (but must be paged), or a memory manager might fail to perform pageout. Mach itself also needs a memory manager to take care of its memory needs. For these cases, Mach provides a default memory manager. The Mach 2.5 default memory manager uses the standard file system to store data that must be written to disk, rather than requiring a separate swap space, as in 4.3 BSD. In Mach 3.0 (and OSF/1), the default memory manager is capable of using either files in a standard file system or dedicated disk partitions. The default memory manager has an interface similar to that of the user-level ones, but with some extensions to support its role as the memory manager that can be relied on to perform pageout when user-level managers fail to do so.

Pageout policy is implemented by an internal kernel thread, the ***pageout daemon***. A paging algorithm based on FIFO with second chance (Section 10.4.5) is used to select pages for replacement. The selected pages are sent to the appropriate manager (either user level or default) for actual pageout. A user-level manager may be more intelligent than the default manager, and it may implement a different paging algorithm suitable to the object it is backing (that is, by selecting some other page and forcibly paging it out). If a user-level manager fails to reduce the resident set of pages when asked to do so by the kernel, the default memory manager is invoked, and it pages out the user-level manager to reduce the user-level manager's resident set size. Should the user-level manager recover from the problem that prevented it from performing its own pageouts, it will touch these pages (causing the kernel to page them in again) and can then page them out as it sees fit.

If a thread needs access to data in a memory object (for instance, a file), it invokes the `vm_map()` system call. Included in this system call is a port that identifies the object and the memory manager that is responsible for the

region. The kernel executes calls on this port when data are to be read or written in that region. An added complexity is that the kernel makes these calls asynchronously, since it would not be reasonable for the kernel to be waiting on a user-level thread. Unlike the situation with pageout, the kernel has no recourse if its request is not satisfied by the external memory manager. The kernel has no knowledge of the contents of an object or of how that object must be manipulated.

Memory managers are responsible for the consistency of the contents of a memory object mapped by tasks on different machines. (Tasks on a single machine share a single copy of a mapped memory object.) Consider a situation in which tasks on two different machines attempt to modify the same page of an object at the same time. It is up to the manager to decide whether these modifications must be serialized. A conservative manager implementing strict memory consistency would force the modifications to be serialized by granting write access to only one kernel at a time. A more sophisticated manager could allow both accesses to proceed concurrently (for example, if the manager knew that the two tasks were modifying distinct areas within the page and that it could merge the modifications successfully at some future time). Most external memory managers written for Mach (for example, those implementing mapped files) do not implement logic for dealing with multiple kernels, due to the complexity of such logic.

When the first `vm_map()` call is made on a memory object, the kernel sends a message to the memory manager port passed in the call, invoking the `memory_manager_init()` routine, which the memory manager must provide as part of its support of a memory object. The two ports passed to the memory manager are a **control port** and a **name port**. The control port is used by the memory manager to provide data to the kernel—for example, pages to be made resident. Name ports are used throughout Mach. They do not receive messages but are used simply as points of reference and comparison. Finally, the memory object must respond to a `memory_manager_init()` call with a `memory_object_set_attributes()` call to indicate that it is ready to accept requests. When all tasks with send rights to a memory object relinquish those rights, the kernel deallocates the object's ports, thus freeing the memory manager and memory object for destruction.

Several kernel calls are needed to support external memory managers. The `vm_map()` call was just discussed. In addition, some commands get and set attributes and provide page-level locking when it is required (for instance, after a page fault has occurred but before the memory manager has returned the appropriate data). Another call is used by the memory manager to pass a page (or multiple pages, if read-ahead is being used) to the kernel in response to a page fault. This call is necessary since the kernel invokes the memory manager asynchronously. Finally, several calls allow the memory manager to report errors to the kernel.

The memory manager itself must provide support for several calls so that it can support an object. We have already discussed `memory_object_init()` and others. When a thread causes a page fault on a memory object's page, the kernel sends a `memory_object_data_request()` to the memory object's port on behalf of the faulting thread. The thread is placed in a wait state until the memory manager either returns the page in a `memory_object_data_provided()` call or returns an appropriate error to the kernel. Any of the pages that have

been modified, or any "precious pages" that the kernel needs to remove from resident memory (due to page aging, for instance), are sent to the memory object via `memory_object_data_write()`. ***Precious pages*** are pages that may not have been modified but that cannot be discarded as they otherwise would be because the memory manager no longer retains a copy. The memory manager declares these pages to be precious and expects the kernel to return them when they are removed from memory. Precious pages save unnecessary duplication and copying of memory.

In the current version, Mach does not allow external memory managers to affect the page-replacement algorithm directly. Mach does not export the memory-access information that would be needed for an external task to select the least recently used page, for instance. Methods of providing such information are currently under investigation. An external memory manager is still useful for a variety of reasons, however:

- It may reject the kernel's replacement victim if it knows of a better candidate (for instance, MRU page replacement).
- It may monitor the memory object it is backing and request pages to be paged out before the memory usage invokes Mach's pageout daemon.
- It is especially important in maintaining consistency of secondary storage for threads on multiple processors, as we show in Section D.6.3.
- It can control the order of operations on secondary storage to enforce consistency constraints demanded by database management systems. For example, in transaction logging, transactions must be written to a log file on disk before they modify the database data.
- It can control mapped file access.

D.6.3 Shared Memory

Mach uses shared memory to reduce the complexity of various system facilities, as well as to provide these features in an efficient manner. Shared memory generally provides extremely fast interprocess communication, reduces overhead in file management, and helps to support multiprocessing and database management. Mach does not use shared memory for all these traditional shared-memory roles, however. For instance, all threads in a task share that task's memory, so no formal shared-memory facility is needed within a task. However, Mach must still provide traditional shared memory to support other operating-system constructs, such as the UNIX `fork()` system call.

It is obviously difficult for tasks on multiple machines to share memory and to maintain data consistency. Mach does not try to solve this problem directly; rather, it provides facilities to allow the problem to be solved. Mach supports consistent shared memory only when the memory is shared by tasks running on processors that share memory. A parent task is able to declare which regions of memory are to be ***inherited*** by its children and which are to be readable –writable. This scheme is different from copy-on-write inheritance, in which each task maintains its own copy of any changed pages. A writable object is addressed from each task's address map, and all changes are made to the same copy. The threads within the tasks are responsible for coordinating changes to memory so that they do not interfere with one another (by writing to the

same location concurrently). This coordination can be done through normal synchronization methods: critical sections or mutual-exclusion locks.

For the case of memory shared among separate machines, Mach allows the use of external memory managers. If a set of unrelated tasks wishes to share a section of memory, the tasks can use the same external memory manager and access the same secondary-storage areas through it. The implementor of this system would need to write the tasks and the external pager. This pager could be as simple or as complicated as needed. A simple implementation would allow no readers while a page was being written to. Any write attempt would cause the pager to invalidate the page in all tasks currently accessing it. The pager would then allow the write and would revalidate the readers with the new version of the page. The readers would simply wait on a page fault until the page again became available. Mach provides such a memory manager: the Network Memory Server (NetMemServer). For multicomputers, the NORMA configuration of Mach 3.0 provides similar support as a standard part of the kernel. This XMM subsystem allows multicomputer systems to use external memory managers that do not incorporate logic for dealing with multiple kernels. The XMM subsystem is responsible for maintaining data consistency among multiple kernels that share memory and makes these kernels appear to be a single kernel to the memory manager. The XMM subsystem also implements virtual copy logic for the mapped objects that it manages. This virtual copy logic includes both copy-on-reference among multicomputer kernels and sophisticated copy-on-write optimizations.

D.7 Programmer Interface

A programmer can work at several levels within Mach. There is, of course, the system-call level, which, in Mach 2.5, is equivalent to the 4.3 BSD system-call interface. Version 2.5 includes most of 4.3 BSD as one thread in the kernel. A BSD system call traps to the kernel and is serviced by this thread on behalf of the caller, much as standard BSD would handle it. The emulation is not multithreaded, so it has limited efficiency.

Mach 3.0 has moved from the single-server model to support of multiple servers. It has therefore become a true microkernel without the full features normally found in a kernel. Rather, full functionality can be provided via emulation libraries, servers, or a combination of the two. In keeping with the definition of a microkernel, the emulation libraries and servers run outside the kernel at user level. In this way, multiple operating systems can run concurrently on one Mach 3.0 kernel.

An emulation library is a set of routines that lives in a read-only part of a program's address space. Any operating-system calls the program makes are translated into subroutine calls to the library. Single-user operating systems, such as MS-DOS and the Macintosh operating system, have been implemented solely as emulation libraries. For efficiency reasons, the emulation library lives in the address space of the program needing its functionality; in theory, however, it could be a separate task.

More complex operating systems are emulated through the use of libraries and one or more servers. System calls that cannot be implemented in the library are redirected to the appropriate server. Servers can be multithreaded for

improved efficiency; BSD and OSF/1 are implemented as single multithreaded servers. Systems can be decomposed into multiple servers for greater modularity.

Functionally, a system call starts in a task and passes through the kernel before being redirected, if appropriate, to the library in the task's address space or to a server. Although this extra transfer of control decreases the efficiency of Mach, this decrease is balanced to some extent by the ability of multiple threads to execute BSD-like code concurrently.

At the next higher programming level is the C threads package. This package is a run-time library that provides a C language interface to the basic Mach threads primitives. It provides convenient access to these primitives, including routines for the forking and joining of threads, mutual exclusion through mutex variables (Section D.4.2), and synchronization through use of condition variables. Unfortunately, it is not appropriate for the C threads package to be used between systems that share no memory (NORMA systems), since it depends on shared memory to implement its constructs. There is currently no equivalent of C threads for NORMA systems. Other run-time libraries have been written for Mach, including threads support for other languages.

Although the use of primitives makes Mach flexible, it also makes many programming tasks repetitive. For instance, significant amounts of code are associated with sending and receiving messages in each task that uses messages (which, in Mach, is most tasks). The designers of Mach therefore provide an interface generator (or stub generator) called *MIG*. MIG is essentially a compiler that takes as input a definition of the interface to be used (declarations of variables, types, and procedures) and generates the RPC interface code needed to send and receive the messages fitting this definition and to connect the messages to the sending and receiving threads.

D.8 Summary

The Mach operating system is designed to incorporate the many recent innovations in operating-system research to produce a fully functional, technically advanced operating system.

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

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

As we have shown, Mach is well on its way to achieving these goals.

Mach 2.5 includes 4.3 BSD in its kernel, which provides the emulation needed but enlarges the kernel. This 4.3 BSD code has been rewritten to provide the same 4.3 functionality but to use the Mach primitives. This change allows the 4.3 BSD support code to run in user space on a Mach 3.0 system.

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.

Further Reading

The Accent operating system was described by [Rashid and Robertson (1981)]. A historical overview of the progression from an even earlier system, RIG, through Accent to Mach was given by [Rashid (1986)]. General discussions concerning the Mach model were offered by [Tevanian et al. (1989)].

[Accetta et al. (1986)] presented an overview of the original design of Mach. 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 [Tevanian et al. (1987b)].

Bibliography

[Accetta et al. (1986)] M. Accetta, R. Baron, W. Bolosky, D. B. Golub, R. Rashid, A. Tevanian, and M. Young, "Mach: A New Kernel Foundation for UNIX Development", *Proceedings of the Summer USENIX Conference* (1986), pages 93–112.

[Black (1990)] D. L. Black, "Scheduling Support for Concurrency and Parallelism in the Mach Operating System", *IEEE Computer*, Volume 23, Number 5 (1990), pages 35–43.

[Rashid (1986)] R. F. Rashid, "From RIG to Accent to Mach: The Evolution of a Network Operating System", *Proceedings of the ACM/IEEE Computer Society, Fall Joint Computer Conference* (1986), pages 1128–1137.

[Rashid and Robertson (1981)] R. Rashid and G. Robertson, "Accent: A Communication-Oriented Network Operating System Kernel", *Proceedings of the ACM Symposium on Operating System Principles* (1981), pages 64–75.

[Tevanian et al. (1987a)] A. Tevanian, Jr., R. F. Rashid, D. B. Golub, D. L. Black, E. Cooper, and M. W. Young, "Mach Threads and the Unix Kernel: The Battle for Control", *Proceedings of the Summer USENIX Conference* (1987).

[Tevanian et al. (1987b)] A. Tevanian, Jr., R. F. Rashid, M. W. Young, D. B. Golub, M. R. Thompson, W. Bolosky, and R. Sanzi, "A UNIX Interface for Shared

Memory and Memory Mapped Files Under Mach", Technical report, Carnegie-Mellon University (1987).

[Tevanian et al. (1989)] A. Tevanian, Jr., and B. Smith, "Mach: The Model for Future Unix", *Byte* (1989).

Additional Exercises

Exercises

Chapter 1 Exercises

1.12 How do clustered systems differ from multiprocessor systems? What is required for two machines belonging to a cluster to cooperate to provide a highly available service?

1.13 Consider a computing cluster consisting of two nodes running a database. Describe two ways in which the cluster software can manage access to the data on the disk. Discuss the benefits and disadvantages of each.

1.14 What is the purpose of interrupts? How does an interrupt differ from a trap? Can traps be generated intentionally by a user program? If so, for what purpose?

1.15 Explain how the Linux kernel variables `HZ` and `jiffies` can be used to determine the number of seconds the system has been running since it was booted.

1.16 Direct memory access is used for high-speed I/O devices in order to avoid increasing the CPU's execution load.

a. How does the CPU interface with the device to coordinate the transfer?

b. How does the CPU know when the memory operations are complete?

c. The CPU is allowed to execute other programs while the DMA controller is transferring data. Does this process interfere with the execution of the user programs? If so, describe what forms of interference are caused.

1.17 Some computer systems do not provide a privileged mode of operation in hardware. Is it possible to construct a secure operating system for these computer systems? Give arguments both that it is and that it is not possible.

1.18 Many SMP systems have different levels of caches; one level is local to each processing core, and another level is shared among all processing cores. Why are caching systems designed this way?

1.19 Rank the following storage systems from slowest to fastest:

a. Hard-disk drives
b. Registers
c. Optical disk
d. Main memory
e. Nonvolatile memory
f. Magnetic tapes
g. Cache

1.20 Consider an SMP system similar to the one shown in Figure 1.8. Illustrate with an example how data residing in memory could in fact have a different value in each of the local caches.

1.21 Discuss, with examples, how the problem of maintaining coherence of cached data manifests itself in the following processing environments:

a. Single-processor systems
b. Multiprocessor systems
c. Distributed systems

1.22 Describe a mechanism for enforcing memory protection in order to prevent a program from modifying the memory associated with other programs.

1.23 Which network configuration—LAN or WAN—would best suit the following environments?

a. A campus student union
b. Several campus locations across a statewide university system
c. A neighborhood

1.24 Describe some of the challenges of designing operating systems for mobile devices compared with designing operating systems for traditional PCs.

1.25 What are some advantages of peer-to-peer systems over client–server systems?

1.26 Describe some distributed applications that would be appropriate for a peer-to-peer system.

1.27 Identify several advantages and several disadvantages of open-source operating systems. Identify the types of people who would find each aspect to be an advantage or a disadvantage.

Chapter 2 Exercises

2.9 The services and functions provided by an operating system can be divided into two main categories. Briefly describe the two categories, and discuss how they differ.

2.10 Describe three general methods for passing parameters to the operating system.

2.11 Describe how you could obtain a statistical profile of the amount of time a program spends executing different sections of its code. Discuss the importance of obtaining such a statistical profile.

2.12 What are the advantages and disadvantages of using the same system-call interface for manipulating both files and devices?

2.13 Would it be possible for the user to develop a new command interpreter using the system-call interface provided by the operating system?

2.14 Describe why Android uses ahead-of-time (AOT) rather than just-in-time (JIT) compilation.

2.15 What are the two models of interprocess communication? What are the strengths and weaknesses of the two approaches?

2.16 Contrast and compare an application programming interface (API) and an application binary interface (ABI).

2.17 Why is the separation of mechanism and policy desirable?

2.18 It is sometimes difficult to achieve a layered approach if two components of the operating system are dependent on each other. Identify a scenario in which it is unclear how to layer two system components that require tight coupling of their functionalities.

2.19 What is the main advantage of the microkernel approach to system design? How do user programs and system services interact in a microkernel architecture? What are the disadvantages of using the microkernel approach?

2.20 What are the advantages of using loadable kernel modules?

2.21 How are iOS and Android similar? How are they different?

2.22 Explain why Java programs running on Android systems do not use the standard Java API and virtual machine.

2.23 The experimental Synthesis operating system has an assembler incorporated in the kernel. To optimize system-call performance, the kernel assembles routines within kernel space to minimize the path that the system call must take through the kernel. This approach is the antithesis of the layered approach, in which the path through the kernel is extended to make building the operating system easier. Discuss the pros and cons of the Synthesis approach to kernel design and system-performance optimization.

Chapter 3 Exercises

3.8 Describe the actions taken by a kernel to context-switch between processes.

3.9 Construct a process tree similar to Figure 3.7. To obtain process information for the UNIX or Linux system, use the command `ps -ael`. Use the command `man ps` to get more information about the `ps` command. The task manager on Windows systems does not provide the parent process ID, but the ***process monitor*** tool, available from `technet.microsoft.com`, provides a process-tree tool.

3.10 Explain the role of the `init` (or `systemd`) process on UNIX and Linux systems in regard to process termination.

3.11 Including the initial parent process, how many processes are created by the program shown in Figure 3.32?

3.12 Explain the circumstances under which the line of code marked `printf("LINE J")` in Figure 3.33 will be reached.

3.13 Using the program in Figure 3.34, identify the values of `pid` at lines `A`, `B`, `C`, and `D`. (Assume that the actual pids of the parent and child are 2600 and 2603, respectively.)

3.14 Give an example of a situation in which ordinary pipes are more suitable than named pipes and an example of a situation in which named pipes are more suitable than ordinary pipes.

3.15 Consider the RPC mechanism. Describe the undesirable consequences that could arise from not enforcing either the "at most once" or "exactly once" semantic. Describe possible uses for a mechanism that has neither of these guarantees.

3.16 Using the program shown in Figure 3.35, explain what the output will be at lines `X` and `Y`.

```
#include <stdio.h>
#include <unistd.h>

int main()
{
    int i;

    for (i = 0; i < 4; i++)
       fork();

    return 0;
}
```

Figure 3.32 How many processes are created?

3.17 What are the benefits and the disadvantages of each of the following? Consider both the system level and the programmer level.

a. Synchronous and asynchronous communication

b. Automatic and explicit buffering

c. Send by copy and send by reference

d. Fixed-sized and variable-sized messages

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
       fprintf(stderr, "Fork Failed");
       return 1;
    }
    else if (pid == 0) { /* child process */
       execlp("/bin/ls","ls",NULL);
       printf("LINE J");
    }
    else { /* parent process */
       /* parent will wait for the child to complete */
       wait(NULL);
       printf("Child Complete");
    }

    return 0;
}
```

Figure 3.33 When will `LINE J` be reached?

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid, pid1;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
       fprintf(stderr, "Fork Failed");
       return 1;
    }
    else if (pid == 0) { /* child process */
       pid1 = getpid();
       printf("child: pid = %d",pid); /* A */
       printf("child: pid1 = %d",pid1); /* B */
    }
    else { /* parent process */
       pid1 = getpid();
       printf("parent: pid = %d",pid); /* C */
       printf("parent: pid1 = %d",pid1); /* D */
       wait(NULL);
    }

    return 0;
}
```

Figure 3.34 What are the pid values?

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

#define SIZE 5

int nums[SIZE] = {0,1,2,3,4};

int main()
{
int i;
pid_t pid;

   pid = fork();

   if (pid == 0) {
      for (i = 0; i < SIZE; i++) {
         nums[i] *= -i;
         printf("CHILD: %d ",nums[i]); /* LINE X */
      }
   }
   else if (pid > 0) {
      wait(NULL);
      for (i = 0; i < SIZE; i++)
         printf("PARENT: %d ",nums[i]); /* LINE Y */
   }

   return 0;
}
```

Figure 3.35 What output will be at Line X and Line Y?

Chapter 4 Exercises

4.8 Provide two programming examples in which multithreading does ***not*** provide better performance than a single-threaded solution.

4.9 Under what circumstances does a multithreaded solution using multiple kernel threads provide better performance than a single-threaded solution on a single-processor system?

4.10 Which of the following components of program state are shared across threads in a multithreaded process?

a. Register values

b. Heap memory

c. Global variables

d. Stack memory

4.11 Can a multithreaded solution using multiple user-level threads achieve better performance on a multiprocessor system than on a single-processor system? Explain.

4.12 In Chapter 3, we discussed Google's Chrome browser and its practice of opening each new tab in a separate process. Would the same benefits have been achieved if, instead, Chrome had been designed to open each new tab in a separate thread? Explain.

4.13 Is it possible to have concurrency but not parallelism? Explain.

4.14 Using Amdahl's Law, calculate the speedup gain for the following applications:

- 40 percent parallel with (a) eight processing cores and (b) sixteen processing cores
- 67 percent parallel with (a) two processing cores and (b) four processing cores
- 90 percent parallel with (a) four processing cores and (b) eight processing cores

4.15 Determine if the following problems exhibit task or data parallelism:

- Using a separate thread to generate a thumbnail for each photo in a collection
- Transposing a matrix in parallel
- A networked application where one thread reads from the network and another writes to the network
- The fork-join array summation application described in Section 4.5.2
- The Grand Central Dispatch system

4.16 A system with two dual-core processors has four processors available for scheduling. A CPU-intensive application is running on this system. All input is performed at program start-up, when a single file must be

opened. Similarly, all output is performed just before the program terminates, when the program results must be written to a single file. Between start-up and termination, the program is entirely CPU-bound. Your task is to improve the performance of this application by multithreading it. The application runs on a system that uses the one-to-one threading model (each user thread maps to a kernel thread).

- How many threads will you create to perform the input and output? Explain.
- How many threads will you create for the CPU-intensive portion of the application? Explain.

4.17 Consider the following code segment:

```
pid_t pid;

pid = fork();
if (pid == 0) { /* child process */
   fork();
   thread_create( . . .);
}
fork();
```

a. How many unique processes are created?

b. How many unique threads are created?

4.18 As described in Section 4.7.2, Linux does not distinguish between processes and threads. Instead, Linux treats both in the same way, allowing a task to be more akin to a process or a thread depending on the set of flags passed to the `clone()` system call. However, other operating systems, such as Windows, treat processes and threads differently. Typically, such systems use a notation in which the data structure for a process contains pointers to the separate threads belonging to the process. Contrast these two approaches for modeling processes and threads within the kernel.

4.19 The program shown in Figure 4.23 uses the Pthreads API. What would be the output from the program at `LINE C` and `LINE P`?

4.20 Consider a multicore system and a multithreaded program written using the many-to-many threading model. Let the number of user-level threads in the program be greater than the number of processing cores in the system. Discuss the performance implications of the following scenarios.

a. The number of kernel threads allocated to the program is less than the number of processing cores.

b. The number of kernel threads allocated to the program is equal to the number of processing cores.

c. The number of kernel threads allocated to the program is greater than the number of processing cores but less than the number of user-level threads.

```
#include <pthread.h>
#include <stdio.h>

int value = 0;
void *runner(void *param); /* the thread */

int main(int argc, char *argv[])
{
pid_t pid;
pthread_t tid;
pthread_attr_t attr;

   pid = fork();

   if (pid == 0) { /* child process */
      pthread_attr_init(&attr);
      pthread_create(&tid,&attr,runner,NULL);
      pthread_join(tid,NULL);
      printf("CHILD: value = %d",value); /* LINE C */
   }
   else if (pid > 0) { /* parent process */
      wait(NULL);
      printf("PARENT: value = %d",value); /* LINE P */
   }
}

void *runner(void *param) {
   value = 5;
   pthread_exit(0);
}
```

Figure 4.23 C program for Exercise 4.19.

4.21 Pthreads provides an API for managing thread cancellation. The `pthread_setcancelstate()` function is used to set the cancellation state. Its prototype appears as follows:

```
pthread_setcancelstate(int state, int *oldstate)
```

The two possible values for the state are `PTHREAD_CANCEL_ENABLE` and `PTHREAD_CANCEL_DISABLE`.

Using the code segment shown in Figure 4.24, provide examples of two operations that would be suitable to perform between the calls to disable and enable thread cancellation.

```
int oldstate;

pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, &oldstate);

/* What operations would be performed here? */

pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, &oldstate);
```

Figure 4.24 C program for Exercise 4.21.

Chapter 5 Exercises

5.11 Of these two types of programs:

a. I/O-bound

b. CPU-bound

which is more likely to have voluntary context switches, and which is more likely to have nonvoluntary context switches? Explain your answer.

5.12 Discuss how the following pairs of scheduling criteria conflict in certain settings.

a. CPU utilization and response time

b. Average turnaround time and maximum waiting time

c. I/O device utilization and CPU utilization

5.13 One technique for implementing **lottery scheduling** works by assigning processes lottery tickets, which are used for allocating CPU time. Whenever a scheduling decision has to be made, a lottery ticket is chosen at random, and the process holding that ticket gets the CPU. The BTV operating system implements lottery scheduling by holding a lottery 50 times each second, with each lottery winner getting 20 milliseconds of CPU time (20 milliseconds × 50 = 1 second). Describe how the BTV scheduler can ensure that higher-priority threads receive more attention from the CPU than lower-priority threads.

5.14 Most scheduling algorithms maintain a **run queue**, which lists processes eligible to run on a processor. On multicore systems, there are two general options: (1) each processing core has its own run queue, or (2) a single run queue is shared by all processing cores. What are the advantages and disadvantages of each of these approaches?

5.15 Consider the exponential average formula used to predict the length of the next CPU burst. What are the implications of assigning the following values to the parameters used by the algorithm?

a. $\alpha = 0$ and $\tau_0 = 100$ milliseconds

b. $\alpha = 0.99$ and $\tau_0 = 10$ milliseconds

5.16 A variation of the round-robin scheduler is the **regressive round-robin** scheduler. This scheduler assigns each process a time quantum and a priority. The initial value of a time quantum is 50 milliseconds. However, every time a process has been allocated the CPU and uses its entire time quantum (does not block for I/O), 10 milliseconds is added to its time quantum, and its priority level is boosted. (The time quantum for a process can be increased to a maximum of 100 milliseconds.) When a process blocks before using its entire time quantum, its time quantum is reduced by 5 milliseconds, but its priority remains the same. What type of process (CPU-bound or I/O-bound) does the regressive round-robin scheduler favor? Explain.

5.17 Consider the following set of processes, with the length of the CPU burst given in milliseconds:

Process	Burst Time	Priority
P_1	5	4
P_2	3	1
P_3	1	2
P_4	7	2
P_5	4	3

The processes are assumed to have arrived in the order P_1, P_2, P_3, P_4, P_5, all at time 0.

a. Draw four Gantt charts that illustrate the execution of these processes using the following scheduling algorithms: FCFS, SJF, nonpreemptive priority (a larger priority number implies a higher priority), and RR (quantum = 2).

b. What is the turnaround time of each process for each of the scheduling algorithms in part a?

c. What is the waiting time of each process for each of these scheduling algorithms?

d. Which of the algorithms results in the minimum average waiting time (over all processes)?

5.18 The following processes are being scheduled using a preemptive, priority-based, round-robin scheduling algorithm.

Process	Priority	Burst	Arrival
P_1	8	15	0
P_2	3	20	0
P_3	4	20	20
P_4	4	20	25
P_5	5	5	45
P_6	5	15	55

Each process is assigned a numerical priority, with a higher number indicating a higher relative priority. The scheduler will execute the highest-priority process. For processes with the same priority, a round-robin scheduler will be used with a time quantum of 10 units. If a process is preempted by a higher-priority process, the preempted process is placed at the end of the queue.

a. Show the scheduling order of the processes using a Gantt chart.

b. What is the turnaround time for each process?

c. What is the waiting time for each process?

5.19 The `nice` command is used to set the nice value of a process on Linux, as well as on other UNIX systems. Explain why some systems may allow any user to assign a process a nice value >= 0 yet allow only the root (or administrator) user to assign nice values < 0.

5.20 Which of the following scheduling algorithms could result in starvation?

a. First-come, first-served

b. Shortest job first

c. Round robin

d. Priority

5.21 Consider a variant of the RR scheduling algorithm in which the entries in the ready queue are pointers to the PCBs.

a. What would be the effect of putting two pointers to the same process in the ready queue?

b. What would be two major advantages and two disadvantages of this scheme?

c. How would you modify the basic RR algorithm to achieve the same effect without the duplicate pointers?

5.22 Consider a system running ten I/O-bound tasks and one CPU-bound task. Assume that the I/O-bound tasks issue an I/O operation once for every millisecond of CPU computing and that each I/O operation takes 10 milliseconds to complete. Also assume that the context-switching overhead is 0.1 millisecond and that all processes are long-running tasks. Describe the CPU utilization for a round-robin scheduler when:

a. The time quantum is 1 millisecond

b. The time quantum is 10 milliseconds

5.23 Consider a system implementing multilevel queue scheduling. What strategy can a computer user employ to maximize the amount of CPU time allocated to the user's process?

5.24 Consider a preemptive priority scheduling algorithm based on dynamically changing priorities. Larger priority numbers imply higher priority. When a process is waiting for the CPU (in the ready queue, but not running), its priority changes at a rate α. When it is running, its priority changes at a rate β. All processes are given a priority of 0 when they enter the ready queue. The parameters α and β can be set to give many different scheduling algorithms.

a. What is the algorithm that results from $\beta > \alpha > 0$?

b. What is the algorithm that results from $\alpha < \beta < 0$?

5.25 Explain the how the following scheduling algorithms discriminate either in favor of or against short processes:

a. FCFS

b. RR

c. Multilevel feedback queues

5.26 Describe why a shared ready queue might suffer from performance problems in an SMP environment.

5.27 Consider a load-balancing algorithm that ensures that each queue has approximately the same number of threads, independent of priority. How effectively would a priority-based scheduling algorithm handle this situation if one run queue had all high-priority threads and a second queue had all low-priority threads?

5.28 Assume that an SMP system has private, per-processor run queues. When a new process is created, it can be placed in either the same queue as the parent process or a separate queue.

a. What are the benefits of placing the new process in the same queue as its parent?

b. What are the benefits of placing the new process in a different queue?

5.29 Assume that a thread has blocked for network I/O and is eligible to run again. Describe why a NUMA-aware scheduling algorithm should reschedule the thread on the same CPU on which it previously ran.

5.30 Using the Windows scheduling algorithm, determine the numeric priority of each of the following threads.

a. A thread in the REALTIME_PRIORITY_CLASS with a relative priority of NORMAL

b. A thread in the ABOVE_NORMAL_PRIORITY_CLASS with a relative priority of HIGHEST

c. A thread in the BELOW_NORMAL_PRIORITY_CLASS with a relative priority of ABOVE_NORMAL

5.31 Assuming that no threads belong to the REALTIME_PRIORITY_CLASS and that none may be assigned a TIME_CRITICAL priority, what combination of priority class and priority corresponds to the highest possible relative priority in Windows scheduling?

5.32 Consider the scheduling algorithm in the Solaris operating system for time-sharing threads.

a. What is the time quantum (in milliseconds) for a thread with priority 15? With priority 40?

b. Assume that a thread with priority 50 has used its entire time quantum without blocking. What new priority will the scheduler assign this thread?

c. Assume that a thread with priority 20 blocks for I/O before its time quantum has expired. What new priority will the scheduler assign this thread?

5.33 Assume that two tasks, *A* and *B*, are running on a Linux system. The nice values of *A* and *B* are −5 and +5, respectively. Using the CFS scheduler as a guide, describe how the respective values of `vruntime` vary between the two processes given each of the following scenarios:

- Both *A* and *B* are CPU-bound.
- *A* is I/O-bound, and *B* is CPU-bound.
- *A* is CPU-bound, and *B* is I/O-bound.

5.34 Provide a specific circumstance that illustrates where rate-monotonic scheduling is inferior to earliest-deadline-first scheduling in meeting real-time process deadlines?

5.35 Consider two processes, P_1 and P_2, where $p_1 = 50$, $t_1 = 25$, $p_2 = 75$, and $t_2 = 30$.

a. Can these two processes be scheduled using rate-monotonic scheduling? Illustrate your answer using a Gantt chart such as the ones in Figure 5.21–Figure 5.24.

b. Illustrate the scheduling of these two processes using earliest-deadline-first (EDF) scheduling.

5.36 Explain why interrupt and dispatch latency times must be bounded in a hard real-time system.

5.37 Describe the advantages of using heterogeneous multiprocessing in a mobile system.

Chapter 6 Exercises

6.7 The pseudocode of Figure 6.15 illustrates the basic `push()` and `pop()` operations of an array-based stack. Assuming that this algorithm could be used in a concurrent environment, answer the following questions:

a. What data have a race condition?

b. How could the race condition be fixed?

6.8 Race conditions are possible in many computer systems. Consider an online auction system where the current highest bid for each item must be maintained. A person who wishes to bid on an item calls the `bid(amount)` function, which compares the amount being bid to the current highest bid. If the amount exceeds the current highest bid, the highest bid is set to the new amount. This is illustrated below:

```
void bid(double amount) {
   if (amount > highestBid)
      highestBid = amount;
}
```

```
push(item) {
   if (top < SIZE) {
      stack[top] = item;
      top++;
   }
   else
      ERROR
}

pop() {
   if (!is_empty()) {
      top--;
      return stack[top];
   }
   else
      ERROR
}

is_empty() {
   if (top == 0)
      return true;
   else
      return false;
}
```

Figure 6.15 Array-based stack for Exercise 6.7.

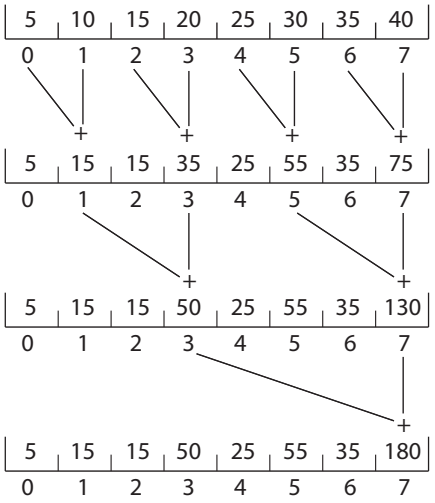


Figure 6.16 Summing an array as a series of partial sums for Exercise 6.9.

Describe how a race condition is possible in this situation and what might be done to prevent the race condition from occurring.

6.9 The following program example can be used to sum the array `values` of size N elements in parallel on a system containing N computing cores (there is a separate processor for each array element):

```
for j = 1 to log_2(N) {
   for k = 1 to N {
      if ((k + 1) % pow(2,j) == 0) {
         values[k] += values[k - pow(2,(j-1))]
      }
   }
}
```

This has the effect of summing the elements in the array as a series of partial sums, as shown in Figure 6.16. After the code has executed, the sum of all elements in the array is stored in the last array location. Are there any race conditions in the above code example? If so, identify where they occur and illustrate with an example. If not, demonstrate why this algorithm is free from race conditions.

6.10 The `compare_and_swap()` instruction can be used to design lock-free data structures such as stacks, queues, and lists. The program example shown in Figure 6.17 presents a possible solution to a lock-free stack using CAS instructions, where the stack is represented as a linked list of `Node` elements with `top` representing the top of the stack. Is this implementation free from race conditions?

```
typedef struct node {
   value_t data;
   struct node *next;
} Node;

Node *top; // top of stack

void push(value_t item) {
   Node *old_node;
   Node *new_node;

   new_node = malloc(sizeof(Node));
   new_node->data = item;

   do {
      old_node = top;
      new_node->next = old_node;
   }
   while (compare_and_swap(top,old_node,new_node) != old_node);
}

value_t pop() {
   Node *old_node;
   Node *new_node;

   do {
      old_node = top;
      if (old_node == NULL)
         return NULL;
      new_node = old_node->next;
   }
   while (compare_and_swap(top,old_node,new_node) != old_node);

   return old_node->data;
}
```

Figure 6.17 Lock-free stack for Exercise 6.10.

6.11 One approach for using `compare_and_swap()` for implementing a spinlock is as follows:

```
void lock_spinlock(int *lock) {
   while (compare_and_swap(lock, 0, 1) != 0)
      ; /* spin */
}
```

A suggested alternative approach is to use the "compare and compare-and-swap" idiom, which checks the status of the lock before invoking the

compare_and_swap() operation. (The rationale behind this approach is to invoke compare_and_swap()only if the lock is currently available.) This strategy is shown below:

```
void lock_spinlock(int *lock) {
{
   while (true) {
      if (*lock == 0) {
         /* lock appears to be available */

         if (!compare_and_swap(lock, 0, 1))
            break;
      }
   }
}
```

Does this "compare and compare-and-swap" idiom work appropriately for implementing spinlocks? If so, explain. If not, illustrate how the integrity of the lock is compromised.

6.12 Some semaphore implementations provide a function getValue() that returns the current value of a semaphore. This function may, for instance, be invoked prior to calling wait() so that a process will only call wait() if the value of the semaphore is > 0, thereby preventing blocking while waiting for the semaphore. For example:

```
if (getValue(&sem) > 0)
   wait(&sem);
```

Many developers argue against such a function and discourage its use. Describe a potential problem that could occur when using the function getValue() in this scenario.

6.13 The first known correct software solution to the critical-section problem for two processes was developed by Dekker. The two processes, P_0 and P_1, share the following variables:

```
boolean flag[2]; /* initially false */
int turn;
```

The structure of process P_i (i == 0 or 1) is shown in Figure 6.18. The other process is P_j (j == 1 or 0). Prove that the algorithm satisfies all three requirements for the critical-section problem.

6.14 The first known correct software solution to the critical-section problem for n processes with a lower bound on waiting of $n - 1$ turns was presented by Eisenberg and McGuire. The processes share the following variables:

```
enum pstate {idle, want_in, in_cs};
pstate flag[n];
int turn;
```

```
while (true) {
   flag[i] = true;

   while (flag[j]) {
      if (turn == j) {
         flag[i] = false;
         while (turn == j)
            ; /* do nothing */
         flag[i] = true;
      }
   }

      /* critical section */

   turn = j;
   flag[i] = false;

      /* remainder section */
}
```

Figure 6.18 The structure of process P_i in Dekker's algorithm.

All the elements of `flag` are initially `idle`. The initial value of `turn` is immaterial (between 0 and `n-1`). The structure of process P_i is shown in Figure 6.19. Prove that the algorithm satisfies all three requirements for the critical-section problem.

6.15 Explain why implementing synchronization primitives by disabling interrupts is not appropriate in a single-processor system if the synchronization primitives are to be used in user-level programs.

6.16 Consider how to implement a mutex lock using the `compare_and_swap()` instruction. Assume that the following structure defining the mutex lock is available:

```
typedef struct {
     int available;
} lock;
```

The value `(available == 0)` indicates that the lock is available, and a value of 1 indicates that the lock is unavailable. Using this `struct`, illustrate how the following functions can be implemented using the `compare_and_swap()` instruction:

- `void acquire(lock *mutex)`
- `void release(lock *mutex)`

Be sure to include any initialization that may be necessary.

```
while (true) {
   while (true) {
      flag[i] = want_in;
      j = turn;

      while (j != i) {
         if (flag[j] != idle) {
            j = turn;
         else
            j = (j + 1) % n;
      }

      flag[i] = in_cs;
      j = 0;

      while ( (j < n) && (j == i || flag[j] != in_cs))
         j++;

      if ( (j >= n) && (turn == i || flag[turn] == idle))
         break;
   }

      /* critical section */

   j = (turn + 1) % n;

   while (flag[j] == idle)
      j = (j + 1) % n;

   turn = j;
   flag[i] = idle;

      /* remainder section */
}
```

Figure 6.19 The structure of process P_i in Eisenberg and McGuire's algorithm.

6.17 Explain why interrupts are not appropriate for implementing synchronization primitives in multiprocessor systems.

6.18 The implementation of mutex locks provided in Section 6.5 suffers from busy waiting. Describe what changes would be necessary so that a process waiting to acquire a mutex lock would be blocked and placed into a waiting queue until the lock became available.

6.19 Assume that a system has multiple processing cores. For each of the following scenarios, describe which is a better locking mechanism—a

spinlock or a mutex lock where waiting processes sleep while waiting for the lock to become available:

- The lock is to be held for a short duration.
- The lock is to be held for a long duration.
- A thread may be put to sleep while holding the lock.

6.20 Assume that a context switch takes T time. Suggest an upper bound (in terms of T) for holding a spinlock. If the spinlock is held for any longer, a mutex lock (where waiting threads are put to sleep) is a better alternative.

6.21 A multithreaded web server wishes to keep track of the number of requests it services (known as ***hits***). Consider the two following strategies to prevent a race condition on the variable `hits`. The first strategy is to use a basic mutex lock when updating `hits`:

```
int hits;
mutex_lock hit_lock;

hit_lock.acquire();
hits++;
hit_lock.release();
```

A second strategy is to use an atomic integer:

```
atomic_t hits;
atomic_inc(&hits);
```

Explain which of these two strategies is more efficient.

6.22 Consider the code example for allocating and releasing processes shown in Figure 6.20.

a. Identify the race condition(s).

b. Assume you have a mutex lock named `mutex` with the operations `acquire()` and `release()`. Indicate where the locking needs to be placed to prevent the race condition(s).

c. Could we replace the integer variable

```
int number_of_processes = 0
```

with the atomic integer

```
atomic_t number_of_processes = 0
```

to prevent the race condition(s)?

6.23 Servers can be designed to limit the number of open connections. For example, a server may wish to have only N socket connections at any point in time. As soon as N connections are made, the server will not accept another incoming connection until an existing connection is

```
#define MAX_PROCESSES 255
int number_of_processes = 0;

/* the implementation of fork() calls this function */
int allocate_process() {
int new_pid;

   if (number_of_processes == MAX_PROCESSES)
      return -1;
   else {
      /* allocate necessary process resources */
      ++number_of_processes;

      return new_pid;
   }
}

/* the implementation of exit() calls this function */
void release_process() {
    /* release process resources */
    --number_of_processes;
}
```

Figure 6.20 Allocating and releasing processes for Exercise 6.22.

released. Illustrate how semaphores can be used by a server to limit the number of concurrent connections.

6.24 In Section 6.7, we use the following illustration as an incorrect use of semaphores to solve the critical-section problem:

```
wait(mutex);
   ...
   critical section
   ...
wait(mutex);
```

Explain why this is an example of a liveness failure.

6.25 Demonstrate that monitors and semaphores are equivalent to the degree that they can be used to implement solutions to the same types of synchronization problems.

6.26 Describe how the `signal()` operation associated with monitors differs from the corresponding operation defined for semaphores.

6.27 Suppose the `signal()` statement can appear only as the last statement in a monitor function. Suggest how the implementation described in Section 6.7 can be simplified in this situation.

6.28 Consider a system consisting of processes $P_1, P_2, ..., P_n$, each of which has a unique priority number. Write a monitor that allocates three identical printers to these processes, using the priority numbers for deciding the order of allocation.

6.29 A file is to be shared among different processes, each of which has a unique number. The file can be accessed simultaneously by several processes, subject to the following constraint: the sum of all unique numbers associated with all the processes currently accessing the file must be less than n. Write a monitor to coordinate access to the file.

6.30 When a signal is performed on a condition inside a monitor, the signaling process can either continue its execution or transfer control to the process that is signaled. How would the solution to the preceding exercise differ with these two different ways in which signaling can be performed?

6.31 Design an algorithm for a monitor that implements an alarm clock that enables a calling program to delay itself for a specified number of time units (*ticks*). You may assume the existence of a real hardware clock that invokes a function `tick()` in your monitor at regular intervals.

6.32 Discuss ways in which the priority inversion problem could be addressed in a real-time system. Also discuss whether the solutions could be implemented within the context of a proportional share scheduler.

Chapter 7 Exercises

7.7 Describe two kernel data structures in which race conditions are possible. Be sure to include a description of how a race condition can occur.

7.8 The Linux kernel has a policy that a process cannot hold a spinlock while attempting to acquire a semaphore. Explain why this policy is in place.

7.9 Design an algorithm for a bounded-buffer monitor in which the buffers (portions) are embedded within the monitor itself.

7.10 The strict mutual exclusion within a monitor makes the bounded-buffer monitor of Exercise 7.9 mainly suitable for small portions.

a. Explain why this is true.

b. Design a new scheme that is suitable for larger portions.

7.11 Discuss the tradeoff between fairness and throughput of operations in the readers–writers problem. Propose a method for solving the readers–writers problem without causing starvation.

7.12 Explain why the call to the `lock()` method in a Java `ReentrantLock` is not placed in the `try` clause for exception handling, yet the call to the `unlock()` method is placed in a `finally` clause.

7.13 Explain the difference between software and hardware transactional memory.

Chapter 8 Exercises

8.12 Consider the traffic deadlock depicted in Figure 8.12.

a. Show that the four necessary conditions for deadlock hold in this example.

b. State a simple rule for avoiding deadlocks in this system.

8.13 Draw the resource-allocation graph that illustrates deadlock from the program example shown in Figure 8.1 in Section 8.2.

8.14 In Section 6.8.1, we described a potential deadlock scenario involving processes P_0 and P_1 and semaphores `S` and `Q`. Draw the resource-allocation graph that illustrates deadlock under the scenario presented in that section.

8.15 Assume that a multithreaded application uses only reader–writer locks for synchronization. Applying the four necessary conditions for deadlock, is deadlock still possible if multiple reader–writer locks are used?

8.16 The program example shown in Figure 8.1 doesn't always lead to deadlock. Describe what role the CPU scheduler plays and how it can contribute to deadlock in this program.

8.17 In Section 8.5.4, we described a situation in which we prevent deadlock by ensuring that all locks are acquired in a certain order. However, we also point out that deadlock is possible in this situation if two threads simultaneously invoke the `transaction()` function. Fix the `transaction()` function to prevent deadlocks.

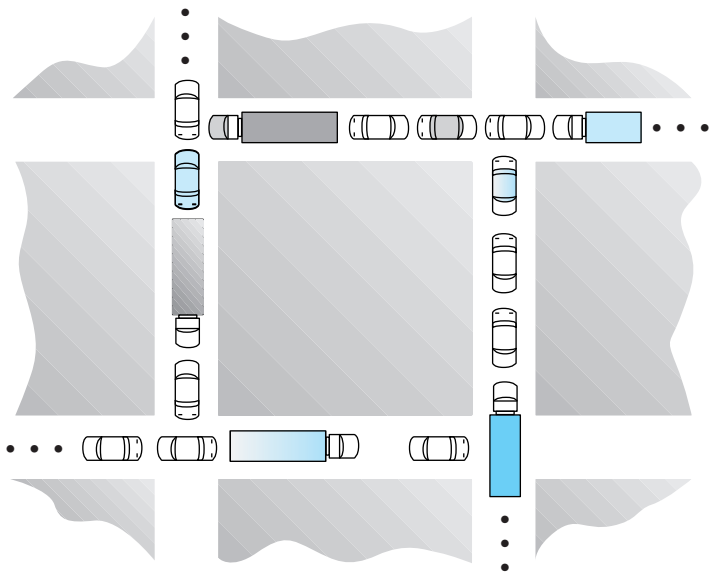


Figure 8.11 Traffic deadlock for Exercise 8.12.

8.18 Which of the six resource-allocation graphs shown in Figure 8.12 illustrate deadlock? For those situations that are deadlocked, provide the cycle of threads and resources. Where there is not a deadlock situation, illustrate the order in which the threads may complete execution.

8.19 Compare the circular-wait scheme with the various deadlock-avoidance schemes (like the banker's algorithm) with respect to the following issues:

a. Runtime overhead

b. System throughput

8.20 In a real computer system, neither the resources available nor the demands of threads for resources are consistent over long periods (months). Resources break or are replaced, new processes and threads come and go, and new resources are bought and added to the system. If deadlock is controlled by the banker's algorithm, which of the

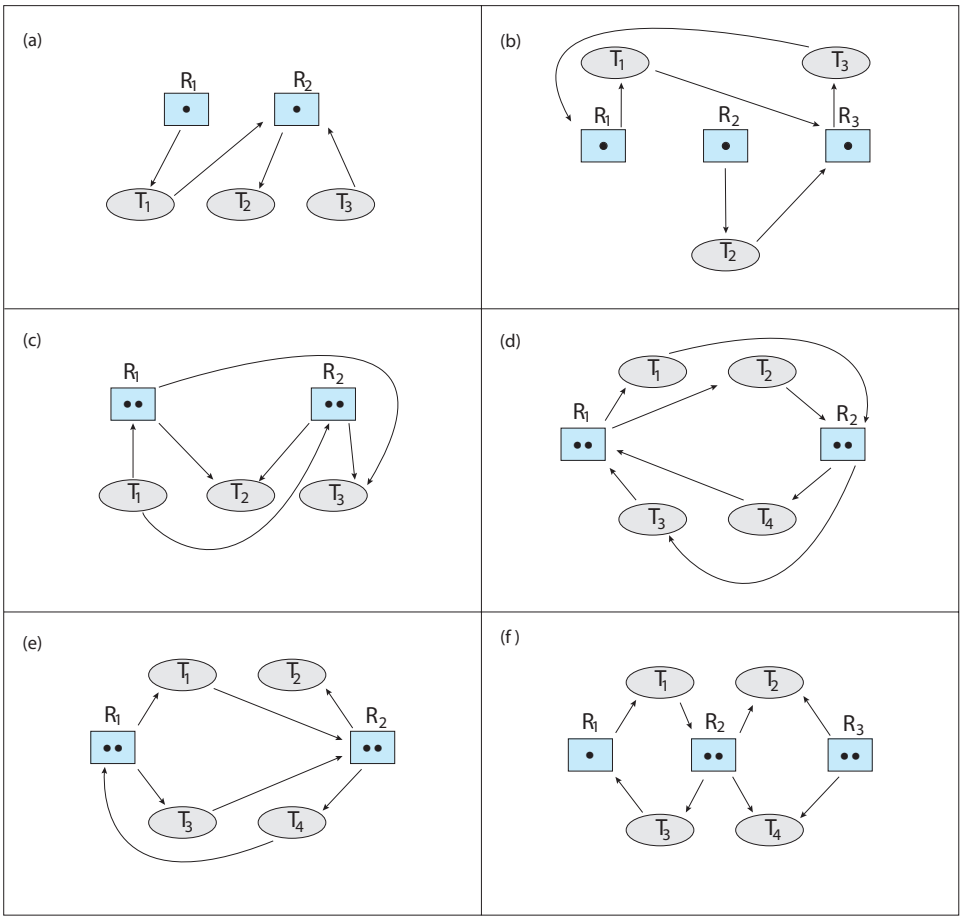


Figure 8.12 Resource-allocation graphs for Exercise 8.18.

following changes can be made safely (without introducing the possibility of deadlock), and under what circumstances?

a. Increase ***Available*** (new resources added).

b. Decrease ***Available*** (resource permanently removed from system).

c. Increase ***Max*** for one thread (the thread needs or wants more resources than allowed).

d. Decrease ***Max*** for one thread (the thread decides it does not need that many resources).

e. Increase the number of threads.

f. Decrease the number of threads.

8.21 Consider the following snapshot of a system:

	Allocation	***Max***
	A B C D	*A B C D*
T_0	2 1 0 6	6 3 2 7
T_1	3 3 1 3	5 4 1 5
T_2	2 3 1 2	6 6 1 4
T_3	1 2 3 4	4 3 4 5
T_4	3 0 3 0	7 2 6 1

What are the contents of the ***Need*** matrix?

8.22 Consider a system consisting of four resources of the same type that are shared by three threads, each of which needs at most two resources. Show that the system is deadlock free.

8.23 Consider a system consisting of m resources of the same type being shared by n threads. A thread can request or release only one resource at a time. Show that the system is deadlock free if the following two conditions hold:

a. The maximum need of each thread is between one resource and m resources.

b. The sum of all maximum needs is less than $m + n$.

8.24 Consider the version of the dining-philosophers problem in which the chopsticks are placed at the center of the table and any two of them can be used by a philosopher. Assume that requests for chopsticks are made one at a time. Describe a simple rule for determining whether a particular request can be satisfied without causing deadlock given the current allocation of chopsticks to philosophers.

8.25 Consider again the setting in the preceding exercise. Assume now that each philosopher requires three chopsticks to eat. Resource requests are still issued one at a time. Describe some simple rules for determining whether a particular request can be satisfied without causing deadlock given the current allocation of chopsticks to philosophers.

8.26 We can obtain the banker's algorithm for a single resource type from the general banker's algorithm simply by reducing the dimensionality of the various arrays by 1.

Show through an example that we cannot implement the multiple-resource-type banker's scheme by applying the single-resource-type scheme to each resource type individually.

8.27 Consider the following snapshot of a system:

	Allocation	***Max***
	A B C D	*A B C D*
T_0	1 2 0 2	4 3 1 6
T_1	0 1 1 2	2 4 2 4
T_2	1 2 4 0	3 6 5 1
T_3	1 2 0 1	2 6 2 3
T_4	1 0 0 1	3 1 1 2

Using the banker's algorithm, determine whether or not each of the following states is unsafe. If the state is safe, illustrate the order in which the threads may complete. Otherwise, illustrate why the state is unsafe.

a. ***Available*** = (2, 2, 2, 3)

b. ***Available*** = (4, 4, 1, 1)

c. ***Available*** = (3, 0, 1, 4)

d. ***Available*** = (1, 5, 2, 2)

8.28 Consider the following snapshot of a system:

	Allocation	***Max***	***Available***
	A B C D	*A B C D*	*A B C D*
T_0	3 1 4 1	6 4 7 3	2 2 2 4
T_1	2 1 0 2	4 2 3 2	
T_2	2 4 1 3	2 5 3 3	
T_3	4 1 1 0	6 3 3 2	
T_4	2 2 2 1	5 6 7 5	

Answer the following questions using the banker's algorithm:

a. Illustrate that the system is in a safe state by demonstrating an order in which the threads may complete.

b. If a request from thread T_4 arrives for (2, 2, 2, 4), can the request be granted immediately?

c. If a request from thread T_2 arrives for (0, 1, 1, 0), can the request be granted immediately?

d. If a request from thread T_3 arrives for (2, 2, 1, 2), can the request be granted immediately?

8.29 What is the optimistic assumption made in the deadlock-detection algorithm? How can this assumption be violated?

8.30 A single-lane bridge connects the two Vermont villages of North Tunbridge and South Tunbridge. Farmers in the two villages use this bridge to deliver their produce to the neighboring town. The bridge can become deadlocked if a northbound and a southbound farmer get on the bridge at the same time. (Vermont farmers are stubborn and are unable to back up.) Using semaphores and/or mutex locks, design an algorithm in pseudocode that prevents deadlock. Initially, do not be concerned about starvation (the situation in which northbound farmers prevent southbound farmers from using the bridge, or vice versa).

8.31 Modify your solution to Exercise 8.30 so that it is starvation-free.

Chapter 9 Exercises

9.11 Explain the difference between internal and external fragmentation.

9.12 Consider the following process for generating binaries. A compiler is used to generate the object code for individual modules, and a linker is used to combine multiple object modules into a single program binary. How does the linker change the binding of instructions and data to memory addresses? What information needs to be passed from the compiler to the linker to facilitate the memory-binding tasks of the linker?

9.13 Given six memory partitions of 100 MB, 170 MB, 40 MB, 205 MB, 300 MB, and 185 MB (in order), how would the first-fit, best-fit, and worst-fit algorithms place processes of size 200 MB, 15 MB, 185 MB, 75 MB, 175 MB, and 80 MB (in order)? Indicate which—if any—requests cannot be satisfied. Comment on how efficiently each of the algorithms manages memory.

9.14 Most systems allow a program to allocate more memory to its address space during execution. Allocation of data in the heap segments of programs is an example of such allocated memory. What is required to support dynamic memory allocation in the following schemes?

a. Contiguous memory allocation

b. Paging

9.15 Compare the memory organization schemes of contiguous memory allocation and paging with respect to the following issues:

a. External fragmentation

b. Internal fragmentation

c. Ability to share code across processes

9.16 On a system with paging, a process cannot access memory that it does not own. Why? How could the operating system allow access to additional memory? Why should it or should it not?

9.17 Explain why mobile operating systems such as iOS and Android do not support swapping.

9.18 Although Android does not support swapping on its boot disk, it is possible to set up a swap space using a separate SD nonvolatile memory card. Why would Android disallow swapping on its boot disk yet allow it on a secondary disk?

9.19 Explain why address-space identifiers (ASIDs) are used in TLBs.

9.20 Program binaries in many systems are typically structured as follows. Code is stored starting with a small, fixed virtual address, such as 0. The code segment is followed by the data segment, which is used for storing the program variables. When the program starts executing, the stack is allocated at the other end of the virtual address space and is allowed to grow toward lower virtual addresses. What is the significance of this structure for the following schemes?

a. Contiguous memory allocation

b. Paging

9.21 Assuming a 1-KB page size, what are the page numbers and offsets for the following address references (provided as decimal numbers)?

a. 21205

b. 164250

c. 121357

d. 16479315

e. 27253187

9.22 The MPV operating system is designed for embedded systems and has a 24-bit virtual address, a 20-bit physical address, and a 4-KB page size. How many entries are there in each of the following?

a. A conventional, single-level page table

b. An inverted page table

What is the maximum amount of physical memory in the MPV operating system?

9.23 Consider a logical address space of 2,048 pages with a 4-KB page size, mapped onto a physical memory of 512 frames.

a. How many bits are required in the logical address?

b. How many bits are required in the physical address?

9.24 Consider a computer system with a 32-bit logical address and 8-KB page size. The system supports up to 1 GB of physical memory. How many entries are there in each of the following?

a. A conventional, single-level page table

b. An inverted page table

9.25 Consider a paging system with the page table stored in memory.

a. If a memory reference takes 50 nanoseconds, how long does a paged memory reference take?

b. If we add TLBs, and if 75 percent of all page-table references are found in the TLBs, what is the effective memory reference time? (Assume that finding a page-table entry in the TLBs takes 2 nanoseconds, if the entry is present.)

9.26 What is the purpose of paging the page tables?

9.27 Consider the IA-32 address-translation scheme shown in Figure 9.22.

a. Describe all the steps taken by the IA-32 in translating a logical address into a physical address.

b. What are the advantages to the operating system of hardware that provides such complicated memory translation?

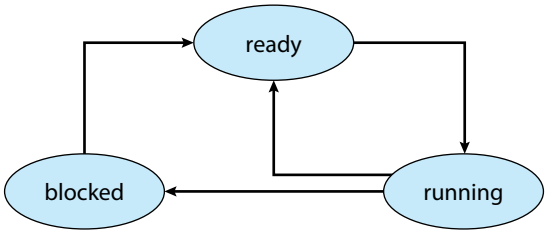
c. Are there any disadvantages to this address-translation system? If so, what are they? If not, why is this scheme not used by every manufacturer?

Chapter 10 Exercises

10.15 Assume that a program has just referenced an address in virtual memory. Describe a scenario in which each of the following can occur. (If no such scenario can occur, explain why.)

- TLB miss with no page fault
- TLB miss with page fault
- TLB hit with no page fault
- TLB hit with page fault

10.16 A simplified view of thread states is ***ready***, ***running***, and ***blocked***, where a thread is either ready and waiting to be scheduled, is running on the processor, or is blocked (for example, waiting for I/O).



Assuming a thread is in the ***running*** state, answer the following questions, and explain your answers:

a. Will the thread change state if it incurs a page fault? If so, to what state will it change?

b. Will the thread change state if it generates a TLB miss that is resolved in the page table? If so, to what state will it change?

c. Will the thread change state if an address reference is resolved in the page table? If so, to what state will it change?

10.17 Consider a system that uses pure demand paging.

a. When a process first starts execution, how would you characterize the page-fault rate?

b. Once the working set for a process is loaded into memory, how would you characterize the page-fault rate?

c. Assume that a process changes its locality and the size of the new working set is too large to be stored in available free memory. Identify some options system designers could choose from to handle this situation.

10.18 The following is a page table for a system with 12-bit virtual and physical addresses and 256-byte pages. Free page frames are to be allocated in the order `9, F, D`. A dash for a page frame indicates that the page is not in memory.

Page	Page Frame
0	0 x 4
1	0 x B
2	0 x A
3	–
4	–
5	0 x 2
6	–
7	0 x 0
8	0 x C
9	0 x 1

Convert the following virtual addresses to their equivalent physical addresses in hexadecimal. All numbers are given in hexadecimal. In the case of a page fault, you must use one of the free frames to update the page table and resolve the logical address to its corresponding physical address.

- `0x2A1`
- `0x4E6`
- `0x94A`
- `0x316`

10.19 What is the copy-on-write feature, and under what circumstances is its use beneficial? What hardware support is required to implement this feature?

10.20 A certain computer provides its users with a virtual memory space of 2^{32} bytes. The computer has 2^{22} bytes of physical memory. The virtual memory is implemented by paging, and the page size is 4,096 bytes. A user process generates the virtual address 11123456. Explain how the system establishes the corresponding physical location. Distinguish between software and hardware operations.

10.21 Assume that we have a demand-paged memory. The page table is held in registers. It takes 8 milliseconds to service a page fault if an empty frame is available or if the replaced page is not modified and 20 milliseconds if the replaced page is modified. Memory-access time is 100 nanoseconds.

Assume that the page to be replaced is modified 70 percent of the time. What is the maximum acceptable page-fault rate for an effective access time of no more than 200 nanoseconds?

10.22 Consider the page table for a system with 16-bit virtual and physical addresses and 4,096-byte pages.

Page	Page Frame	Reference Bit
0	9	0
1	–	0
2	10	0
3	15	0
4	6	0
5	13	0
6	8	0
7	12	0
8	7	0
9	–	0
10	5	0
11	4	0
12	1	0
13	0	0
14	–	0
15	2	0

The reference bit for a page is set to 1 when the page has been referenced. Periodically, a thread zeroes out all values of the reference bit. A dash for a page frame indicates that the page is not in memory. The page-replacement algorithm is localized LRU, and all numbers are provided in decimal.

a. Convert the following virtual addresses (in hexadecimal) to the equivalent physical addresses. You may provide answers in either hexadecimal or decimal. Also set the reference bit for the appropriate entry in the page table.
   - `0x621C`
   - `0xF0A3`
   - `0xBC1A`
   - `0x5BAA`
   - `0x0BA1`

b. Using the above addresses as a guide, provide an example of a logical address (in hexadecimal) that results in a page fault.

c. From what set of page frames will the LRU page-replacement algorithm choose in resolving a page fault?

10.23 When a page fault occurs, the process requesting the page must block while waiting for the page to be brought from disk into physical memory. Assume that there exists a process with five user-level threads and that the mapping of user threads to kernel threads is many to one. If

one user thread incurs a page fault while accessing its stack, would the other user threads belonging to the same process also be affected by the page fault—that is, would they also have to wait for the faulting page to be brought into memory? Explain.

10.24 Apply the (1) FIFO, (2) LRU, and (3) optimal (OPT) replacement algorithms for the following page-reference strings:

- 2, 6, 9, 2, 4, 2, 1, 7, 3, 0, 5, 2, 1, 2, 9, 5, 7, 3, 8, 5
- 0, 6, 3, 0, 2, 6, 3, 5, 2, 4, 1, 3, 0, 6, 1, 4, 2, 3, 5, 7
- 3, 1, 4, 2, 5, 4, 1, 3, 5, 2, 0, 1, 1, 0, 2, 3, 4, 5, 0, 1
- 4, 2, 1, 7, 9, 8, 3, 5, 2, 6, 8, 1, 0, 7, 2, 4, 1, 3, 5, 8
- 0, 1, 2, 3, 4, 4, 3, 2, 1, 0, 0, 1, 2, 3, 4, 4, 3, 2, 1, 0

Indicate the number of page faults for each algorithm assuming demand paging with three frames.

10.25 Assume that you are monitoring the rate at which the pointer in the clock algorithm moves. (The pointer indicates the candidate page for replacement.) What can you say about the system if you notice the following behavior:

a. Pointer is moving fast.

b. Pointer is moving slow.

10.26 Discuss situations in which the least frequently used (LFU) page-replacement algorithm generates fewer page faults than the least recently used (LRU) page-replacement algorithm. Also discuss under what circumstances the opposite holds.

10.27 Discuss situations in which the most frequently used (MFU) page-replacement algorithm generates fewer page faults than the least recently used (LRU) page-replacement algorithm. Also discuss under what circumstances the opposite holds.

10.28 The KHIE (pronounced "k-hi") operating system uses a FIFO replacement algorithm for resident pages and a free-frame pool of recently used pages. Assume that the free-frame pool is managed using the LRU replacement policy. Answer the following questions:

a. If a page fault occurs and the page does not exist in the free-frame pool, how is free space generated for the newly requested page?

b. If a page fault occurs and the page exists in the free-frame pool, how are the resident page set and the free-frame pool managed to make space for the requested page?

c. To what does the system degenerate if the number of resident pages is set to one?

d. To what does the system degenerate if the number of pages in the free-frame pool is zero?

10.29 Consider a demand-paging system with the following time-measured utilizations:

CPU utilization	20%
Paging disk	97.7%
Other I/O devices	5%

For each of the following, indicate whether it will (or is likely to) improve CPU utilization. Explain your answers.

a. Install a faster CPU.

b. Install a bigger paging disk.

c. Increase the degree of multiprogramming.

d. Decrease the degree of multiprogramming.

e. Install more main memory.

f. Install a faster hard disk or multiple controllers with multiple hard disks.

g. Add prepaging to the page-fetch algorithms.

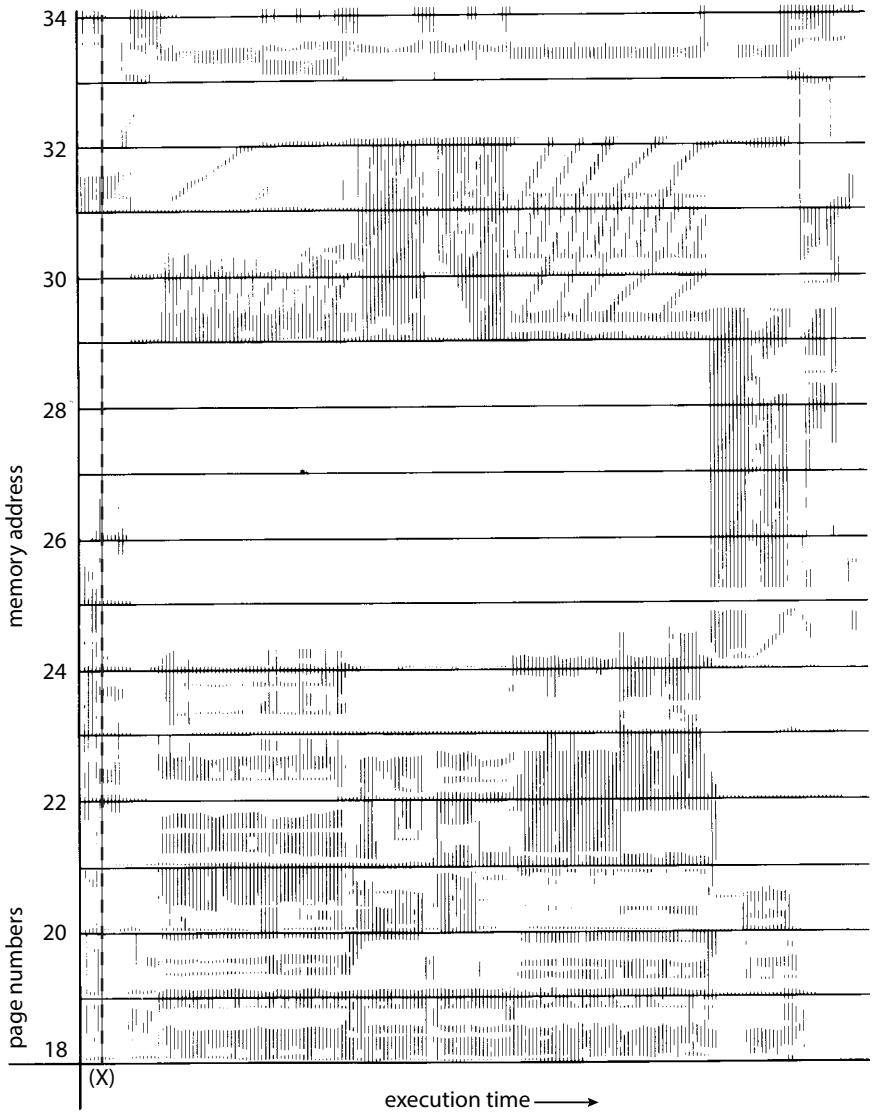
h. Increase the page size.

10.30 Explain why minor page faults take less time to resolve than major page faults.

10.31 Explain why compressed memory is used in operating systems for mobile devices.

10.32 Suppose that a machine provides instructions that can access memory locations using the one-level indirect addressing scheme. What sequence of page faults is incurred when all of the pages of a program are currently nonresident and the first instruction of the program is an indirect memory-load operation? What happens when the operating system is using a per-process frame allocation technique and only two pages are allocated to this process?

10.33 Consider the page references:



What pages represent the locality at time (*X*)?

10.34 Suppose that your replacement policy (in a paged system) is to examine each page regularly and to discard that page if it has not been used since the last examination. What would you gain and what would you lose by using this policy rather than LRU or second-chance replacement?

10.35 A page-replacement algorithm should minimize the number of page faults. We can achieve this minimization by distributing heavily used pages evenly over all of memory, rather than having them compete for a small number of page frames. We can associate with each page frame a counter of the number of pages associated with that frame. Then,

to replace a page, we can search for the page frame with the smallest counter.

a. Define a page-replacement algorithm using this basic idea. Specifically address these problems:

- What is the initial value of the counters?
- When are counters increased?
- When are counters decreased?
- How is the page to be replaced selected?

b. How many page faults occur for your algorithm for the following reference string with four page frames?

1, 2, 3, 4, 5, 3, 4, 1, 6, 7, 8, 7, 8, 9, 7, 8, 9, 5, 4, 5, 4, 2.

c. What is the minimum number of page faults for an optimal page-replacement strategy for the reference string in part b with four page frames?

10.36 Consider a demand-paging system with a paging disk that has an average access and transfer time of 20 milliseconds. Addresses are translated through a page table in main memory, with an access time of 1 microsecond per memory access. Thus, each memory reference through the page table takes two accesses. To improve this time, we have added an associative memory that reduces access time to one memory reference if the page-table entry is in the associative memory.

Assume that 80 percent of the accesses are in the associative memory and that, of those remaining, 10 percent (or 2 percent of the total) cause page faults. What is the effective memory access time?

10.37 What is the cause of thrashing? How does the system detect thrashing? Once it detects thrashing, what can the system do to eliminate this problem?

10.38 Is it possible for a process to have two working sets, one representing data and another representing code? Explain.

10.39 Consider the parameter Δ used to define the working-set window in the working-set model. When Δ is set to a low value, what is the effect on the page-fault frequency and the number of active (nonsuspended) processes currently executing in the system? What is the effect when Δ is set to a very high value?

10.40 In a 1,024-KB segment, memory is allocated using the buddy system. Using Figure 10.26 as a guide, draw a tree illustrating how the following memory requests are allocated:

- Request 5-KB
- Request 135 KB.
- Request 14 KB.
- Request 3 KB.

- Request 12 KB.

Next, modify the tree for the following releases of memory. Perform coalescing whenever possible:

- Release 3 KB.
- Release 5 KB.
- Release 14 KB.
- Release 12 KB.

10.41 A system provides support for user-level and kernel-level threads. The mapping in this system is one to one (there is a corresponding kernel thread for each user thread). Does a multithreaded process consist of (a) a working set for the entire process or (b) a working set for each thread? Explain

10.42 The slab-allocation algorithm uses a separate cache for each different object type. Assuming there is one cache per object type, explain why this scheme doesn't scale well with multiple CPUs. What could be done to address this scalability issue?

10.43 Consider a system that allocates pages of different sizes to its processes. What are the advantages of such a paging scheme? What modifications to the virtual memory system would be needed to provide this functionality?

Chapter 11 Exercises

11.11 None of the disk-scheduling disciplines, except FCFS, is truly fair (starvation may occur).

a. Explain why this assertion is true.

b. Describe a way to modify algorithms such as SCAN to ensure fairness.

c. Explain why fairness is an important goal in a multi-user systems.

d. Give three or more examples of circumstances in which it is important that the operating system be unfair in serving I/O requests.

11.12 Explain why NVM devices often use an FCFS disk-scheduling algorithm.

11.13 Suppose that a disk drive has 5,000 cylinders, numbered 0 to 4,999. The drive is currently serving a request at cylinder 2,150, and the previous request was at cylinder 1,805. The queue of pending requests, in FIFO order, is:

2,069; 1,212; 2,296; 2,800; 544; 1,618; 356; 1,523; 4,965; 3,681

Starting from the current head position, what is the total distance (in cylinders) that the disk arm moves to satisfy all the pending requests for each of the following disk-scheduling algorithms?

a. FCFS

b. SCAN

c. C-SCAN

11.14 Elementary physics states that when an object is subjected to a constant acceleration a, the relationship between distance d and time t is given by $d = \frac{1}{2}at^2$. Suppose that, during a seek, the disk in Exercise 11.13 accelerates the disk arm at a constant rate for the first half of the seek, then decelerates the disk arm at the same rate for the second half of the seek. Assume that the disk can perform a seek to an adjacent cylinder in 1 millisecond and a full-stroke seek over all 5,000 cylinders in 18 milliseconds.

a. The distance of a seek is the number of cylinders over which the head moves. Explain why the seek time is proportional to the square root of the seek distance.

b. Write an equation for the seek time as a function of the seek distance. This equation should be of the form $t = x + y\sqrt{L}$, where t is the time in milliseconds and L is the seek distance in cylinders.

c. Calculate the total seek time for each of the schedules in Exercise 11.13. Determine which schedule is the fastest (has the smallest total seek time).

d. The **percentage speedup** is the time saved divided by the original time. What is the percentage speedup of the fastest schedule over FCFS?

11.15 Suppose that the disk in Exercise 11.14 rotates at 7,200 RPM.

a. What is the average rotational latency of this disk drive?

b. What seek distance can be covered in the time that you found for part a?

11.16 Compare and contrast HDDs and NVM devices. What are the best applications for each type?

11.17 Describe some advantages and disadvantages of using NVM devices as a caching tier and as a disk-drive replacement compared with using only HDDs.

11.18 Compare the performance of C-SCAN and SCAN scheduling, assuming a uniform distribution of requests. Consider the average response time (the time between the arrival of a request and the completion of that request's service), the variation in response time, and the effective bandwidth. How does performance depend on the relative sizes of seek time and rotational latency?

11.19 Requests are not usually uniformly distributed. For example, we can expect a cylinder containing the file-system metadata to be accessed more frequently than a cylinder containing only files. Suppose you know that 50 percent of the requests are for a small, fixed number of cylinders.

a. Would any of the scheduling algorithms discussed in this chapter be particularly good for this case? Explain your answer.

b. Propose a disk-scheduling algorithm that gives even better performance by taking advantage of this "hot spot" on the disk.

11.20 Consider a RAID level 5 organization comprising five disks, with the parity for sets of four blocks on four disks stored on the fifth disk. How many blocks are accessed in order to perform the following?

a. A write of one block of data

b. A write of seven continuous blocks of data

11.21 Compare the throughput achieved by a RAID level 5 organization with that achieved by a RAID level 1 organization for the following:

a. Read operations on single blocks

b. Read operations on multiple contiguous blocks

11.22 Compare the performance of write operations achieved by a RAID level 5 organization with that achieved by a RAID level 1 organization.

11.23 Assume that you have a mixed configuration comprising disks organized as RAID level 1 and RAID level 5 disks. Assume that the system has flexibility in deciding which disk organization to use for storing a

particular file. Which files should be stored in the RAID level 1 disks and which in the RAID level 5 disks in order to optimize performance?

11.24 The reliability of a storage device is typically described in terms of mean time between failures (MTBF). Although this quantity is called a "time," the MTBF actually is measured in drive-hours per failure.

a. If a system contains 1,000 disk drives, each of which has a 750,000-hour MTBF, which of the following best describes how often a drive failure will occur in that disk farm: once per thousand years, once per century, once per decade, once per year, once per month, once per week, once per day, once per hour, once per minute, or once per second?

b. Mortality statistics indicate that, on the average, a U.S. resident has about 1 chance in 1,000 of dying between the ages of 20 and 21. Deduce the MTBF hours for 20-year-olds. Convert this figure from hours to years. What does this MTBF tell you about the expected lifetime of a 20-year-old?

c. The manufacturer guarantees a 1-million-hour MTBF for a certain model of disk drive. What can you conclude about the number of years for which one of these drives is under warranty?

11.25 Discuss the relative advantages and disadvantages of sector sparing and sector slipping.

11.26 Discuss the reasons why the operating system might require accurate information on how blocks are stored on a disk. How could the operating system improve file-system performance with this knowledge?

Chapter 12 Exercises

12.8 When multiple interrupts from different devices appear at about the same time, a priority scheme could be used to determine the order in which the interrupts would be serviced. Discuss what issues need to be considered in assigning priorities to different interrupts.

12.9 What are the advantages and disadvantages of supporting memory-mapped I/O to device-control registers?

12.10 Consider the following I/O scenarios on a single-user PC:

a. A mouse used with a graphical user interface

b. A tape drive on a multitasking operating system (with no device preallocation available)

c. A disk drive containing user files

d. A graphics card with direct bus connection, accessible through memory-mapped I/O

For each of these scenarios, would you design the operating system to use buffering, spooling, caching, or a combination? Would you use polled I/O or interrupt-driven I/O? Give reasons for your choices.

12.11 In most multiprogrammed systems, user programs access memory through virtual addresses, while the operating system uses raw physical addresses to access memory. What are the implications of this design for the initiation of I/O operations by the user program and their execution by the operating system?

12.12 What are the various kinds of performance overhead associated with servicing an interrupt?

12.13 Describe three circumstances under which blocking I/O should be used. Describe three circumstances under which nonblocking I/O should be used. Why not just implement nonblocking I/O and have processes busy-wait until their devices are ready?

12.14 Typically, at the completion of a device I/O, a single interrupt is raised and appropriately handled by the host processor. In certain settings, however, the code that is to be executed at the completion of the I/O can be broken into two separate pieces. The first piece executes immediately after the I/O completes and schedules a second interrupt for the remaining piece of code to be executed at a later time. What is the purpose of using this strategy in the design of interrupt handlers?

12.15 Some DMA controllers support direct virtual memory access, where the targets of I/O operations are specified as virtual addresses and a translation from virtual to physical address is performed during the DMA. How does this design complicate the design of the DMA controller? What are the advantages of providing such functionality?

12.16 UNIX coordinates the activities of the kernel I/O components by manipulating shared in-kernel data structures, whereas Windows uses object-

oriented message passing between kernel I/O components. Discuss three pros and three cons of each approach.

12.17 Write (in pseudocode) an implementation of virtual clocks, including the queueing and management of timer requests for the kernel and applications. Assume that the hardware provides three timer channels.

12.18 Discuss the advantages and disadvantages of guaranteeing reliable transfer of data between modules in the STREAMS abstraction.

Chapter 13 Exercises

13.9 Consider a file system in which a file can be deleted and its disk space reclaimed while links to that file still exist. What problems may occur if a new file is created in the same storage area or with the same absolute path name? How can these problems be avoided?

13.10 The open-file table is used to maintain information about files that are currently open. Should the operating system maintain a separate table for each user or maintain just one table that contains references to files that are currently being accessed by all users? If the same file is being accessed by two different programs or users, should there be separate entries in the open-file table? Explain.

13.11 What are the advantages and disadvantages of providing mandatory locks instead of advisory locks whose use is left to users' discretion?

13.12 Provide examples of applications that typically access files according to the following methods:

- Sequential
- Random

13.13 Some systems automatically open a file when it is referenced for the first time and close the file when the job terminates. Discuss the advantages and disadvantages of this scheme compared with the more traditional one, where the user has to open and close the file explicitly.

13.14 If the operating system knew that a certain application was going to access file data in a sequential manner, how could it exploit this information to improve performance?

13.15 Give an example of an application that could benefit from operating-system support for random access to indexed files.

13.16 Some systems provide file sharing by maintaining a single copy of a file. Other systems maintain several copies, one for each of the users sharing the file. Discuss the relative merits of each approach.

Chapter 14 Exercises

14.7 Consider a file system that uses a modified contiguous-allocation scheme with support for extents. A file is a collection of extents, with each extent corresponding to a contiguous set of blocks. A key issue in such systems is the degree of variability in the size of the extents. What are the advantages and disadvantages of the following schemes?

a. All extents are of the same size, and the size is predetermined.

b. Extents can be of any size and are allocated dynamically.

c. Extents can be of a few fixed sizes, and these sizes are predetermined.

14.8 Contrast the performance of the three techniques for allocating disk blocks (contiguous, linked, and indexed) for both sequential and random file access.

14.9 What are the advantages of the variant of linked allocation that uses a FAT to chain together the blocks of a file?

14.10 Consider a system where free space is kept in a free-space list.

a. Suppose that the pointer to the free-space list is lost. Can the system reconstruct the free-space list? Explain your answer.

b. Consider a file system similar to the one used by UNIX with indexed allocation. How many disk I/O operations might be required to read the contents of a small local file at `/a/b/c`? Assume that none of the disk blocks is currently being cached.

c. Suggest a scheme to ensure that the pointer is never lost as a result of memory failure.

14.11 Some file systems allow disk storage to be allocated at different levels of granularity. For instance, a file system could allocate 4 KB of disk space as a single 4-KB block or as eight 512-byte blocks. How could we take advantage of this flexibility to improve performance? What modifications would have to be made to the free-space management scheme in order to support this feature?

14.12 Discuss how performance optimizations for file systems might result in difficulties in maintaining the consistency of the systems in the event of computer crashes.

14.13 Discuss the advantages and disadvantages of supporting links to files that cross mount points (that is, the file link refers to a file that is stored in a different volume).

14.14 Consider a file system on a disk that has both logical and physical block sizes of 512 bytes. Assume that the information about each file is already in memory. For each of the three allocation strategies (contiguous, linked, and indexed), answer these questions:

a. How is the logical-to-physical address mapping accomplished in this system? (For the indexed allocation, assume that a file is always less than 512 blocks long.)

b. If we are currently at logical block 10 (the last block accessed was block 10) and want to access logical block 4, how many physical blocks must be read from the disk?

14.15 Consider a file system that uses inodes to represent files. Disk blocks are 8 KB in size, and a pointer to a disk block requires 4 bytes. This file system has 12 direct disk blocks, as well as single, double, and triple indirect disk blocks. What is the maximum size of a file that can be stored in this file system?

14.16 Fragmentation on a storage device can be eliminated through compaction. Typical disk devices do not have relocation or base registers (such as those used when memory is to be compacted), so how can we relocate files? Give three reasons why compacting and relocating files are often avoided.

14.17 Explain why logging metadata updates ensures recovery of a file system after a file-system crash.

14.18 Consider the following backup scheme:

- **Day 1**. Copy to a backup medium all files from the disk.
- **Day 2**. Copy to another medium all files changed since day 1.
- **Day 3**. Copy to another medium all files changed since day 1.

This differs from the schedule given in Section 14.7.4 by having all subsequent backups copy all files modified since the first full backup. What are the benefits of this system over the one in Section 14.7.4? What are the drawbacks? Are restore operations made easier or more difficult? Explain your answer.

14.19 Discuss the advantages and disadvantages of associating with remote file systems (stored on file servers) a set of failure semantics different from those associated with local file systems.

14.20 What are the implications of supporting UNIX consistency semantics for shared access to files stored on remote file systems?

Chapter 15 Exercises

15.6 Assume that in a particular augmentation of a remote-file-access protocol, each client maintains a name cache that caches translations from file names to corresponding file handles. What issues should we take into account in implementing the name cache?

15.7 Given a mounted file system with write operations underway, and a system crash or power loss, what must be done before the file system is remounted if: (a) The file system is not log-structured? (b) The file system is log-structured?

15.8 Why do operating systems mount the root file system automatically at boot time?

15.9 Why do operating systems require file systems other than root to be mounted?

Chapter 16 Exercises

16.1 Buffer-overflow attacks can be avoided by adopting a better programming methodology or by using special hardware support. Discuss these solutions.

16.2 A password may become known to other users in a variety of ways. Is there a simple method for detecting that such an event has occurred? Explain your answer.

16.3 What is the purpose of using a "salt" along with a user-provided password? Where should the salt be stored, and how should it be used?

16.4 The list of all passwords is kept in the operating system. Thus, if a user manages to read this list, password protection is no longer provided. Suggest a scheme that will avoid this problem. (Hint: Use different internal and external representations.)

16.5 An experimental addition to UNIX allows a user to connect a **watchdog** program to a file. The watchdog is invoked whenever a program requests access to the file. The watchdog then either grants or denies access to the file. Discuss two pros and two cons of using watchdogs for security.

16.6 Discuss a means by which managers of systems connected to the Internet could design their systems to limit or eliminate the damage done by worms. What are the drawbacks of making the change that you suggest?

16.7 Make a list of six security concerns for a bank's computer system. For each item on your list, state whether this concern relates to physical, human, or operating-system security.

16.8 What are two advantages of encrypting data stored in the computer system?

16.9 What commonly used computer programs are prone to man-in-the-middle attacks? Discuss solutions for preventing this form of attack.

16.10 Compare symmetric and asymmetric encryption schemes, and discuss the circumstances under which a distributed system would use one or the other.

16.11 Why doesn't $D_{kd,N}(E_{ke,N}(m))$ provide authentication of the sender? To what uses can such an encryption be put?

16.12 Discuss how the asymmetric encryption algorithm can be used to achieve the following goals.

a. Authentication: the receiver knows that only the sender could have generated the message.

b. Secrecy: only the receiver can decrypt the message.

c. Authentication and secrecy: only the receiver can decrypt the message, and the receiver knows that only the sender could have generated the message.

16.13 Consider a system that generates 10 million audit records per day. Assume that, on average, there are 10 attacks per day on this system and each attack is reflected in 20 records. If the intrusion-detection system has a true-alarm rate of 0.6 and a false-alarm rate of 0.0005, what percentage of alarms generated by the system corresponds to real intrusions?

16.14 Mobile operating systems such as iOS and Android place the user data and the system files into two separate partitions. Aside from security, what is an advantage of that separation?

Chapter 17 Exercises

17.11 The access-control matrix can be used to determine whether a process can switch from, say, domain A to domain B and enjoy the access privileges of domain B. Is this approach equivalent to including the access privileges of domain B in those of domain A?

17.12 Consider a computer system in which computer games can be played by students only between 10 P.M. and 6 A.M., by faculty members between 5 P.M. and 8 A.M., and by the computer center staff at all times. Suggest a scheme for implementing this policy efficiently.

17.13 What hardware features does a computer system need for efficient capability manipulation? Can these features be used for memory protection?

17.14 Discuss the strengths and weaknesses of implementing an access matrix using access lists that are associated with objects.

17.15 Discuss the strengths and weaknesses of implementing an access matrix using capabilities that are associated with domains.

17.16 Explain why a capability-based system provides greater flexibility than a ring-protection scheme in enforcing protection policies.

17.17 What is the need-to-know principle? Why is it important for a protection system to adhere to this principle?

17.18 Discuss which of the following systems allow module designers to enforce the need-to-know principle.

a. Ring-protection scheme

b. JVM's stack-inspection scheme

17.19 Describe how the Java protection model would be compromised if a Java program were allowed to directly alter the annotations of its stack frame.

17.20 How are the access-matrix facility and the role-based access-control facility similar? How do they differ?

17.21 How does the principle of least privilege aid in the creation of protection systems?

17.22 How can systems that implement the principle of least privilege still have protection failures that lead to security violations?

Chapter 18 Exercises

18.1 Describe the three types of traditional hypervisors.

18.2 Describe four virtualization-like execution environments, and explain how they differ from "true" virtualization.

18.3 Describe four benefits of virtualization.

18.4 Why are VMMs unable to implement trap-and-emulate-based virtualization on some CPUs? Lacking the ability to trap and emulate, what method can a VMM use to implement virtualization?

18.5 What hardware assistance for virtualization can be provided by modern CPUs?

18.6 Why is live migration possible in virtual environments but much less possible for a native operating system?

Chapter 19 Exercises

19.7 What is the difference between computation migration and process migration? Which is easier to implement, and why?

19.8 Even though the OSI model of networking specifies seven layers of functionality, most computer systems use fewer layers to implement a network. Why do they use fewer layers? What problems could the use of fewer layers cause?

19.9 Explain why doubling the speed of the systems on an Ethernet segment may result in decreased network performance when the UDP transport protocol is used. What changes could help solve this problem?

19.10 What are the advantages of using dedicated hardware devices for routers? What are the disadvantages of using these devices compared with using general-purpose computers?

19.11 In what ways is using a name server better than using static host tables? What problems or complications are associated with name servers? What methods could you use to decrease the amount of traffic name servers generate to satisfy translation requests?

19.12 Name servers are organized in a hierarchical manner. What is the purpose of using a hierarchical organization?

19.13 The lower layers of the OSI network model provide datagram service, with no delivery guarantees for messages. A transport-layer protocol such as TCP is used to provide reliability. Discuss the advantages and disadvantages of supporting reliable message delivery at the lowest possible layer.

19.14 Run the program shown in Figure 19.4 and determine the IP addresses of the following host names:

- www.wiley.com
- www.cs.yale.edu
- www.apple.com
- www.westminstercollege.edu
- www.ietf.org

19.15 A DNS name can map to multiple servers, such as www.google.com. However, if we run the program shown in Figure 19.4, we get only one IP address. Modify the program to display all the server IP addresses instead of just one.

19.16 The original HTTP protocol used TCP/IP as the underlying network protocol. For each page, graphic, or applet, a separate TCP session was constructed, used, and torn down. Because of the overhead of building and destroying TCP/IP connections, performance problems resulted from this implementation method. Would using UDP rather than TCP be a good alternative? What other changes could you make to improve HTTP performance?

19.17 What are the advantages and the disadvantages of making the computer network transparent to the user?

19.18 What are the benefits of a DFS compared with a file system in a centralized system?

19.19 For each of the following workloads, identify whether a cluster-based or a client–server DFS model would handle the workload best. Explain your answers.

- Hosting student files in a university lab.
- Processing data sent by the Hubble telescope.
- Sharing data with multiple devices from a home server.

19.20 Discuss whether OpenAFS and NFS provide the following: (a) location transparency and (b) location independence.

19.21 Under what circumstances would a client prefer a location-transparent DFS? Under what circumstances would she prefer a location-independent DFS? Discuss the reasons for these preferences.

19.22 What aspects of a distributed system would you select for a system running on a totally reliable network?

19.23 Compare and contrast the techniques of caching disk blocks locally, on a client system, and remotely, on a server.

19.24 Which scheme would likely result in a greater space saving on a multiuser DFS: file-level deduplication or block-level deduplication? Explain your answer.

19.25 What types of extra metadata information would need to be stored in a DFS that uses deduplication?

Chapter 20 Exercises

20.7 What are the advantages and disadvantages of writing an operating system in a high-level language, such as C?

20.8 In what circumstances is the system-call sequence `fork()` `exec()` most appropriate? When is `vfork()` preferable?

20.9 What socket type should be used to implement an intercomputer file-transfer program? What type should be used for a program that periodically tests to see whether another computer is up on the network? Explain your answer.

20.10 Linux runs on a variety of hardware platforms. What steps must Linux developers take to ensure that the system is portable to different processors and memory-management architectures and to minimize the amount of architecture-specific kernel code?

20.11 What are the advantages and disadvantages of making only some of the symbols defined inside a kernel accessible to a loadable kernel module?

20.12 What are the primary goals of the conflict-resolution mechanism used by the Linux kernel for loading kernel modules?

20.13 Discuss how the clone() operation supported by Linux is used to support both processes and threads.

20.14 Would you classify Linux threads as user-level threads or as kernel-level threads? Support your answer with the appropriate arguments.

20.15 What extra costs are incurred in the creation and scheduling of a process, compared with the cost of a cloned thread?

20.16 How does Linux's Completely Fair Scheduler (CFS) provide improved fairness over a traditional UNIX process scheduler? When is the fairness guaranteed?

20.17 What are the two configurable variables of the Completely Fair Scheduler (CFS)? What are the pros and cons of setting each of them to very small and very large values?

20.18 The Linux scheduler implements "soft" real-time scheduling. What features necessary for certain real-time programming tasks are missing? How might they be added to the kernel? What are the costs (downsides) of such features?

20.19 Under what circumstances would a user process request an operation that results in the allocation of a demand-zero memory region?

20.20 What scenarios would cause a page of memory to be mapped into a user program's address space with the copy-on-write attribute enabled?

20.21 In Linux, shared libraries perform many operations central to the operating system. What is the advantage of keeping this functionality out of the kernel? Are there any drawbacks? Explain your answer.

20.22 What are the benefits of a journaling file system such as Linux's ext3? What are the costs? Why does ext3 provide the option to journal only metadata?

20.23 The directory structure of a Linux operating system could include files corresponding to several different file systems, including the Linux `/proc` file system. How might the need to support different file-system types affect the structure of the Linux kernel?

20.24 In what ways does the Linux `setuid` feature differ from the `setuid` feature SVR4?

20.25 The Linux source code is freely and widely available over the Internet and from CD-ROM vendors. What are three implications of this availability for the security of the Linux system?

Chapter 21 Exercises

21.14 Under what circumstances would one use the deferred procedure calls facility in Windows?

21.15 What is a handle, and how does a process obtain a handle?

21.16 Describe the management scheme of the virtual memory manager. How does the VM manager improve performance?

21.17 Describe a useful application of the no-access page facility provided in Windows.

21.18 Describe the three techniques used for communicating data in a local procedure call. What settings are most conducive to the application of the different message-passing techniques?

21.19 What manages caching in Windows? How is the cache managed?

21.20 How does the NTFS directory structure differ from the directory structure used in UNIX operating systems?

21.21 What is a process, and how is it managed in Windows?

21.22 What is the fiber abstraction provided by Windows? How does it differ from the thread abstraction?

21.23 How does user-mode scheduling (UMS) in Windows 7 differ from fibers? What are some trade-offs between fibers and UMS?

21.24 UMS considers a thread to have two parts, a UT and a KT. How might it be useful to allow UTs to continue executing in parallel with their KTs?

21.25 What is the performance trade-off of allowing KTs and UTs to execute on different processors?

21.26 Why does the self-map occupy large amounts of virtual address space but no additional virtual memory?

21.27 How does the self-map make it easy for the VM manager to move the page-table pages to and from disk? Where are the page-table pages kept on disk?

21.28 When a Windows system hibernates, the system is powered off. Suppose you changed the CPU or the amount of RAM on a hibernating system. Do you think that would work? Why or why not?

21.29 Give an example showing how the use of a suspend count is helpful in suspending and resuming threads in Windows.