



Part Seven

Case Studies

In the final part of the book, we integrate the concepts described earlier by examining real operating systems. We cover two such systems in detail—Linux and Windows 7. We chose Linux for several reasons: it is popular, it is freely available, and it represents a full-featured UNIX system. This gives a student of operating systems an opportunity to read—and modify—**real** operating-system source code.

We also cover Windows 7 in detail. This recent operating system from Microsoft is gaining popularity not only in the standalone-machine market but also in the workgroup-server market. We chose Windows 7 because it provides an opportunity to study a modern operating system that has a design and implementation drastically different from those of UNIX.

In addition, we briefly discuss other highly influential operating systems. Finally, we provide on-line coverage of two more systems: FreeBSD and Mach. The FreeBSD system is another UNIX system. However, whereas Linux combines features from several UNIX systems, FreeBSD is based on the BSD model. FreeBSD source code, like Linux source code, is freely available. Mach is a modern operating system that provides compatibility with BSD UNIX.

The Linux System



Updated by Robert Love

This chapter presents an in-depth examination of the Linux operating system. By examining a complete, real system, we can see how the concepts we have discussed relate both to one another and to practice.

Linux is a variant of UNIX that has gained popularity over the last several decades, powering devices as small as mobile phones and as large as room-filling supercomputers. In this chapter, we look at the history and development of Linux and cover the user and programmer interfaces that Linux presents—interfaces that owe a great deal to the UNIX tradition. We also discuss the design and implementation of these interfaces. Linux is a rapidly evolving operating system. This chapter describes developments through the Linux 3.2 kernel, which was released in 2012.

CHAPTER OBJECTIVES

- To explore the history of the UNIX operating system from which Linux is derived and the principles upon which Linux's design is based.
- To examine the Linux process model and illustrate how Linux schedules processes and provides interprocess communication.
- To look at memory management in Linux.
- To explore how Linux implements file systems and manages I/O devices.

18.1 Linux History

Linux looks and feels much like any other UNIX system; indeed, UNIX compatibility has been a major design goal of the Linux project. However, Linux is much younger than most UNIX systems. Its development began in 1991, when a Finnish university student, Linus Torvalds, began developing a small but self-contained kernel for the 80386 processor, the first true 32-bit processor in Intel's range of PC-compatible CPUs.

Early in its development, the Linux source code was made available free—both at no cost and with minimal distributional restrictions—on the Internet. As a result, Linux’s history has been one of collaboration by many developers from all around the world, corresponding almost exclusively over the Internet. From an initial kernel that partially implemented a small subset of the UNIX system services, the Linux system has grown to include all of the functionality expected of a modern UNIX system.

In its early days, Linux development revolved largely around the central operating-system kernel—the core, privileged executive that manages all system resources and interacts directly with the computer hardware. We need much more than this kernel, of course, to produce a full operating system. We thus need to make a distinction between the Linux kernel and a complete Linux system. The **Linux kernel** is an original piece of software developed from scratch by the Linux community. The **Linux system**, as we know it today, includes a multitude of components, some written from scratch, others borrowed from other development projects, and still others created in collaboration with other teams.

The basic Linux system is a standard environment for applications and user programming, but it does not enforce any standard means of managing the available functionality as a whole. As Linux has matured, a need has arisen for another layer of functionality on top of the Linux system. This need has been met by various Linux distributions. A **Linux distribution** includes all the standard components of the Linux system, plus a set of administrative tools to simplify the initial installation and subsequent upgrading of Linux and to manage installation and removal of other packages on the system. A modern distribution also typically includes tools for management of file systems, creation and management of user accounts, administration of networks, Web browsers, word processors, and so on.

18.1.1 The Linux Kernel

The first Linux kernel released to the public was version 0.01, dated May 14, 1991. It had no networking, ran only on 80386-compatible Intel processors and PC hardware, and had extremely limited device-driver support. The virtual memory subsystem was also fairly basic and included no support for memory-mapped files; however, even this early incarnation supported shared pages with copy-on-write and protected address spaces. The only file system supported was the Minix file system, as the first Linux kernels were cross-developed on a Minix platform.

The next milestone, Linux 1.0, was released on March 14, 1994. This release culminated three years of rapid development of the Linux kernel. Perhaps the single biggest new feature was networking: 1.0 included support for UNIX’s standard TCP/IP networking protocols, as well as a BSD-compatible socket interface for networking programming. Device-driver support was added for running IP over Ethernet or (via the PPP or SLIP protocols) over serial lines or modems.

The 1.0 kernel also included a new, much enhanced file system without the limitations of the original Minix file system, and it supported a range of SCSI controllers for high-performance disk access. The developers extended the virtual memory subsystem to support paging to swap files and memory mapping

of arbitrary files (but only read-only memory mapping was implemented in 1.0).

A range of extra hardware support was included in this release. Although still restricted to the Intel PC platform, hardware support had grown to include floppy-disk and CD-ROM devices, as well as sound cards, a range of mice, and international keyboards. Floating-point emulation was provided in the kernel for 80386 users who had no 80387 math coprocessor. System V UNIX-style **interprocess communication (IPC)**, including shared memory, semaphores, and message queues, was implemented.

At this point, development started on the 1.1 kernel stream, but numerous bug-fix patches were released subsequently for 1.0. A pattern was adopted as the standard numbering convention for Linux kernels. Kernels with an odd minor-version number, such as 1.1 or 2.5, are **development kernels**; even-numbered minor-version numbers are stable **production kernels**. Updates for the stable kernels are intended only as remedial versions, whereas the development kernels may include newer and relatively untested functionality. As we will see, this pattern remained in effect until version 3.

In March 1995, the 1.2 kernel was released. This release did not offer nearly the same improvement in functionality as the 1.0 release, but it did support a much wider variety of hardware, including the new PCI hardware bus architecture. Developers added another PC-specific feature—support for the 80386 CPU's virtual 8086 mode—to allow emulation of the DOS operating system for PC computers. They also updated the IP implementation with support for accounting and firewalling. Simple support for dynamically loadable and unloadable kernel modules was supplied as well.

The 1.2 kernel was the final PC-only Linux kernel. The source distribution for Linux 1.2 included partially implemented support for SPARC, Alpha, and MIPS CPUs, but full integration of these other architectures did not begin until after the 1.2 stable kernel was released.

The Linux 1.2 release concentrated on wider hardware support and more complete implementations of existing functionality. Much new functionality was under development at the time, but integration of the new code into the main kernel source code was deferred until after the stable 1.2 kernel was released. As a result, the 1.3 development stream saw a great deal of new functionality added to the kernel.

This work was released in June 1996 as Linux version 2.0. This release was given a major version-number increment because of two major new capabilities: support for multiple architectures, including a 64-bit native Alpha port, and symmetric multiprocessing (SMP) support. Additionally, the memory-management code was substantially improved to provide a unified cache for file-system data independent of the caching of block devices. As a result of this change, the kernel offered greatly increased file-system and virtual-memory performance. For the first time, file-system caching was extended to networked file systems, and writable memory-mapped regions were also supported. Other major improvements included the addition of internal kernel threads, a mechanism exposing dependencies between loadable modules, support for the automatic loading of modules on demand, file-system quotas, and POSIX-compatible real-time process-scheduling classes.

Improvements continued with the release of Linux 2.2 in 1999. A port to UltraSPARC systems was added. Networking was enhanced with more flexible firewalling, improved routing and traffic management, and support for TCP large window and selective acknowledgement. Acorn, Apple, and NT disks could now be read, and NFS was enhanced with a new kernel-mode NFS daemon. Signal handling, interrupts, and some I/O were locked at a finer level than before to improve symmetric multiprocessor (SMP) performance.

Advances in the 2.4 and 2.6 releases of the kernel included increased support for SMP systems, journaling file systems, and enhancements to the memory-management and block I/O systems. The process scheduler was modified in version 2.6, providing an efficient $O(1)$ scheduling algorithm. In addition, the 2.6 kernel was preemptive, allowing a process to be preempted even while running in kernel mode.

Linux kernel version 3.0 was released in July 2011. The major version bump from 2 to 3 occurred to commemorate the twentieth anniversary of Linux. New features include improved virtualization support, a new page write-back facility, improvements to the memory-management system, and yet another new process scheduler—the Completely Fair Scheduler (CFS). We focus on this newest kernel in the remainder of this chapter.

18.1.2 The Linux System

As we noted earlier, the Linux kernel forms the core of the Linux project, but other components make up a complete Linux operating system. Whereas the Linux kernel is composed entirely of code written from scratch specifically for the Linux project, much of the supporting software that makes up the Linux system is not exclusive to Linux but is common to a number of UNIX-like operating systems. In particular, Linux uses many tools developed as part of Berkeley’s BSD operating system, MIT’s X Window System, and the Free Software Foundation’s GNU project.

This sharing of tools has worked in both directions. The main system libraries of Linux were originated by the GNU project, but the Linux community greatly improved the libraries by addressing omissions, inefficiencies, and bugs. Other components, such as the **GNU C compiler (gcc)**, were already of sufficiently high quality to be used directly in Linux. The network administration tools under Linux were derived from code first developed for 4.3 BSD, but more recent BSD derivatives, such as FreeBSD, have borrowed code from Linux in return. Examples of this sharing include the Intel floating-point-emulation math library and the PC sound-hardware device drivers.

The Linux system as a whole is maintained by a loose network of developers collaborating over the Internet, with small groups or individuals having responsibility for maintaining the integrity of specific components. A small number of public Internet file-transfer-protocol (FTP) archive sites act as de facto standard repositories for these components. The **File System Hierarchy Standard** document is also maintained by the Linux community as a means of ensuring compatibility across the various system components. This standard specifies the overall layout of a standard Linux file system; it determines under which directory names configuration files, libraries, system binaries, and run-time data files should be stored.

18.1.3 Linux Distributions

In theory, anybody can install a Linux system by fetching the latest revisions of the necessary system components from the FTP sites and compiling them. In Linux's early days, this is precisely what a Linux user had to do. As Linux has matured, however, various individuals and groups have attempted to make this job less painful by providing standard, precompiled sets of packages for easy installation.

These collections, or distributions, include much more than just the basic Linux system. They typically include extra system-installation and management utilities, as well as precompiled and ready-to-install packages of many of the common UNIX tools, such as news servers, web browsers, text-processing and editing tools, and even games.

The first distributions managed these packages by simply providing a means of unpacking all the files into the appropriate places. One of the important contributions of modern distributions, however, is advanced package management. Today's Linux distributions include a package-tracking database that allows packages to be installed, upgraded, or removed painlessly.

The SLS distribution, dating back to the early days of Linux, was the first collection of Linux packages that was recognizable as a complete distribution. Although it could be installed as a single entity, SLS lacked the package-management tools now expected of Linux distributions. The **Slackware** distribution represented a great improvement in overall quality, even though it also had poor package management. In fact, it is still one of the most widely installed distributions in the Linux community.

Since Slackware's release, many commercial and noncommercial Linux distributions have become available. **Red Hat** and **Debian** are particularly popular distributions; the first comes from a commercial Linux support company and the second from the free-software Linux community. Other commercially supported versions of Linux include distributions from **Canonical** and **SuSE**, and others too numerous to list here. There are too many Linux distributions in circulation for us to list all of them here. The variety of distributions does not prevent Linux distributions from being compatible, however. The RPM package file format is used, or at least understood, by the majority of distributions, and commercial applications distributed in this format can be installed and run on any distribution that can accept RPM files.

18.1.4 Linux Licensing

The Linux kernel is distributed under version 2.0 of the GNU General Public License (GPL), the terms of which are set out by the Free Software Foundation. Linux is not public-domain software. **Public domain** implies that the authors have waived copyright rights in the software, but copyright rights in Linux code are still held by the code's various authors. Linux is *free* software, however, in the sense that people can copy it, modify it, use it in any manner they want, and give away (or sell) their own copies.

The main implication of Linux's licensing terms is that nobody using Linux, or creating a derivative of Linux (a legitimate exercise), can distribute the derivative without including the source code. Software released under the GPL cannot be redistributed as a binary-only product. If you release software that includes any components covered by the GPL, then, under the GPL, you must

make source code available alongside any binary distributions. (This restriction does not prohibit making—or even selling—binary software distributions, as long as anybody who receives binaries is also given the opportunity to get the originating source code for a reasonable distribution charge.)

18.2 Design Principles

In its overall design, Linux resembles other traditional, nonmicrokernel UNIX implementations. It is a multiuser, preemptively multitasking system with a full set of UNIX-compatible tools. Linux's file system adheres to traditional UNIX semantics, and the standard UNIX networking model is fully implemented. The internal details of Linux's design have been influenced heavily by the history of this operating system's development.

Although Linux runs on a wide variety of platforms, it was originally developed exclusively on PC architecture. A great deal of that early development was carried out by individual enthusiasts rather than by well-funded development or research facilities, so from the start Linux attempted to squeeze as much functionality as possible from limited resources. Today, Linux can run happily on a multiprocessor machine with many gigabytes of main memory and many terabytes of disk space, but it is still capable of operating usefully in under 16 MB of RAM.

As PCs became more powerful and as memory and hard disks became cheaper, the original, minimalist Linux kernels grew to implement more UNIX functionality. Speed and efficiency are still important design goals, but much recent and current work on Linux has concentrated on a third major design goal: standardization. One of the prices paid for the diversity of UNIX implementations currently available is that source code written for one may not necessarily compile or run correctly on another. Even when the same system calls are present on two different UNIX systems, they do not necessarily behave in exactly the same way. The POSIX standards comprise a set of specifications for different aspects of operating-system behavior. There are POSIX documents for common operating-system functionality and for extensions such as process threads and real-time operations. Linux is designed to comply with the relevant POSIX documents, and at least two Linux distributions have achieved official POSIX certification.

Because it gives standard interfaces to both the programmer and the user, Linux presents few surprises to anybody familiar with UNIX. We do not detail these interfaces here. The sections on the programmer interface (Section A.3) and user interface (Section A.4) of BSD apply equally well to Linux. By default, however, the Linux programming interface adheres to SVR4 UNIX semantics, rather than to BSD behavior. A separate set of libraries is available to implement BSD semantics in places where the two behaviors differ significantly.

Many other standards exist in the UNIX world, but full certification of Linux with respect to these standards is sometimes slowed because certification is often available only for a fee, and the expense involved in certifying an operating system's compliance with most standards is substantial. However, supporting a wide base of applications is important for any operating system, so implementation of standards is a major goal for Linux development, even if the implementation is not formally certified. In addition to the basic POSIX

standard, Linux currently supports the POSIX threading extensions—Pthreads—and a subset of the POSIX extensions for real-time process control.

18.2.1 Components of a Linux System

The Linux system is composed of three main bodies of code, in line with most traditional UNIX implementations:

- 1. **Kernel.** The kernel is responsible for maintaining all the important abstractions of the operating system, including such things as virtual memory and processes.
- 2. **System libraries.** The system libraries define a standard set of functions through which applications can interact with the kernel. These functions implement much of the operating-system functionality that does not need the full privileges of kernel code. The most important system library is the **C library**, known as `libc`. In addition to providing the standard C library, `libc` implements the user mode side of the Linux system call interface, as well as other critical system-level interfaces.
- 3. **System utilities.** The system utilities are programs that perform individual, specialized management tasks. Some system utilities are invoked just once to initialize and configure some aspect of the system. Others—known as **daemons** in UNIX terminology—run permanently, handling such tasks as responding to incoming network connections, accepting logon requests from terminals, and updating log files.

Figure 18.1 illustrates the various components that make up a full Linux system. The most important distinction here is between the kernel and everything else. All the kernel code executes in the processor’s privileged mode with full access to all the physical resources of the computer. Linux refers to this privileged mode as **kernel mode**. Under Linux, no user code is built into the kernel. Any operating-system-support code that does not need to run in kernel mode is placed into the system libraries and runs in **user mode**. Unlike kernel mode, user mode has access only to a controlled subset of the system’s resources.

system-management programs	user processes	user utility programs	compilers
system shared libraries			
Linux kernel			
loadable kernel modules			

Figure 18.1 Components of the Linux system.

Although various modern operating systems have adopted a message-passing architecture for their kernel internals, Linux retains UNIX's historical model: the kernel is created as a single, monolithic binary. The main reason is performance. Because all kernel code and data structures are kept in a single address space, no context switches are necessary when a process calls an operating-system function or when a hardware interrupt is delivered. Moreover, the kernel can pass data and make requests between various subsystems using relatively cheap C function invocation and not more complicated inter-process communication (IPC). This single address space contains not only the core scheduling and virtual memory code but all kernel code, including all device drivers, file systems, and networking code.

Even though all the kernel components share this same melting pot, there is still room for modularity. In the same way that user applications can load shared libraries at run time to pull in a needed piece of code, so the Linux kernel can load (and unload) modules dynamically at run time. The kernel does not need to know in advance which modules may be loaded—they are truly independent loadable components.

The Linux kernel forms the core of the Linux operating system. It provides all the functionality necessary to run processes, and it provides system services to give arbitrated and protected access to hardware resources. The kernel implements all the features required to qualify as an operating system. On its own, however, the operating system provided by the Linux kernel is not a complete UNIX system. It lacks much of the functionality and behavior of UNIX, and the features that it does provide are not necessarily in the format in which a UNIX application expects them to appear. The operating-system interface visible to running applications is not maintained directly by the kernel. Rather, applications make calls to the system libraries, which in turn call the operating-system services as necessary.

The system libraries provide many types of functionality. At the simplest level, they allow applications to make system calls to the Linux kernel. Making a system call involves transferring control from unprivileged user mode to privileged kernel mode; the details of this transfer vary from architecture to architecture. The libraries take care of collecting the system-call arguments and, if necessary, arranging those arguments in the special form necessary to make the system call.

The libraries may also provide more complex versions of the basic system calls. For example, the C language's buffered file-handling functions are all implemented in the system libraries, providing more advanced control of file I/O than the basic kernel system calls. The libraries also provide routines that do not correspond to system calls at all, such as sorting algorithms, mathematical functions, and string-manipulation routines. All the functions necessary to support the running of UNIX or POSIX applications are implemented in the system libraries.

The Linux system includes a wide variety of user-mode programs—both system utilities and user utilities. The system utilities include all the programs necessary to initialize and then administer the system, such as those to set up networking interfaces and to add and remove users from the system. User utilities are also necessary to the basic operation of the system but do not require elevated privileges to run. They include simple file-management utilities such as those to copy files, create directories, and edit text files. One

of the most important user utilities is the [shell](#), the standard command-line interface on UNIX systems. Linux supports many shells; the most common is the [bourne-Again shell \(bash\)](#).

18.3 Kernel Modules

The Linux kernel has the ability to load and unload arbitrary sections of kernel code on demand. These loadable kernel modules run in privileged kernel mode and as a consequence have full access to all the hardware capabilities of the machine on which they run. In theory, there is no restriction on what a kernel module is allowed to do. Among other things, a kernel module can implement a device driver, a file system, or a networking protocol.

Kernel modules are convenient for several reasons. Linux's source code is free, so anybody wanting to write kernel code is able to compile a modified kernel and to reboot into that new functionality. However, recompiling, relinking, and reloading the entire kernel is a cumbersome cycle to undertake when you are developing a new driver. If you use kernel modules, you do not have to make a new kernel to test a new driver—the driver can be compiled on its own and loaded into the already running kernel. Of course, once a new driver is written, it can be distributed as a module so that other users can benefit from it without having to rebuild their kernels.

This latter point has another implication. Because it is covered by the GPL license, the Linux kernel cannot be released with proprietary components added to it unless those new components are also released under the GPL and the source code for them is made available on demand. The kernel's module interface allows third parties to write and distribute, on their own terms, device drivers or file systems that could not be distributed under the GPL.

Kernel modules allow a Linux system to be set up with a standard minimal kernel, without any extra device drivers built in. Any device drivers that the user needs can be either loaded explicitly by the system at startup or loaded automatically by the system on demand and unloaded when not in use. For example, a mouse driver can be loaded when a USB mouse is plugged into the system and unloaded when the mouse is unplugged.

The module support under Linux has four components:

1. The [module-management system](#) allows modules to be loaded into memory and to communicate with the rest of the kernel.
2. The [module loader and unloader](#), which are user-mode utilities, work with the module-management system to load a module into memory.
3. The [driver-registration system](#) allows modules to tell the rest of the kernel that a new driver has become available.
4. A [conflict-resolution mechanism](#) allows different device drivers to reserve hardware resources and to protect those resources from accidental use by another driver.

18.3.1 Module Management

Loading a module requires more than just loading its binary contents into kernel memory. The system must also make sure that any references the

module makes to kernel symbols or entry points are updated to point to the correct locations in the kernel's address space. Linux deals with this reference updating by splitting the job of module loading into two separate sections: the management of sections of module code in kernel memory and the handling of symbols that modules are allowed to reference.

Linux maintains an internal symbol table in the kernel. This symbol table does not contain the full set of symbols defined in the kernel during the latter's compilation; rather, a symbol must be explicitly exported. The set of exported symbols constitutes a well-defined interface by which a module can interact with the kernel.

Although exporting symbols from a kernel function requires an explicit request by the programmer, no special effort is needed to import those symbols into a module. A module writer just uses the standard external linking of the C language. Any external symbols referenced by the module but not declared by it are simply marked as unresolved in the final module binary produced by the compiler. When a module is to be loaded into the kernel, a system utility first scans the module for these unresolved references. All symbols that still need to be resolved are looked up in the kernel's symbol table, and the correct addresses of those symbols in the currently running kernel are substituted into the module's code. Only then is the module passed to the kernel for loading. If the system utility cannot resolve all references in the module by looking them up in the kernel's symbol table, then the module is rejected.

The loading of the module is performed in two stages. First, the module-loader utility asks the kernel to reserve a continuous area of virtual kernel memory for the module. The kernel returns the address of the memory allocated, and the loader utility can use this address to relocate the module's machine code to the correct loading address. A second system call then passes the module, plus any symbol table that the new module wants to export, to the kernel. The module itself is now copied verbatim into the previously allocated space, and the kernel's symbol table is updated with the new symbols for possible use by other modules not yet loaded.

The final module-management component is the module requester. The kernel defines a communication interface to which a module-management program can connect. With this connection established, the kernel will inform the management process whenever a process requests a device driver, file system, or network service that is not currently loaded and will give the manager the opportunity to load that service. The original service request will complete once the module is loaded. The manager process regularly queries the kernel to see whether a dynamically loaded module is still in use and unloads that module when it is no longer actively needed.

18.3.2 Driver Registration

Once a module is loaded, it remains no more than an isolated region of memory until it lets the rest of the kernel know what new functionality it provides. The kernel maintains dynamic tables of all known drivers and provides a set of routines to allow drivers to be added to or removed from these tables at any time. The kernel makes sure that it calls a module's startup routine when that module is loaded and calls the module's cleanup routine before

that module is unloaded. These routines are responsible for registering the module's functionality.

A module may register many types of functionality; it is not limited to only one type. For example, a device driver might want to register two separate mechanisms for accessing the device. Registration tables include, among others, the following items:

- **Device drivers.** These drivers include character devices (such as printers, terminals, and mice), block devices (including all disk drives), and network interface devices.
- **File systems.** The file system may be anything that implements Linux's virtual file system calling routines. It might implement a format for storing files on a disk, but it might equally well be a network file system, such as NFS, or a virtual file system whose contents are generated on demand, such as Linux's /proc file system.
- **Network protocols.** A module may implement an entire networking protocol, such as TCP or simply a new set of packet-filtering rules for a network firewall.
- **Binary format.** This format specifies a way of recognizing, loading, and executing a new type of executable file.

In addition, a module can register a new set of entries in the `sysctl` and `/proc` tables, to allow that module to be configured dynamically (Section 18.7.4).

18.3.3 Conflict Resolution

Commercial UNIX implementations are usually sold to run on a vendor's own hardware. One advantage of a single-supplier solution is that the software vendor has a good idea about what hardware configurations are possible. PC hardware, however, comes in a vast number of configurations, with large numbers of possible drivers for devices such as network cards and video display adapters. The problem of managing the hardware configuration becomes more severe when modular device drivers are supported, since the currently active set of devices becomes dynamically variable.

Linux provides a central conflict-resolution mechanism to help arbitrate access to certain hardware resources. Its aims are as follows:

- To prevent modules from clashing over access to hardware resources
- To prevent **autoprobes**—device-driver probes that auto-detect device configuration—from interfering with existing device drivers
- To resolve conflicts among multiple drivers trying to access the same hardware—as, for example, when both the parallel printer driver and the parallel line IP (PLIP) network driver try to talk to the parallel port

To these ends, the kernel maintains lists of allocated hardware resources. The PC has a limited number of possible I/O ports (addresses in its hardware I/O address space), interrupt lines, and DMA channels. When any device driver wants to access such a resource, it is expected to reserve the resource with

the kernel database first. This requirement incidentally allows the system administrator to determine exactly which resources have been allocated by which driver at any given point.

A module is expected to use this mechanism to reserve in advance any hardware resources that it expects to use. If the reservation is rejected because the resource is not present or is already in use, then it is up to the module to decide how to proceed. It may fail in its initialization attempt and request that it be unloaded if it cannot continue, or it may carry on, using alternative hardware resources.

18.4 Process Management

A process is the basic context in which all user-requested activity is serviced within the operating system. To be compatible with other UNIX systems, Linux must use a process model similar to those of other versions of UNIX. Linux operates differently from UNIX in a few key places, however. In this section, we review the traditional UNIX process model (Section A.3.2) and introduce Linux's threading model.

18.4.1 The `fork()` and `exec()` Process Model

The basic principle of UNIX process management is to separate into two steps two operations that are usually combined into one: the creation of a new process and the running of a new program. A new process is created by the `fork()` system call, and a new program is run after a call to `exec()`. These are two distinctly separate functions. We can create a new process with `fork()` without running a new program—the new subprocess simply continues to execute exactly the same program, at exactly the same point, that the first (parent) process was running. In the same way, running a new program does not require that a new process be created first. Any process may call `exec()` at any time. A new binary object is loaded into the process's address space and the new executable starts executing in the context of the existing process.

This model has the advantage of great simplicity. It is not necessary to specify every detail of the environment of a new program in the system call that runs that program. The new program simply runs in its existing environment. If a parent process wishes to modify the environment in which a new program is to be run, it can fork and then, still running the original executable in a child process, make any system calls it requires to modify that child process before finally executing the new program.

Under UNIX, then, a process encompasses all the information that the operating system must maintain to track the context of a single execution of a single program. Under Linux, we can break down this context into a number of specific sections. Broadly, process properties fall into three groups: the process identity, environment, and context.

18.4.1.1 Process Identity

A process identity consists mainly of the following items:

- **Process ID (PID).** Each process has a unique identifier. The PID is used to specify the process to the operating system when an application makes a

system call to signal, modify, or wait for the process. Additional identifiers associate the process with a process group (typically, a tree of processes forked by a single user command) and login session.

- **Credentials.** Each process must have an associated user ID and one or more group IDs (user groups are discussed in Section 11.6.2) that determine the rights of a process to access system resources and files.
- **Personality.** Process personalities are not traditionally found on UNIX systems, but under Linux each process has an associated personality identifier that can slightly modify the semantics of certain system calls. Personalities are primarily used by emulation libraries to request that system calls be compatible with certain varieties of UNIX.
- **Namespace.** Each process is associated with a specific view of the file-system hierarchy, called its **namespace**. Most processes share a common namespace and thus operate on a shared file-system hierarchy. Processes and their children can, however, have different namespaces, each with a unique file-system hierarchy—their own root directory and set of mounted file systems.

Most of these identifiers are under the limited control of the process itself. The process group and session identifiers can be changed if the process wants to start a new group or session. Its credentials can be changed, subject to appropriate security checks. However, the primary PID of a process is unchangeable and uniquely identifies that process until termination.

18.4.1.2 Process Environment

A process's environment is inherited from its parent and is composed of two null-terminated vectors: the argument vector and the environment vector. The **argument vector** simply lists the command-line arguments used to invoke the running program; it conventionally starts with the name of the program itself. The **environment vector** is a list of "NAME=VALUE" pairs that associates named environment variables with arbitrary textual values. The environment is not held in kernel memory but is stored in the process's own user-mode address space as the first datum at the top of the process's stack.

The argument and environment vectors are not altered when a new process is created. The new child process will inherit the environment of its parent. However, a completely new environment is set up when a new program is invoked. On calling `exec()`, a process must supply the environment for the new program. The kernel passes these environment variables to the next program, replacing the process's current environment. The kernel otherwise leaves the environment and command-line vectors alone—their interpretation is left entirely to the user-mode libraries and applications.

The passing of environment variables from one process to the next and the inheriting of these variables by the children of a process provide flexible ways to pass information to components of the user-mode system software. Various important environment variables have conventional meanings to related parts of the system software. For example, the `TERM` variable is set up to name the type of terminal connected to a user's login session. Many programs use this

variable to determine how to perform operations on the user's display, such as moving the cursor and scrolling a region of text. Programs with multilingual support use the `LANG` variable to determine the language in which to display system messages for programs that include multilingual support.

The environment-variable mechanism custom-tailors the operating system on a per-process basis. Users can choose their own languages or select their own editors independently of one another.

18.4.1.3 Process Context

The process identity and environment properties are usually set up when a process is created and not changed until that process exits. A process may choose to change some aspects of its identity if it needs to do so, or it may alter its environment. In contrast, process context is the state of the running program at any one time; it changes constantly. Process context includes the following parts:

- **Scheduling context.** The most important part of the process context is its scheduling context—the information that the scheduler needs to suspend and restart the process. This information includes saved copies of all the process's registers. Floating-point registers are stored separately and are restored only when needed. Thus, processes that do not use floating-point arithmetic do not incur the overhead of saving that state. The scheduling context also includes information about scheduling priority and about any outstanding signals waiting to be delivered to the process. A key part of the scheduling context is the process's kernel stack, a separate area of kernel memory reserved for use by kernel-mode code. Both system calls and interrupts that occur while the process is executing will use this stack.
- **Accounting.** The kernel maintains accounting information about the resources currently being consumed by each process and the total resources consumed by the process in its entire lifetime so far.
- **File table.** The file table is an array of pointers to kernel file structures representing open files. When making file-I/O system calls, processes refer to files by an integer, known as a **file descriptor (fd)**, that the kernel uses to index into this table.
- **File-system context.** Whereas the file table lists the existing open files, the file-system context applies to requests to open new files. The file-system context includes the process's root directory, current working directory, and namespace.
- **Signal-handler table.** UNIX systems can deliver asynchronous signals to a process in response to various external events. The signal-handler table defines the action to take in response to a specific signal. Valid actions include ignoring the signal, terminating the process, and invoking a routine in the process's address space.
- **Virtual memory context.** The virtual memory context describes the full contents of a process's private address space; we discuss it in Section 18.6.

18.4.2 Processes and Threads

Linux provides the `fork()` system call, which duplicates a process without loading a new executable image. Linux also provides the ability to create threads via the `clone()` system call. Linux does not distinguish between processes and threads, however. In fact, Linux generally uses the term *task*—rather than *process* or *thread*—when referring to a flow of control within a program. The `clone()` system call behaves identically to `fork()`, except that it accepts as arguments a set of flags that dictate what resources are shared between the parent and child (whereas a process created with `fork()` shares no resources with its parent). The flags include:

flag	meaning
<code>CLONE_FS</code>	File-system information is shared.
<code>CLONE_VM</code>	The same memory space is shared.
<code>CLONE_SIGHAND</code>	Signal handlers are shared.
<code>CLONE_FILES</code>	The set of open files is shared.

Thus, if `clone()` is passed the flags `CLONE_FS`, `CLONE_VM`, `CLONE_SIGHAND`, and `CLONE_FILES`, the parent and child tasks will share the same file-system information (such as the current working directory), the same memory space, the same signal handlers, and the same set of open files. Using `clone()` in this fashion is equivalent to creating a thread in other systems, since the parent task shares most of its resources with its child task. If none of these flags is set when `clone()` is invoked, however, the associated resources are not shared, resulting in functionality similar to that of the `fork()` system call.

The lack of distinction between processes and threads is possible because Linux does not hold a process's entire context within the main process data structure. Rather, it holds the context within independent subcontexts. Thus, a process's file-system context, file-descriptor table, signal-handler table, and virtual memory context are held in separate data structures. The process data structure simply contains pointers to these other structures, so any number of processes can easily share a subcontext by pointing to the same subcontext and incrementing a reference count.

The arguments to the `clone()` system call tell it which subcontexts to copy and which to share. The new process is always given a new identity and a new scheduling context—these are the essentials of a Linux process. According to the arguments passed, however, the kernel may either create new subcontext data structures initialized so as to be copies of the parent's or set up the new process to use the same subcontext data structures being used by the parent. The `fork()` system call is nothing more than a special case of `clone()` that copies all subcontexts, sharing none.

18.5 Scheduling

Scheduling is the job of allocating CPU time to different tasks within an operating system. Linux, like all UNIX systems, supports **preemptive multitasking**. In such a system, the process scheduler decides which process runs and when.

Making these decisions in a way that balances fairness and performance across many different workloads is one of the more complicated challenges in modern operating systems.

Normally, we think of scheduling as the running and interrupting of user processes, but another aspect of scheduling is also important to Linux: the running of the various kernel tasks. Kernel tasks encompass both tasks that are requested by a running process and tasks that execute internally on behalf of the kernel itself, such as tasks spawned by Linux's I/O subsystem.

18.5.1 Process Scheduling

Linux has two separate process-scheduling algorithms. One is a time-sharing algorithm for fair, preemptive scheduling among multiple processes. The other is designed for real-time tasks, where absolute priorities are more important than fairness.

The scheduling algorithm used for routine time-sharing tasks received a major overhaul with version 2.6 of the kernel. Earlier versions ran a variation of the traditional UNIX scheduling algorithm. This algorithm does not provide adequate support for SMP systems, does not scale well as the number of tasks on the system grows, and does not maintain fairness among interactive tasks, particularly on systems such as desktops and mobile devices. The process scheduler was first overhauled with version 2.5 of the kernel. Version 2.5 implemented a scheduling algorithm that selects which task to run in constant time—known as $O(1)$ —regardless of the number of tasks or processors in the system. The new scheduler also provided increased support for SMP, including processor affinity and load balancing. These changes, while improving scalability, did not improve interactive performance or fairness—and, in fact, made these problems worse under certain workloads. Consequently, the process scheduler was overhauled a second time, with Linux kernel version 2.6. This version ushered in the **Completely Fair Scheduler (CFS)**.

The Linux scheduler is a preemptive, priority-based algorithm with two separate priority ranges: a **real-time** range from 0 to 99 and a **nice value** ranging from -20 to 19. Smaller **nice** values indicate higher priorities. Thus, by increasing the **nice** value, you are decreasing your priority and being “nice” to the rest of the system.

CFS is a significant departure from the traditional UNIX process scheduler. In the latter, the core variables in the scheduling algorithm are priority and time slice. The **time slice** is the length of time—the *slice* of the processor—that a process is afforded. Traditional UNIX systems give processes a fixed time slice, perhaps with a boost or penalty for high- or low-priority processes, respectively. A process may run for the length of its time slice, and higher-priority processes run before lower-priority processes. It is a simple algorithm that many non-UNIX systems employ. Such simplicity worked well for early time-sharing systems but has proved incapable of delivering good interactive performance and fairness on today's modern desktops and mobile devices.

CFS introduced a new scheduling algorithm called **fair scheduling** that eliminates time slices in the traditional sense. Instead of time slices, all processes are allotted a proportion of the processor's time. CFS calculates how long a process should run as a function of the total number of runnable processes.

To start, CFS says that if there are N runnable processes, then each should be afforded $1/N$ of the processor's time. CFS then adjusts this allotment by weighting each process's allotment by its `nice` value. Processes with the default `nice` value have a weight of 1—their priority is unchanged. Processes with a smaller `nice` value (higher priority) receive a higher weight, while processes with a larger `nice` value (lower priority) receive a lower weight. CFS then runs each process for a “time slice” proportional to the process's weight divided by the total weight of all runnable processes.

To calculate the actual length of time a process runs, CFS relies on a configurable variable called **target latency**, which is the interval of time during which every runnable task should run at least once. For example, assume that the target latency is 10 milliseconds. Further assume that we have two runnable processes of the same priority. Each of these processes has the same weight and therefore receives the same proportion of the processor's time. In this case, with a target latency of 10 milliseconds, the first process runs for 5 milliseconds, then the other process runs for 5 milliseconds, then the first process runs for 5 milliseconds again, and so forth. If we have 10 runnable processes, then CFS will run each for a millisecond before repeating.

But what if we had, say, 1,000 processes? Each process would run for 1 microsecond if we followed the procedure just described. Due to switching costs, scheduling processes for such short lengths of time is inefficient. CFS consequently relies on a second configurable variable, the **minimum granularity**, which is a minimum length of time any process is allotted the processor. All processes, regardless of the target latency, will run for at least the minimum granularity. In this manner, CFS ensures that switching costs do not grow unacceptably large when the number of runnable processes grows too large. In doing so, it violates its attempts at fairness. In the usual case, however, the number of runnable processes remains reasonable, and both fairness and switching costs are maximized.

With the switch to fair scheduling, CFS behaves differently from traditional UNIX process schedulers in several ways. Most notably, as we have seen, CFS eliminates the concept of a static time slice. Instead, each process receives a proportion of the processor's time. How long that allotment is depends on how many other processes are runnable. This approach solves several problems in mapping priorities to time slices inherent in preemptive, priority-based scheduling algorithms. It is possible, of course, to solve these problems in other ways without abandoning the classic UNIX scheduler. CFS, however, solves the problems with a simple algorithm that performs well on interactive workloads such as mobile devices without compromising throughput performance on the largest of servers.

18.5.2 Real-Time Scheduling

Linux's real-time scheduling algorithm is significantly simpler than the fair scheduling employed for standard time-sharing processes. Linux implements the two real-time scheduling classes required by POSIX.1b: first-come, first-served (FCFS) and round-robin (Section 6.3.1 and Section 6.3.4, respectively). In both cases, each process has a priority in addition to its scheduling class. The scheduler always runs the process with the highest priority. Among processes of equal priority, it runs the process that has been waiting longest. The only

difference between FCFS and round-robin scheduling is that FCFS processes continue to run until they either exit or block, whereas a round-robin process will be preempted after a while and will be moved to the end of the scheduling queue, so round-robin processes of equal priority will automatically time-share among themselves.

Linux’s real-time scheduling is soft—rather than hard—real time. The scheduler offers strict guarantees about the relative priorities of real-time processes, but the kernel does not offer any guarantees about how quickly a real-time process will be scheduled once that process becomes runnable. In contrast, a hard real-time system can guarantee a minimum latency between when a process becomes runnable and when it actually runs.

18.5.3 Kernel Synchronization

The way the kernel schedules its own operations is fundamentally different from the way it schedules processes. A request for kernel-mode execution can occur in two ways. A running program may request an operating-system service, either explicitly via a system call or implicitly—for example, when a page fault occurs. Alternatively, a device controller may deliver a hardware interrupt that causes the CPU to start executing a kernel-defined handler for that interrupt.

The problem for the kernel is that all these tasks may try to access the same internal data structures. If one kernel task is in the middle of accessing some data structure when an interrupt service routine executes, then that service routine cannot access or modify the same data without risking data corruption. This fact relates to the idea of critical sections—portions of code that access shared data and thus must not be allowed to execute concurrently. As a result, kernel synchronization involves much more than just process scheduling. A framework is required that allows kernel tasks to run without violating the integrity of shared data.

Prior to version 2.6, Linux was a nonpreemptive kernel, meaning that a process running in kernel mode could not be preempted—even if a higher-priority process became available to run. With version 2.6, the Linux kernel became fully preemptive. Now, a task can be preempted when it is running in the kernel.

The Linux kernel provides spinlocks and semaphores (as well as reader–writer versions of these two locks) for locking in the kernel. On SMP machines, the fundamental locking mechanism is a spinlock, and the kernel is designed so that spinlocks are held for only short durations. On single-processor machines, spinlocks are not appropriate for use and are replaced by enabling and disabling kernel preemption. That is, rather than holding a spinlock, the task disables kernel preemption. When the task would otherwise release the spinlock, it enables kernel preemption. This pattern is summarized below:

single processor	multiple processors
Disable kernel preemption.	Acquire spin lock.
Enable kernel preemption.	Release spin lock.

Linux uses an interesting approach to disable and enable kernel preemption. It provides two simple kernel interfaces—`preempt_disable()` and `preempt_enable()`. In addition, the kernel is not preemptible if a kernel-mode task is holding a spinlock. To enforce this rule, each task in the system has a `thread_info` structure that includes the field `preempt_count`, which is a counter indicating the number of locks being held by the task. The counter is incremented when a lock is acquired and decremented when a lock is released. If the value of `preempt_count` for the task currently running is greater than zero, it is not safe to preempt the kernel, as this task currently holds a lock. If the count is zero, the kernel can safely be interrupted, assuming there are no outstanding calls to `preempt_disable()`.

Spinlocks—along with the enabling and disabling of kernel preemption—are used in the kernel only when the lock is held for short durations. When a lock must be held for longer periods, semaphores are used.

The second protection technique used by Linux applies to critical sections that occur in interrupt service routines. The basic tool is the processor's interrupt-control hardware. By disabling interrupts (or using spinlocks) during a critical section, the kernel guarantees that it can proceed without the risk of concurrent access to shared data structures.

However, there is a penalty for disabling interrupts. On most hardware architectures, interrupt enable and disable instructions are not cheap. More importantly, as long as interrupts remain disabled, all I/O is suspended, and any device waiting for servicing will have to wait until interrupts are reenabled; thus, performance degrades. To address this problem, the Linux kernel uses a synchronization architecture that allows long critical sections to run for their entire duration without having interrupts disabled. This ability is especially useful in the networking code. An interrupt in a network device driver can signal the arrival of an entire network packet, which may result in a great deal of code being executed to disassemble, route, and forward that packet within the interrupt service routine.

Linux implements this architecture by separating interrupt service routines into two sections: the top half and the bottom half. The *top half* is the standard interrupt service routine that runs with recursive interrupts disabled. Interrupts of the same number (or line) are disabled, but other interrupts may run. The *bottom half* of a service routine is run, with all interrupts enabled, by a miniature scheduler that ensures that bottom halves never interrupt themselves. The bottom-half scheduler is invoked automatically whenever an interrupt service routine exits.

This separation means that the kernel can complete any complex processing that has to be done in response to an interrupt without worrying about being interrupted itself. If another interrupt occurs while a bottom half is executing, then that interrupt can request that the same bottom half execute, but the execution will be deferred until the one currently running completes. Each execution of the bottom half can be interrupted by a top half but can never be interrupted by a similar bottom half.

The top-half/bottom-half architecture is completed by a mechanism for disabling selected bottom halves while executing normal, foreground kernel code. The kernel can code critical sections easily using this system. Interrupt handlers can code their critical sections as bottom halves; and when the foreground kernel wants to enter a critical section, it can disable any relevant

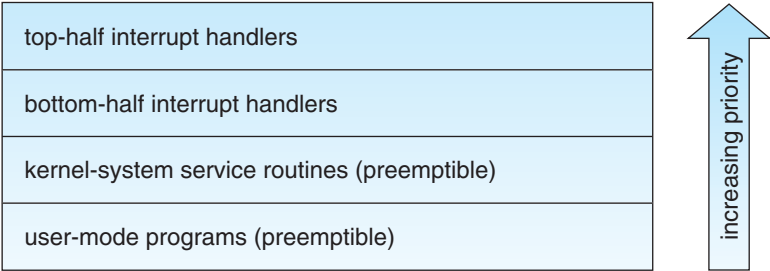


Figure 18.2 Interrupt protection levels.

bottom halves to prevent any other critical sections from interrupting it. At the end of the critical section, the kernel can reenable the bottom halves and run any bottom-half tasks that have been queued by top-half interrupt service routines during the critical section.

Figure 18.2 summarizes the various levels of interrupt protection within the kernel. Each level may be interrupted by code running at a higher level but will never be interrupted by code running at the same or a lower level. Except for user-mode code, user processes can always be preempted by another process when a time-sharing scheduling interrupt occurs.

18.5.4 Symmetric Multiprocessing

The Linux 2.0 kernel was the first stable Linux kernel to support **symmetric multiprocessor (SMP)** hardware, allowing separate processes to execute in parallel on separate processors. The original implementation of SMP imposed the restriction that only one processor at a time could be executing kernel code.

In version 2.2 of the kernel, a single kernel spinlock (sometimes termed **BKL** for “big kernel lock”) was created to allow multiple processes (running on different processors) to be active in the kernel concurrently. However, the BKL provided a very coarse level of locking granularity, resulting in poor scalability to machines with many processors and processes. Later releases of the kernel made the SMP implementation more scalable by splitting this single kernel spinlock into multiple locks, each of which protects only a small subset of the kernel’s data structures. Such spinlocks are described in Section 18.5.3. The 3.0 kernel provides additional SMP enhancements, including ever-finer locking, processor affinity, and load-balancing algorithms.

18.6 Memory Management

Memory management under Linux has two components. The first deals with allocating and freeing physical memory—pages, groups of pages, and small blocks of RAM. The second handles virtual memory, which is memory-mapped into the address space of running processes. In this section, we describe these two components and then examine the mechanisms by which the loadable components of a new program are brought into a process’s virtual memory in response to an `exec()` system call.

18.6.1 Management of Physical Memory

Due to specific hardware constraints, Linux separates physical memory into four different **zones**, or regions:

- ZONE_DMA
- ZONE_DMA32
- ZONE_NORMAL
- ZONE_HIGHMEM

These zones are architecture specific. For example, on the Intel x86-32 architecture, certain ISA (industry standard architecture) devices can only access the lower 16 MB of physical memory using DMA. On these systems, the first 16 MB of physical memory comprise ZONE_DMA. On other systems, certain devices can only access the first 4 GB of physical memory, despite supporting 64-bit addresses. On such systems, the first 4 GB of physical memory comprise ZONE_DMA32. ZONE_HIGHMEM (for “high memory”) refers to physical memory that is not mapped into the kernel address space. For example, on the 32-bit Intel architecture (where 2^{32} provides a 4-GB address space), the kernel is mapped into the first 896 MB of the address space; the remaining memory is referred to as **high memory** and is allocated from ZONE_HIGHMEM. Finally, ZONE_NORMAL comprises everything else—the normal, regularly mapped pages. Whether an architecture has a given zone depends on its constraints. A modern, 64-bit architecture such as Intel x86-64 has a small 16 MB ZONE_DMA (for legacy devices) and all the rest of its memory in ZONE_NORMAL, with no “high memory”.

The relationship of zones and physical addresses on the Intel x86-32 architecture is shown in Figure 18.3. The kernel maintains a list of free pages for each zone. When a request for physical memory arrives, the kernel satisfies the request using the appropriate zone.

The primary physical-memory manager in the Linux kernel is the **page allocator**. Each zone has its own allocator, which is responsible for allocating and freeing all physical pages for the zone and is capable of allocating ranges of physically contiguous pages on request. The allocator uses a buddy system (Section 9.8.1) to keep track of available physical pages. In this scheme, adjacent units of allocatable memory are paired together (hence its name). Each allocatable memory region has an adjacent partner (or buddy). Whenever two allocated partner regions are freed up, they are combined to form a larger region—a **buddy heap**. That larger region also has a partner, with which it can combine to form a still larger free region. Conversely, if a small memory request

zone	physical memory
ZONE_DMA	< 16 MB
ZONE_NORMAL	16 .. 896 MB
ZONE_HIGHMEM	> 896 MB

Figure 18.3 Relationship of zones and physical addresses in Intel x86-32.

cannot be satisfied by allocation of an existing small free region, then a larger free region will be subdivided into two partners to satisfy the request. Separate linked lists are used to record the free memory regions of each allowable size. Under Linux, the smallest size allocatable under this mechanism is a single physical page. Figure 18.4 shows an example of buddy-heap allocation. A 4-KB region is being allocated, but the smallest available region is 16 KB. The region is broken up recursively until a piece of the desired size is available.

Ultimately, all memory allocations in the Linux kernel are made either statically, by drivers that reserve a contiguous area of memory during system boot time, or dynamically, by the page allocator. However, kernel functions do not have to use the basic allocator to reserve memory. Several specialized memory-management subsystems use the underlying page allocator to manage their own pools of memory. The most important are the virtual memory system, described in Section 18.6.2; the `kmalloc()` variable-length allocator; the slab allocator, used for allocating memory for kernel data structures; and the page cache, used for caching pages belonging to files.

Many components of the Linux operating system need to allocate entire pages on request, but often smaller blocks of memory are required. The kernel provides an additional allocator for arbitrary-sized requests, where the size of a request is not known in advance and may be only a few bytes. Analogous to the C language’s `malloc()` function, this `kmalloc()` service allocates entire physical pages on demand but then splits them into smaller pieces. The kernel maintains lists of pages in use by the `kmalloc()` service. Allocating memory involves determining the appropriate list and either taking the first free piece available on the list or allocating a new page and splitting it up. Memory regions claimed by the `kmalloc()` system are allocated permanently until they are freed explicitly with a corresponding call to `kfree()`; the `kmalloc()` system cannot reallocate or reclaim these regions in response to memory shortages.

Another strategy adopted by Linux for allocating kernel memory is known as slab allocation. A **slab** is used for allocating memory for kernel data structures and is made up of one or more physically contiguous pages. A **cache** consists of one or more slabs. There is a single cache for each unique kernel data structure—for example, a cache for the data structure representing process descriptors, a cache for file objects, a cache for inodes, and so forth.

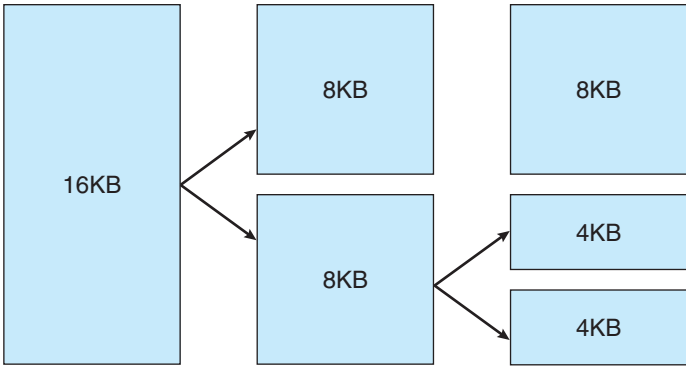


Figure 18.4 Splitting of memory in the buddy system.

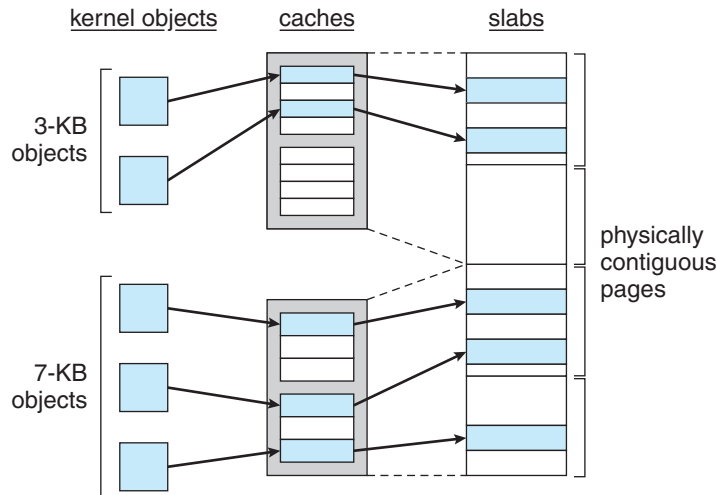


Figure 18.5 Slab allocator in Linux.

Each cache is populated with **objects** that are instantiations of the kernel data structure the cache represents. For example, the cache representing inodes stores instances of inode structures, and the cache representing process descriptors stores instances of process descriptor structures. The relationship among slabs, caches, and objects is shown in Figure 18.5. The figure shows two kernel objects 3 KB in size and three objects 7 KB in size. These objects are stored in the respective caches for 3-KB and 7-KB objects.

The slab-allocation algorithm uses caches to store kernel objects. When a cache is created, a number of objects are allocated to the cache. The number of objects in the cache depends on the size of the associated slab. For example, a 12-KB slab (made up of three contiguous 4-KB pages) could store six 2-KB objects. Initially, all the objects in the cache are marked as *free*. When a new object for a kernel data structure is needed, the allocator can assign any free object from the cache to satisfy the request. The object assigned from the cache is marked as *used*.

Let's consider a scenario in which the kernel requests memory from the slab allocator for an object representing a process descriptor. In Linux systems, a process descriptor is of the type `struct task_struct`, which requires approximately 1.7 KB of memory. When the Linux kernel creates a new task, it requests the necessary memory for the `struct task_struct` object from its cache. The cache will fulfill the request using a `struct task_struct` object that has already been allocated in a slab and is marked as *free*.

In Linux, a slab may be in one of three possible states:

1. **Full.** All objects in the slab are marked as *used*.
2. **Empty.** All objects in the slab are marked as *free*.
3. **Partial.** The slab consists of both *used* and *free* objects.

The slab allocator first attempts to satisfy the request with a *free* object in a *partial* slab. If none exist, a *free* object is assigned from an *empty* slab. If no *empty* slabs are available, a new slab is allocated from contiguous physical

pages and assigned to a cache; memory for the object is allocated from this slab.

Two other main subsystems in Linux do their own management of physical pages: the page cache and the virtual memory system. These systems are closely related to each other. The **page cache** is the kernel's main cache for files and is the main mechanism through which I/O to block devices (Section 18.8.1) is performed. File systems of all types, including the native Linux disk-based file systems and the NFS networked file system, perform their I/O through the page cache. The page cache stores entire pages of file contents and is not limited to block devices. It can also cache networked data. The virtual memory system manages the contents of each process's virtual address space. These two systems interact closely with each other because reading a page of data into the page cache requires mapping pages in the page cache using the virtual memory system. In the following section, we look at the virtual memory system in greater detail.

18.6.2 Virtual Memory

The Linux virtual memory system is responsible for maintaining the address space accessible to each process. It creates pages of virtual memory on demand and manages loading those pages from disk and swapping them back out to disk as required. Under Linux, the virtual memory manager maintains two separate views of a process's address space: as a set of separate regions and as a set of pages.

The first view of an address space is the logical view, describing instructions that the virtual memory system has received concerning the layout of the address space. In this view, the address space consists of a set of nonoverlapping regions, each region representing a continuous, page-aligned subset of the address space. Each region is described internally by a single `vm_area_struct` structure that defines the properties of the region, including the process's read, write, and execute permissions in the region as well as information about any files associated with the region. The regions for each address space are linked into a balanced binary tree to allow fast lookup of the region corresponding to any virtual address.

The kernel also maintains a second, physical view of each address space. This view is stored in the hardware page tables for the process. The page-table entries identify the exact current location of each page of virtual memory, whether it is on disk or in physical memory. The physical view is managed by a set of routines, which are invoked from the kernel's software-interrupt handlers whenever a process tries to access a page that is not currently present in the page tables. Each `vm_area_struct` in the address-space description contains a field pointing to a table of functions that implement the key page-management functionality for any given virtual memory region. All requests to read or write an unavailable page are eventually dispatched to the appropriate handler in the function table for the `vm_area_struct`, so that the central memory-management routines do not have to know the details of managing each possible type of memory region.

18.6.2.1 Virtual Memory Regions

Linux implements several types of virtual memory regions. One property that characterizes virtual memory is the backing store for the region, which describes where the pages for the region come from. Most memory regions are backed either by a file or by nothing. A region backed by nothing is the simplest type of virtual memory region. Such a region represents **demand-zero memory**: when a process tries to read a page in such a region, it is simply given back a page of memory filled with zeros.

A region backed by a file acts as a viewport onto a section of that file. Whenever the process tries to access a page within that region, the page table is filled with the address of a page within the kernel's page cache corresponding to the appropriate offset in the file. The same page of physical memory is used by both the page cache and the process's page tables, so any changes made to the file by the file system are immediately visible to any processes that have mapped that file into their address space. Any number of processes can map the same region of the same file, and they will all end up using the same page of physical memory for the purpose.

A virtual memory region is also defined by its reaction to writes. The mapping of a region into the process's address space can be either *private* or *shared*. If a process writes to a privately mapped region, then the pager detects that a copy-on-write is necessary to keep the changes local to the process. In contrast, writes to a shared region result in updating of the object mapped into that region, so that the change will be visible immediately to any other process that is mapping that object.

18.6.2.2 Lifetime of a Virtual Address Space

The kernel creates a new virtual address space in two situations: when a process runs a new program with the `exec()` system call and when a new process is created by the `fork()` system call. The first case is easy. When a new program is executed, the process is given a new, completely empty virtual address space. It is up to the routines for loading the program to populate the address space with virtual memory regions.

The second case, creating a new process with `fork()`, involves creating a complete copy of the existing process's virtual address space. The kernel copies the parent process's `vm_area_struct` descriptors, then creates a new set of page tables for the child. The parent's page tables are copied directly into the child's, and the reference count of each page covered is incremented. Thus, after the fork, the parent and child share the same physical pages of memory in their address spaces.

A special case occurs when the copying operation reaches a virtual memory region that is mapped privately. Any pages to which the parent process has written within such a region are private, and subsequent changes to these pages by either the parent or the child must not update the page in the other process's address space. When the page-table entries for such regions are copied, they are set to be read only and are marked for copy-on-write. As long as neither process modifies these pages, the two processes share the same page of physical memory. However, if either process tries to modify a copy-on-write page, the reference count on the page is checked. If the page is still shared, then the

process copies the page's contents to a brand-new page of physical memory and uses its copy instead. This mechanism ensures that private data pages are shared between processes whenever possible and copies are made only when absolutely necessary.

18.6.2.3 Swapping and Paging

An important task for a virtual memory system is to relocate pages of memory from physical memory out to disk when that memory is needed. Early UNIX systems performed this relocation by swapping out the contents of entire processes at once, but modern versions of UNIX rely more on paging—the movement of individual pages of virtual memory between physical memory and disk. Linux does not implement whole-process swapping; it uses the newer paging mechanism exclusively.

The paging system can be divided into two sections. First, the **policy algorithm** decides which pages to write out to disk and when to write them. Second, the **paging mechanism** carries out the transfer and pages data back into physical memory when they are needed again.

Linux's **pageout policy** uses a modified version of the standard clock (or second-chance) algorithm described in Section 9.4.5.2. Under Linux, a multiple-pass clock is used, and every page has an *age* that is adjusted on each pass of the clock. The age is more precisely a measure of the page's youthfulness, or how much activity the page has seen recently. Frequently accessed pages will attain a higher age value, but the age of infrequently accessed pages will drop toward zero with each pass. This age valuing allows the pager to select pages to page out based on a least frequently used (LFU) policy.

The paging mechanism supports paging both to dedicated swap devices and partitions and to normal files, although swapping to a file is significantly slower due to the extra overhead incurred by the file system. Blocks are allocated from the swap devices according to a bitmap of used blocks, which is maintained in physical memory at all times. The allocator uses a next-fit algorithm to try to write out pages to continuous runs of disk blocks for improved performance. The allocator records the fact that a page has been paged out to disk by using a feature of the page tables on modern processors: the page-table entry's page-not-present bit is set, allowing the rest of the page-table entry to be filled with an index identifying where the page has been written.

18.6.2.4 Kernel Virtual Memory

Linux reserves for its own internal use a constant, architecture-dependent region of the virtual address space of every process. The page-table entries that map to these kernel pages are marked as protected, so that the pages are not visible or modifiable when the processor is running in user mode. This kernel virtual memory area contains two regions. The first is a static area that contains page-table references to every available physical page of memory in the system, so that a simple translation from physical to virtual addresses occurs when kernel code is run. The core of the kernel, along with all pages allocated by the normal page allocator, resides in this region.

The remainder of the kernel's reserved section of address space is not reserved for any specific purpose. Page-table entries in this address range can be modified by the kernel to point to any other areas of memory. The kernel provides a pair of facilities that allow kernel code to use this virtual memory. The `vmalloc()` function allocates an arbitrary number of physical pages of memory that may not be physically contiguous into a single region of virtually contiguous kernel memory. The `vremap()` function maps a sequence of virtual addresses to point to an area of memory used by a device driver for memory-mapped I/O.

18.6.3 Execution and Loading of User Programs

The Linux kernel's execution of user programs is triggered by a call to the `exec()` system call. This `exec()` call commands the kernel to run a new program within the current process, completely overwriting the current execution context with the initial context of the new program. The first job of this system service is to verify that the calling process has permission rights to the file being executed. Once that matter has been checked, the kernel invokes a loader routine to start running the program. The loader does not necessarily load the contents of the program file into physical memory, but it does at least set up the mapping of the program into virtual memory.

There is no single routine in Linux for loading a new program. Instead, Linux maintains a table of possible loader functions, and it gives each such function the opportunity to try loading the given file when an `exec()` system call is made. The initial reason for this loader table was that, between the releases of the 1.0 and 1.2 kernels, the standard format for Linux's binary files was changed. Older Linux kernels understood the `a.out` format for binary files—a relatively simple format common on older UNIX systems. Newer Linux systems use the more modern **ELF** format, now supported by most current UNIX implementations. ELF has a number of advantages over `a.out`, including flexibility and extendability. New sections can be added to an ELF binary (for example, to add extra debugging information) without causing the loader routines to become confused. By allowing registration of multiple loader routines, Linux can easily support the ELF and `a.out` binary formats in a single running system.

In Section 18.6.3.1 and Section 18.6.3.2, we concentrate exclusively on the loading and running of ELF-format binaries. The procedure for loading `a.out` binaries is simpler but similar in operation.

18.6.3.1 Mapping of Programs into Memory

Under Linux, the binary loader does not load a binary file into physical memory. Rather, the pages of the binary file are mapped into regions of virtual memory. Only when the program tries to access a given page will a page fault result in the loading of that page into physical memory using demand paging.

It is the responsibility of the kernel's binary loader to set up the initial memory mapping. An ELF-format binary file consists of a header followed by several page-aligned sections. The ELF loader works by reading the header and mapping the sections of the file into separate regions of virtual memory.

Figure 18.6 shows the typical layout of memory regions set up by the ELF loader. In a reserved region at one end of the address space sits the kernel, in

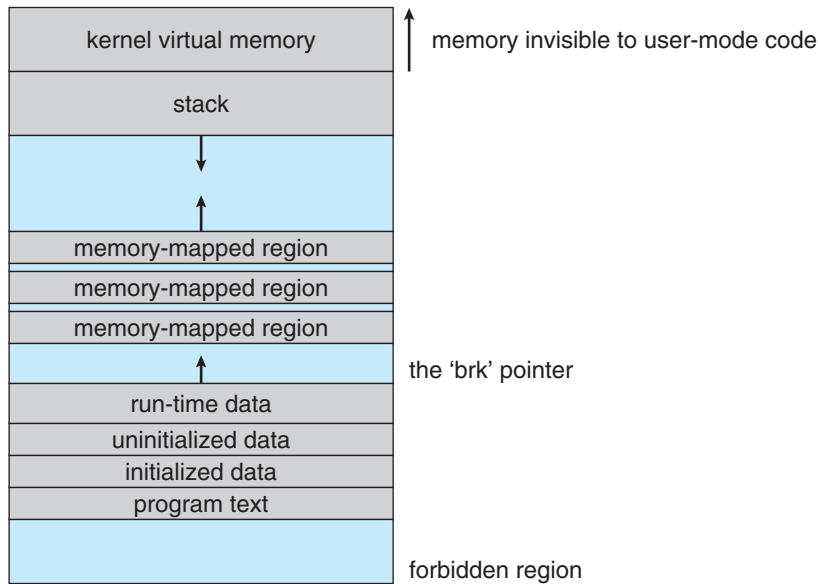


Figure 18.6 Memory layout for ELF programs.

its own privileged region of virtual memory inaccessible to normal user-mode programs. The rest of virtual memory is available to applications, which can use the kernel’s memory-mapping functions to create regions that map a portion of a file or that are available for application data.

The loader’s job is to set up the initial memory mapping to allow the execution of the program to start. The regions that need to be initialized include the stack and the program’s text and data regions.

The stack is created at the top of the user-mode virtual memory; it grows downward toward lower-numbered addresses. It includes copies of the arguments and environment variables given to the program in the `exec()` system call. The other regions are created near the bottom end of virtual memory. The sections of the binary file that contain program text or read-only data are mapped into memory as a write-protected region. Writable initialized data are mapped next; then any uninitialized data are mapped in as a private demand-zero region.

Directly beyond these fixed-sized regions is a variable-sized region that programs can expand as needed to hold data allocated at run time. Each process has a pointer, `brk`, that points to the current extent of this data region, and processes can extend or contract their `brk` region with a single system call — `sbrk()`.

Once these mappings have been set up, the loader initializes the process’s program-counter register with the starting point recorded in the ELF header, and the process can be scheduled.

18.6.3.2 Static and Dynamic Linking

Once the program has been loaded and has started running, all the necessary contents of the binary file have been loaded into the process’s virtual address

space. However, most programs also need to run functions from the system libraries, and these library functions must also be loaded. In the simplest case, the necessary library functions are embedded directly in the program's executable binary file. Such a program is statically linked to its libraries, and statically linked executables can commence running as soon as they are loaded.

The main disadvantage of static linking is that every program generated must contain copies of exactly the same common system library functions. It is much more efficient, in terms of both physical memory and disk-space usage, to load the system libraries into memory only once. Dynamic linking allows that to happen.

Linux implements dynamic linking in user mode through a special linker library. Every dynamically linked program contains a small, statically linked function that is called when the program starts. This static function just maps the link library into memory and runs the code that the function contains. The link library determines the dynamic libraries required by the program and the names of the variables and functions needed from those libraries by reading the information contained in sections of the ELF binary. It then maps the libraries into the middle of virtual memory and resolves the references to the symbols contained in those libraries. It does not matter exactly where in memory these shared libraries are mapped: they are compiled into **position-independent code (PIC)**, which can run at any address in memory.

18.7 File Systems

Linux retains UNIX's standard file-system model. In UNIX, a file does not have to be an object stored on disk or fetched over a network from a remote file server. Rather, UNIX files can be anything capable of handling the input or output of a stream of data. Device drivers can appear as files, and interprocess-communication channels or network connections also look like files to the user.

The Linux kernel handles all these types of files by hiding the implementation details of any single file type behind a layer of software, the virtual file system (VFS). Here, we first cover the virtual file system and then discuss the standard Linux file system—ext3.

18.7.1 The Virtual File System

The Linux VFS is designed around object-oriented principles. It has two components: a set of definitions that specify what file-system objects are allowed to look like and a layer of software to manipulate the objects. The VFS defines four main object types:

- An **inode object** represents an individual file.
- A **file object** represents an open file.
- A **superblock object** represents an entire file system.
- A **dentry object** represents an individual directory entry.

For each of these four object types, the VFS defines a set of operations. Every object of one of these types contains a pointer to a function table. The function table lists the addresses of the actual functions that implement the defined operations for that object. For example, an abbreviated API for some of the file object's operations includes:

- `int open(. . .)` — Open a file.
- `ssize_t read(. . .)` — Read from a file.
- `ssize_t write(. . .)` — Write to a file.
- `int mmap(. . .)` — Memory-map a file.

The complete definition of the file object is specified in the `struct file_operations`, which is located in the file `/usr/include/linux/fs.h`. An implementation of the file object (for a specific file type) is required to implement each function specified in the definition of the file object.

The VFS software layer can perform an operation on one of the file-system objects by calling the appropriate function from the object's function table, without having to know in advance exactly what kind of object it is dealing with. The VFS does not know, or care, whether an inode represents a networked file, a disk file, a network socket, or a directory file. The appropriate function for that file's `read()` operation will always be at the same place in its function table, and the VFS software layer will call that function without caring how the data are actually read.

The inode and file objects are the mechanisms used to access files. An inode object is a data structure containing pointers to the disk blocks that contain the actual file contents, and a file object represents a point of access to the data in an open file. A process cannot access an inode's contents without first obtaining a file object pointing to the inode. The file object keeps track of where in the file the process is currently reading or writing, to keep track of sequential file I/O. It also remembers the permissions (for example, read or write) requested when the file was opened and tracks the process's activity if necessary to perform adaptive read-ahead, fetching file data into memory before the process requests the data, to improve performance.

File objects typically belong to a single process, but inode objects do not. There is one file object for every instance of an open file, but always only a single inode object. Even when a file is no longer in use by any process, its inode object may still be cached by the VFS to improve performance if the file is used again in the near future. All cached file data are linked onto a list in the file's inode object. The inode also maintains standard information about each file, such as the owner, size, and time most recently modified.

Directory files are dealt with slightly differently from other files. The UNIX programming interface defines a number of operations on directories, such as creating, deleting, and renaming a file in a directory. The system calls for these directory operations do not require that the user open the files concerned, unlike the case for reading or writing data. The VFS therefore defines these directory operations in the inode object, rather than in the file object.

The superblock object represents a connected set of files that form a self-contained file system. The operating-system kernel maintains a single

superblock object for each disk device mounted as a file system and for each networked file system currently connected. The main responsibility of the superblock object is to provide access to inodes. The VFS identifies every inode by a unique file-system/inode number pair, and it finds the inode corresponding to a particular inode number by asking the superblock object to return the inode with that number.

Finally, a dentry object represents a directory entry, which may include the name of a directory in the path name of a file (such as `/usr`) or the actual file (such as `stdio.h`). For example, the file `/usr/include/stdio.h` contains the directory entries (1) `/`, (2) `usr`, (3) `include`, and (4) `stdio.h`. Each of these values is represented by a separate dentry object.

As an example of how dentry objects are used, consider the situation in which a process wishes to open the file with the pathname `/usr/include/stdio.h` using an editor. Because Linux treats directory names as files, translating this path requires first obtaining the inode for the root—`/`. The operating system must then read through this file to obtain the inode for the file `include`. It must continue this process until it obtains the inode for the file `stdio.h`. Because path-name translation can be a time-consuming task, Linux maintains a cache of dentry objects, which is consulted during path-name translation. Obtaining the inode from the dentry cache is considerably faster than having to read the on-disk file.

18.7.2 The Linux ext3 File System

The standard on-disk file system used by Linux is called **ext3**, for historical reasons. Linux was originally programmed with a Minix-compatible file system, to ease exchanging data with the Minix development system, but that file system was severely restricted by 14-character file-name limits and a maximum file-system size of 64 MB. The Minix file system was superseded by a new file system, which was christened the **extended file system (extfs)**. A later redesign to improve performance and scalability and to add a few missing features led to the **second extended file system (ext2)**. Further development added journaling capabilities, and the system was renamed the **third extended file system (ext3)**. Linux kernel developers are working on augmenting ext3 with modern file-system features such as extents. This new file system is called the **fourth extended file system (ext4)**. The rest of this section discusses ext3, however, since it remains the most-deployed Linux file system. Most of the discussion applies equally to ext4.

Linux's ext3 has much in common with the BSD Fast File System (FFS) (Section A.7.7). It uses a similar mechanism for locating the data blocks belonging to a specific file, storing data-block pointers in indirect blocks throughout the file system with up to three levels of indirection. As in FFS, directory files are stored on disk just like normal files, although their contents are interpreted differently. Each block in a directory file consists of a linked list of entries. In turn, each entry contains the length of the entry, the name of a file, and the inode number of the inode to which that entry refers.

The main differences between ext3 and FFS lie in their disk-allocation policies. In FFS, the disk is allocated to files in blocks of 8 KB. These blocks are subdivided into fragments of 1 KB for storage of small files or partially filled blocks at the ends of files. In contrast, ext3 does not use fragments at all

but performs all its allocations in smaller units. The default block size on ext3 varies as a function of the total size of the file system. Supported block sizes are 1, 2, 4, and 8 KB.

To maintain high performance, the operating system must try to perform I/O operations in large chunks whenever possible by clustering physically adjacent I/O requests. Clustering reduces the per-request overhead incurred by device drivers, disks, and disk-controller hardware. A block-sized I/O request size is too small to maintain good performance, so ext3 uses allocation policies designed to place logically adjacent blocks of a file into physically adjacent blocks on disk, so that it can submit an I/O request for several disk blocks as a single operation.

The ext3 allocation policy works as follows: As in FFS, an ext3 file system is partitioned into multiple segments. In ext3, these are called **block groups**. FFS uses the similar concept of **cylinder groups**, where each group corresponds to a single cylinder of a physical disk. (Note that modern disk-drive technology packs sectors onto the disk at different densities, and thus with different cylinder sizes, depending on how far the disk head is from the center of the disk. Therefore, fixed-sized cylinder groups do not necessarily correspond to the disk's geometry.)

When allocating a file, ext3 must first select the block group for that file. For data blocks, it attempts to allocate the file to the block group to which the file's inode has been allocated. For inode allocations, it selects the block group in which the file's parent directory resides for nondirectory files. Directory files are not kept together but rather are dispersed throughout the available block groups. These policies are designed not only to keep related information within the same block group but also to spread out the disk load among the disk's block groups to reduce the fragmentation of any one area of the disk.

Within a block group, ext3 tries to keep allocations physically contiguous if possible, reducing fragmentation if it can. It maintains a bitmap of all free blocks in a block group. When allocating the first blocks for a new file, it starts searching for a free block from the beginning of the block group. When extending a file, it continues the search from the block most recently allocated to the file. The search is performed in two stages. First, ext3 searches for an entire free byte in the bitmap; if it fails to find one, it looks for any free bit. The search for free bytes aims to allocate disk space in chunks of at least eight blocks where possible.

Once a free block has been identified, the search is extended backward until an allocated block is encountered. When a free byte is found in the bitmap, this backward extension prevents ext3 from leaving a hole between the most recently allocated block in the previous nonzero byte and the zero byte found. Once the next block to be allocated has been found by either bit or byte search, ext3 extends the allocation forward for up to eight blocks and preallocates these extra blocks to the file. This preallocation helps to reduce fragmentation during interleaved writes to separate files and also reduces the CPU cost of disk allocation by allocating multiple blocks simultaneously. The preallocated blocks are returned to the free-space bitmap when the file is closed.

Figure 18.7 illustrates the allocation policies. Each row represents a sequence of set and unset bits in an allocation bitmap, indicating used and free blocks on disk. In the first case, if we can find any free blocks sufficiently near the start of the search, then we allocate them no matter how fragmented

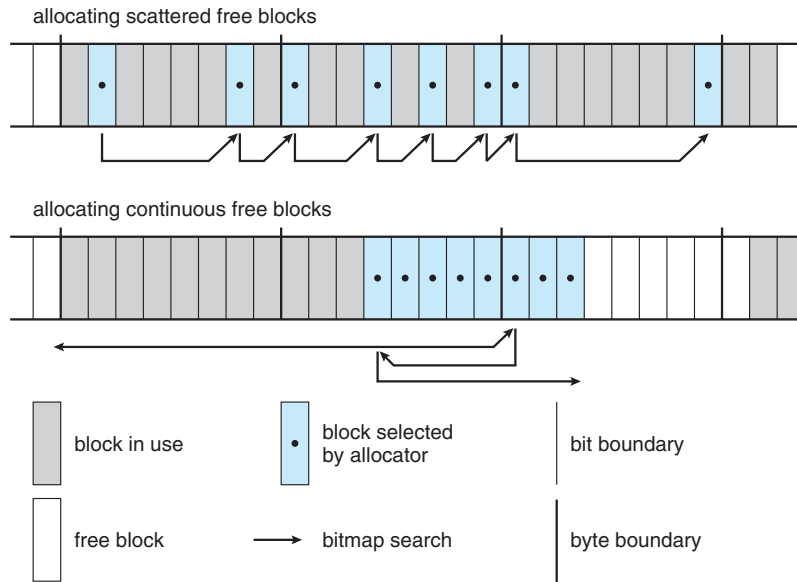


Figure 18.7 ext3 block-allocation policies.

they may be. The fragmentation is partially compensated for by the fact that the blocks are close together and can probably all be read without any disk seeks. Furthermore, allocating them all to one file is better in the long run than allocating isolated blocks to separate files once large free areas become scarce on disk. In the second case, we have not immediately found a free block close by, so we search forward for an entire free byte in the bitmap. If we allocated that byte as a whole, we would end up creating a fragmented area of free space between it and the allocation preceding it. Thus, before allocating, we back up to make this allocation flush with the allocation preceding it, and then we allocate forward to satisfy the default allocation of eight blocks.

18.7.3 Journaling

The ext3 file system supports a popular feature called **journaling**, whereby modifications to the file system are written sequentially to a journal. A set of operations that performs a specific task is a **transaction**. Once a transaction is written to the journal, it is considered to be committed. Meanwhile, the journal entries relating to the transaction are replayed across the actual file-system structures. As the changes are made, a pointer is updated to indicate which actions have completed and which are still incomplete. When an entire committed transaction is completed, it is removed from the journal. The journal, which is actually a circular buffer, may be in a separate section of the file system, or it may even be on a separate disk spindle. It is more efficient, but more complex, to have it under separate read–write heads, thereby decreasing head contention and seek times.

If the system crashes, some transactions may remain in the journal. Those transactions were never completed to the file system even though they were committed by the operating system, so they must be completed once the system

recovers. The transactions can be executed from the pointer until the work is complete, and the file-system structures remain consistent. The only problem occurs when a transaction has been aborted—that is, it was not committed before the system crashed. Any changes from those transactions that were applied to the file system must be undone, again preserving the consistency of the file system. This recovery is all that is needed after a crash, eliminating all problems with consistency checking.

Journaling file systems may perform some operations faster than non-journaling systems, as updates proceed much faster when they are applied to the in-memory journal rather than directly to the on-disk data structures. The reason for this improvement is found in the performance advantage of sequential I/O over random I/O. Costly synchronous random writes to the file system are turned into much less costly synchronous sequential writes to the file system's journal. Those changes, in turn, are replayed asynchronously via random writes to the appropriate structures. The overall result is a significant gain in performance of file-system metadata-oriented operations, such as file creation and deletion. Due to this performance improvement, ext3 can be configured to journal only metadata and not file data.

18.7.4 The Linux Process File System

The flexibility of the Linux VFS enables us to implement a file system that does not store data persistently at all but rather provides an interface to some other functionality. The Linux **process file system**, known as the `/proc` file system, is an example of a file system whose contents are not actually stored anywhere but are computed on demand according to user file I/O requests.

A `/proc` file system is not unique to Linux. SVR4 UNIX introduced a `/proc` file system as an efficient interface to the kernel's process debugging support. Each subdirectory of the file system corresponded not to a directory on any disk but rather to an active process on the current system. A listing of the file system reveals one directory per process, with the directory name being the ASCII decimal representation of the process's unique process identifier (PID).

Linux implements such a `/proc` file system but extends it greatly by adding a number of extra directories and text files under the file system's root directory. These new entries correspond to various statistics about the kernel and the associated loaded drivers. The `/proc` file system provides a way for programs to access this information as plain text files; the standard UNIX user environment provides powerful tools to process such files. For example, in the past, the traditional UNIX `ps` command for listing the states of all running processes has been implemented as a privileged process that reads the process state directly from the kernel's virtual memory. Under Linux, this command is implemented as an entirely unprivileged program that simply parses and formats the information from `/proc`.

The `/proc` file system must implement two things: a directory structure and the file contents within. Because a UNIX file system is defined as a set of file and directory inodes identified by their inode numbers, the `/proc` file system must define a unique and persistent inode number for each directory and the associated files. Once such a mapping exists, the file system can use this inode number to identify just what operation is required when a user tries to read from a particular file inode or to perform a lookup in a particular directory

inode. When data are read from one of these files, the `/proc` file system will collect the appropriate information, format it into textual form, and place it into the requesting process's read buffer.

The mapping from inode number to information type splits the inode number into two fields. In Linux, a PID is 16 bits in size, but an inode number is 32 bits. The top 16 bits of the inode number are interpreted as a PID, and the remaining bits define what type of information is being requested about that process.

A PID of zero is not valid, so a zero PID field in the inode number is taken to mean that this inode contains global—rather than process-specific—information. Separate global files exist in `/proc` to report information such as the kernel version, free memory, performance statistics, and drivers currently running.

Not all the inode numbers in this range are reserved. The kernel can allocate new `/proc` inode mappings dynamically, maintaining a bitmap of allocated inode numbers. It also maintains a tree data structure of registered global `/proc` file-system entries. Each entry contains the file's inode number, file name, and access permissions, along with the special functions used to generate the file's contents. Drivers can register and deregister entries in this tree at any time, and a special section of the tree—appearing under the `/proc/sys` directory—is reserved for kernel variables. Files under this tree are managed by a set of common handlers that allow both reading and writing of these variables, so a system administrator can tune the value of kernel parameters simply by writing out the new desired values in ASCII decimal to the appropriate file.

To allow efficient access to these variables from within applications, the `/proc/sys` subtree is made available through a special system call, `sysctl()`, that reads and writes the same variables in binary, rather than in text, without the overhead of the file system. `sysctl()` is not an extra facility; it simply reads the `/proc` dynamic entry tree to identify the variables to which the application is referring.

18.8 Input and Output

To the user, the I/O system in Linux looks much like that in any UNIX system. That is, to the extent possible, all device drivers appear as normal files. Users can open an access channel to a device in the same way they open any other file—devices can appear as objects within the file system. The system administrator can create special files within a file system that contain references to a specific device driver, and a user opening such a file will be able to read from and write to the device referenced. By using the normal file-protection system, which determines who can access which file, the administrator can set access permissions for each device.

Linux splits all devices into three classes: block devices, character devices, and network devices. Figure 18.8 illustrates the overall structure of the device-driver system.

Block devices include all devices that allow random access to completely independent, fixed-sized blocks of data, including hard disks and floppy disks, CD-ROMs and Blu-ray discs, and flash memory. Block devices are typically

used to store file systems, but direct access to a block device is also allowed so that programs can create and repair the file system that the device contains. Applications can also access these block devices directly if they wish. For example, a database application may prefer to perform its own fine-tuned layout of data onto a disk rather than using the general-purpose file system.

Character devices include most other devices, such as mice and keyboards. The fundamental difference between block and character devices is random access—block devices are accessed randomly, while character devices are accessed serially. For example, seeking to a certain position in a file might be supported for a DVD but makes no sense for a pointing device such as a mouse.

Network devices are dealt with differently from block and character devices. Users cannot directly transfer data to network devices. Instead, they must communicate indirectly by opening a connection to the kernel's networking subsystem. We discuss the interface to network devices separately in Section 18.10.

18.8.1 Block Devices

Block devices provide the main interface to all disk devices in a system. Performance is particularly important for disks, and the block-device system must provide functionality to ensure that disk access is as fast as possible. This functionality is achieved through the scheduling of I/O operations.

In the context of block devices, a block represents the unit with which the kernel performs I/O. When a block is read into memory, it is stored in a buffer. The **request manager** is the layer of software that manages the reading and writing of buffer contents to and from a block-device driver.

A separate list of requests is kept for each block-device driver. Traditionally, these requests have been scheduled according to a unidirectional-elevator (C-SCAN) algorithm that exploits the order in which requests are inserted in and removed from the lists. The request lists are maintained in sorted order of increasing starting-sector number. When a request is accepted for processing by a block-device driver, it is not removed from the list. It is removed only after the I/O is complete, at which point the driver continues with the next request in the list, even if new requests have been inserted in the list before the active

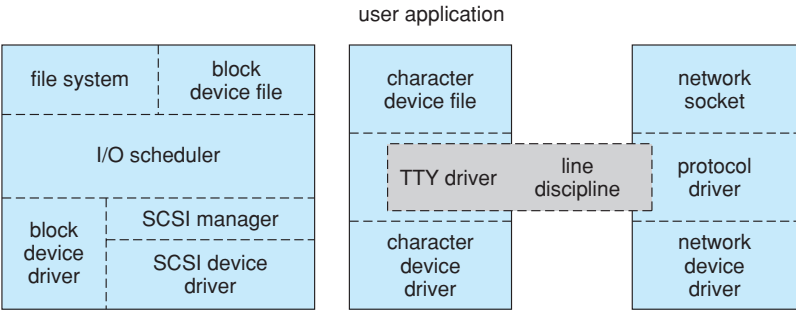


Figure 18.8 Device-driver block structure.

request. As new I/O requests are made, the request manager attempts to merge requests in the lists.

Linux kernel version 2.6 introduced a new I/O scheduling algorithm. Although a simple elevator algorithm remains available, the default I/O scheduler is now the **Completely Fair Queueing (CFQ)** scheduler. The CFQ I/O scheduler is fundamentally different from elevator-based algorithms. Instead of sorting requests into a list, CFQ maintains a set of lists—by default, one for each process. Requests originating from a process go in that process's list. For example, if two processes are issuing I/O requests, CFQ will maintain two separate lists of requests, one for each process. The lists are maintained according to the C-SCAN algorithm.

CFQ services the lists differently as well. Where a traditional C-SCAN algorithm is indifferent to a specific process, CFQ services each process's list round-robin. It pulls a configurable number of requests (by default, four) from each list before moving on to the next. This method results in fairness at the process level—each process receives an equal fraction of the disk's bandwidth. The result is beneficial with interactive workloads where I/O latency is important. In practice, however, CFQ performs well with most workloads.

18.8.2 Character Devices

A character-device driver can be almost any device driver that does not offer random access to fixed blocks of data. Any character-device drivers registered to the Linux kernel must also register a set of functions that implement the file I/O operations that the driver can handle. The kernel performs almost no preprocessing of a file read or write request to a character device. It simply passes the request to the device in question and lets the device deal with the request.

The main exception to this rule is the special subset of character-device drivers that implement terminal devices. The kernel maintains a standard interface to these drivers by means of a set of `tty_struct` structures. Each of these structures provides buffering and flow control on the data stream from the terminal device and feeds those data to a line discipline.

A **line discipline** is an interpreter for the information from the terminal device. The most common line discipline is the `tty` discipline, which glues the terminal's data stream onto the standard input and output streams of a user's running processes, allowing those processes to communicate directly with the user's terminal. This job is complicated by the fact that several such processes may be running simultaneously, and the `tty` line discipline is responsible for attaching and detaching the terminal's input and output from the various processes connected to it as those processes are suspended or awakened by the user.

Other line disciplines also are implemented that have nothing to do with I/O to a user process. The PPP and SLIP networking protocols are ways of encoding a networking connection over a terminal device such as a serial line. These protocols are implemented under Linux as drivers that at one end appear to the terminal system as line disciplines and at the other end appear to the networking system as network-device drivers. After one of these line disciplines has been enabled on a terminal device, any data appearing on that terminal will be routed directly to the appropriate network-device driver.

18.9 Interprocess Communication

Linux provides a rich environment for processes to communicate with each other. Communication may be just a matter of letting another process know that some event has occurred, or it may involve transferring data from one process to another.

18.9.1 Synchronization and Signals

The standard Linux mechanism for informing a process that an event has occurred is the **signal**. Signals can be sent from any process to any other process, with restrictions on signals sent to processes owned by another user. However, a limited number of signals are available, and they cannot carry information. Only the fact that a signal has occurred is available to a process. Signals are not generated only by processes. The kernel also generates signals internally. For example, it can send a signal to a server process when data arrive on a network channel, to a parent process when a child terminates, or to a waiting process when a timer expires.

Internally, the Linux kernel does not use signals to communicate with processes running in kernel mode. If a kernel-mode process is expecting an event to occur, it will not use signals to receive notification of that event. Rather, communication about incoming asynchronous events within the kernel takes place through the use of scheduling states and `wait_queue` structures. These mechanisms allow kernel-mode processes to inform one another about relevant events, and they also allow events to be generated by device drivers or by the networking system. Whenever a process wants to wait for some event to complete, it places itself on a wait queue associated with that event and tells the scheduler that it is no longer eligible for execution. Once the event has completed, every process on the wait queue will be awoken. This procedure allows multiple processes to wait for a single event. For example, if several processes are trying to read a file from a disk, then they will all be awakened once the data have been read into memory successfully.

Although signals have always been the main mechanism for communicating asynchronous events among processes, Linux also implements the semaphore mechanism of System V UNIX. A process can wait on a semaphore as easily as it can wait for a signal, but semaphores have two advantages: large numbers of semaphores can be shared among multiple independent processes, and operations on multiple semaphores can be performed atomically. Internally, the standard Linux wait queue mechanism synchronizes processes that are communicating with semaphores.

18.9.2 Passing of Data among Processes

Linux offers several mechanisms for passing data among processes. The standard UNIX **pipe** mechanism allows a child process to inherit a communication channel from its parent; data written to one end of the pipe can be read at the other. Under Linux, pipes appear as just another type of inode to virtual file system software, and each pipe has a pair of wait queues to synchronize the reader and writer. UNIX also defines a set of networking facilities that can send streams of data to both local and remote processes. Networking is covered in Section 18.10.

Another process communications method, shared memory, offers an extremely fast way to communicate large or small amounts of data. Any data written by one process to a shared memory region can be read immediately by any other process that has mapped that region into its address space. The main disadvantage of shared memory is that, on its own, it offers no synchronization. A process can neither ask the operating system whether a piece of shared memory has been written to nor suspend execution until such a write occurs. Shared memory becomes particularly powerful when used in conjunction with another interprocess-communication mechanism that provides the missing synchronization.

A shared-memory region in Linux is a persistent object that can be created or deleted by processes. Such an object is treated as though it were a small, independent address space. The Linux paging algorithms can elect to page shared-memory pages out to disk, just as they can page out a process's data pages. The shared-memory object acts as a backing store for shared-memory regions, just as a file can act as a backing store for a memory-mapped memory region. When a file is mapped into a virtual address space region, then any page faults that occur cause the appropriate page of the file to be mapped into virtual memory. Similarly, shared-memory mappings direct page faults to map in pages from a persistent shared-memory object. Also just as for files, shared-memory objects remember their contents even if no processes are currently mapping them into virtual memory.

18.10 Network Structure

Networking is a key area of functionality for Linux. Not only does Linux support the standard Internet protocols used for most UNIX-to-UNIX communications, but it also implements a number of protocols native to other, non-UNIX operating systems. In particular, since Linux was originally implemented primarily on PCs, rather than on large workstations or on server-class systems, it supports many of the protocols typically used on PC networks, such as AppleTalk and IPX.

Internally, networking in the Linux kernel is implemented by three layers of software:

1. The socket interface
2. Protocol drivers
3. Network-device drivers

User applications perform all networking requests through the socket interface. This interface is designed to look like the 4.3 BSD socket layer, so that any programs designed to make use of Berkeley sockets will run on Linux without any source-code changes. This interface is described in Section A.9.1. The BSD socket interface is sufficiently general to represent network addresses for a wide range of networking protocols. This single interface is used in Linux to access not just those protocols implemented on standard BSD systems but all the protocols supported by the system.

The next layer of software is the protocol stack, which is similar in organization to BSD's own framework. Whenever any networking data arrive at this layer, either from an application's socket or from a network-device driver, the data are expected to have been tagged with an identifier specifying which network protocol they contain. Protocols can communicate with one another if they desire; for example, within the Internet protocol set, separate protocols manage routing, error reporting, and reliable retransmission of lost data.

The protocol layer may rewrite packets, create new packets, split or reassemble packets into fragments, or simply discard incoming data. Ultimately, once the protocol layer has finished processing a set of packets, it passes them on, either upward to the socket interface if the data are destined for a local connection or downward to a device driver if the data need to be transmitted remotely. The protocol layer decides to which socket or device it will send the packet.

All communication between the layers of the networking stack is performed by passing single `skbuff` (socket buffer) structures. Each of these structures contains a set of pointers into a single continuous area of memory, representing a buffer inside which network packets can be constructed. The valid data in a `skbuff` do not need to start at the beginning of the `skbuff`'s buffer, and they do not need to run to the end. The networking code can add data to or trim data from either end of the packet, as long as the result still fits into the `skbuff`. This capacity is especially important on modern microprocessors, where improvements in CPU speed have far outstripped the performance of main memory. The `skbuff` architecture allows flexibility in manipulating packet headers and checksums while avoiding any unnecessary data copying.

The most important set of protocols in the Linux networking system is the TCP/IP protocol suite. This suite comprises a number of separate protocols. The IP protocol implements routing between different hosts anywhere on the network. On top of the routing protocol are the UDP, TCP, and ICMP protocols. The UDP protocol carries arbitrary individual datagrams between hosts. The TCP protocol implements reliable connections between hosts with guaranteed in-order delivery of packets and automatic retransmission of lost data. The ICMP protocol carries various error and status messages between hosts.

Each packet (`skbuff`) arriving at the networking stack's protocol software is expected to be already tagged with an internal identifier indicating the protocol to which the packet is relevant. Different networking-device drivers encode the protocol type in different ways; thus, the protocol for incoming data must be identified in the device driver. The device driver uses a hash table of known networking-protocol identifiers to look up the appropriate protocol and passes the packet to that protocol. New protocols can be added to the hash table as kernel-loadable modules.

Incoming IP packets are delivered to the IP driver. The job of this layer is to perform routing. After deciding where the packet is to be sent, the IP driver forwards the packet to the appropriate internal protocol driver to be delivered locally or injects it back into a selected network-device-driver queue to be forwarded to another host. It performs the routing decision using two tables: the persistent forwarding information base (FIB) and a cache of recent routing decisions. The FIB holds routing-configuration information and can specify routes based either on a specific destination address or on a wildcard

representing multiple destinations. The FIB is organized as a set of hash tables indexed by destination address; the tables representing the most specific routes are always searched first. Successful lookups from this table are added to the route-caching table, which caches routes only by specific destination. No wildcards are stored in the cache, so lookups can be made quickly. An entry in the route cache expires after a fixed period with no hits.

At various stages, the IP software passes packets to a separate section of code for **firewall management**—selective filtering of packets according to arbitrary criteria, usually for security purposes. The firewall manager maintains a number of separate **firewall chains** and allows a `skbuff` to be matched against any chain. Chains are reserved for separate purposes: one is used for forwarded packets, one for packets being input to this host, and one for data generated at this host. Each chain is held as an ordered list of rules, where a rule specifies one of a number of possible firewall-decision functions plus some arbitrary data for matching purposes.

Two other functions performed by the IP driver are disassembly and reassembly of large packets. If an outgoing packet is too large to be queued to a device, it is simply split up into smaller **fragments**, which are all queued to the driver. At the receiving host, these fragments must be reassembled. The IP driver maintains an `ipfrag` object for each fragment awaiting reassembly and an `ipq` for each datagram being assembled. Incoming fragments are matched against each known `ipq`. If a match is found, the fragment is added to it; otherwise, a new `ipq` is created. Once the final fragment has arrived for a `ipq`, a completely new `skbuff` is constructed to hold the new packet, and this packet is passed back into the IP driver.

Packets identified by the IP as destined for this host are passed on to one of the other protocol drivers. The UDP and TCP protocols share a means of associating packets with source and destination sockets: each connected pair of sockets is uniquely identified by its source and destination addresses and by the source and destination port numbers. The socket lists are linked to hash tables keyed on these four address and port values for socket lookup on incoming packets. The TCP protocol has to deal with unreliable connections, so it maintains ordered lists of unacknowledged outgoing packets to retransmit after a timeout and of incoming out-of-order packets to be presented to the socket when the missing data have arrived.

18.11 Security

Linux's security model is closely related to typical UNIX security mechanisms. The security concerns can be classified in two groups:

1. **Authentication.** Making sure that nobody can access the system without first proving that she has entry rights
2. **Access control.** Providing a mechanism for checking whether a user has the right to access a certain object and preventing access to objects as required

18.11.1 Authentication

Authentication in UNIX has typically been performed through the use of a publicly readable password file. A user's password is combined with a random "salt" value, and the result is encoded with a one-way transformation function and stored in the password file. The use of the one-way function means that the original password cannot be deduced from the password file except by trial and error. When a user presents a password to the system, the password is recombined with the salt value stored in the password file and passed through the same one-way transformation. If the result matches the contents of the password file, then the password is accepted.

Historically, UNIX implementations of this mechanism have had several drawbacks. Passwords were often limited to eight characters, and the number of possible salt values was so low that an attacker could easily combine a dictionary of commonly used passwords with every possible salt value and have a good chance of matching one or more passwords in the password file, gaining unauthorized access to any accounts compromised as a result. Extensions to the password mechanism have been introduced that keep the encrypted password secret in a file that is not publicly readable, that allow longer passwords, or that use more secure methods of encoding the password. Other authentication mechanisms have been introduced that limit the periods during which a user is permitted to connect to the system. Also, mechanisms exist to distribute authentication information to all the related systems in a network.

A new security mechanism has been developed by UNIX vendors to address authentication problems. The **pluggable authentication modules (PAM)** system is based on a shared library that can be used by any system component that needs to authenticate users. An implementation of this system is available under Linux. PAM allows authentication modules to be loaded on demand as specified in a system-wide configuration file. If a new authentication mechanism is added at a later date, it can be added to the configuration file, and all system components will immediately be able to take advantage of it. PAM modules can specify authentication methods, account restrictions, session-setup functions, and password-changing functions (so that, when users change their passwords, all the necessary authentication mechanisms can be updated at once).

18.11.2 Access Control

Access control under UNIX systems, including Linux, is performed through the use of unique numeric identifiers. A user identifier (UID) identifies a single user or a single set of access rights. A group identifier (GID) is an extra identifier that can be used to identify rights belonging to more than one user.

Access control is applied to various objects in the system. Every file available in the system is protected by the standard access-control mechanism. In addition, other shared objects, such as shared-memory sections and semaphores, employ the same access system.

Every object in a UNIX system under user and group access control has a single UID and a single GID associated with it. User processes also have a single UID, but they may have more than one GID. If a process's UID matches the UID of an object, then the process has **user rights** or **owner rights** to that object.

If the UIDs do not match but any GID of the process matches the object's GID, then **group rights** are conferred; otherwise, the process has **world rights** to the object.

Linux performs access control by assigning objects a **protection mask** that specifies which access modes—read, write, or execute—are to be granted to processes with owner, group, or world access. Thus, the owner of an object might have full read, write, and execute access to a file; other users in a certain group might be given read access but denied write access; and everybody else might be given no access at all.

The only exception is the privileged **root** UID. A process with this special UID is granted automatic access to any object in the system, bypassing normal access checks. Such processes are also granted permission to perform privileged operations, such as reading any physical memory or opening reserved network sockets. This mechanism allows the kernel to prevent normal users from accessing these resources: most of the kernel's key internal resources are implicitly owned by the root UID.

Linux implements the standard UNIX **setuid** mechanism described in Section A.3.2. This mechanism allows a program to run with privileges different from those of the user running the program. For example, the **lpr** program (which submits a job to a print queue) has access to the system's print queues even if the user running that program does not. The UNIX implementation of **setuid** distinguishes between a process's real and effective UID. The real UID is that of the user running the program; the effective UID is that of the file's owner.

Under Linux, this mechanism is augmented in two ways. First, Linux implements the POSIX specification's **saved user-id** mechanism, which allows a process to drop and reacquire its effective UID repeatedly. For security reasons, a program may want to perform most of its operations in a safe mode, waiving the privileges granted by its **setuid** status; but it may wish to perform selected operations with all its privileges. Standard UNIX implementations achieve this capacity only by swapping the real and effective UIDs. When this is done, the previous effective UID is remembered, but the program's real UID does not always correspond to the UID of the user running the program. Saved UIDs allow a process to set its effective UID to its real UID and then return to the previous value of its effective UID without having to modify the real UID at any time.

The second enhancement provided by Linux is the addition of a process characteristic that grants just a subset of the rights of the effective UID. The **fsuid** and **fsgid** process properties are used when access rights are granted to files. The appropriate property is set every time the effective UID or GID is set. However, the **fsuid** and **fsgid** can be set independently of the effective ids, allowing a process to access files on behalf of another user without taking on the identity of that other user in any other way. Specifically, server processes can use this mechanism to serve files to a certain user without becoming vulnerable to being killed or suspended by that user.

Finally, Linux provides a mechanism for flexible passing of rights from one program to another—a mechanism that has become common in modern versions of UNIX. When a local network socket has been set up between any two processes on the system, either of those processes may send to the other process a file descriptor for one of its open files; the other process receives a

duplicate file descriptor for the same file. This mechanism allows a client to pass access to a single file selectively to some server process without granting that process any other privileges. For example, it is no longer necessary for a print server to be able to read all the files of a user who submits a new print job. The print client can simply pass the server file descriptors for any files to be printed, denying the server access to any of the user's other files.

18.12 Summary

Linux is a modern, free operating system based on UNIX standards. It has been designed to run efficiently and reliably on common PC hardware; it also runs on a variety of other platforms, such as mobile phones. It provides a programming interface and user interface compatible with standard UNIX systems and can run a large number of UNIX applications, including an increasing number of commercially supported applications.

Linux has not evolved in a vacuum. A complete Linux system includes many components that were developed independently of Linux. The core Linux operating-system kernel is entirely original, but it allows much existing free UNIX software to run, resulting in an entire UNIX-compatible operating system free from proprietary code.

The Linux kernel is implemented as a traditional monolithic kernel for performance reasons, but it is modular enough in design to allow most drivers to be dynamically loaded and unloaded at run time.

Linux is a multiuser system, providing protection between processes and running multiple processes according to a time-sharing scheduler. Newly created processes can share selective parts of their execution environment with their parent processes, allowing multithreaded programming. Interprocess communication is supported by both System V mechanisms—message queues, semaphores, and shared memory—and BSD's socket interface. Multiple networking protocols can be accessed simultaneously through the socket interface.

The memory-management system uses page sharing and copy-on-write to minimize the duplication of data shared by different processes. Pages are loaded on demand when they are first referenced and are paged back out to backing store according to an LFU algorithm if physical memory needs to be reclaimed.

To the user, the file system appears as a hierarchical directory tree that obeys UNIX semantics. Internally, Linux uses an abstraction layer to manage multiple file systems. Device-oriented, networked, and virtual file systems are supported. Device-oriented file systems access disk storage through a page cache that is unified with the virtual memory system.

Practice Exercises

- 18.1 Dynamically loadable kernel modules give flexibility when drivers are added to a system, but do they have disadvantages too? Under what circumstances would a kernel be compiled into a single binary file, and when would it be better to keep it split into modules? Explain your answer.

- 18.2 Multithreading is a commonly used programming technique. Describe three different ways to implement threads, and compare these three methods with the Linux `clone()` mechanism. When might using each alternative mechanism be better or worse than using clones?
- 18.3 The Linux kernel does not allow paging out of kernel memory. What effect does this restriction have on the kernel's design? What are two advantages and two disadvantages of this design decision?
- 18.4 Discuss three advantages of dynamic (shared) linkage of libraries compared with static linkage. Describe two cases in which static linkage is preferable.
- 18.5 Compare the use of networking sockets with the use of shared memory as a mechanism for communicating data between processes on a single computer. What are the advantages of each method? When might each be preferred?
- 18.6 At one time, UNIX systems used disk-layout optimizations based on the rotation position of disk data, but modern implementations, including Linux, simply optimize for sequential data access. Why do they do so? Of what hardware characteristics does sequential access take advantage? Why is rotational optimization no longer so useful?

Exercises

- 18.7 What are the advantages and disadvantages of writing an operating system in a high-level language, such as C?
- 18.8 In what circumstances is the system-call sequence `fork()` `exec()` most appropriate? When is `vfork()` preferable?
- 18.9 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.
- 18.10 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?
- 18.11 What are the advantages and disadvantages of making only some of the symbols defined inside a kernel accessible to a loadable kernel module?
- 18.12 What are the primary goals of the conflict-resolution mechanism used by the Linux kernel for loading kernel modules?
- 18.13 Discuss how the `clone()` operation supported by Linux is used to support both processes and threads.
- 18.14 Would you classify Linux threads as user-level threads or as kernel-level threads? Support your answer with the appropriate arguments.
- 18.15 What extra costs are incurred in the creation and scheduling of a process, compared with the cost of a cloned thread?

- 18.16 How does Linux's Completely Fair Scheduler (CFS) provide improved fairness over a traditional UNIX process scheduler? When is the fairness guaranteed?
- 18.17 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?
- 18.18 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?
- 18.19 Under what circumstances would a user process request an operation that results in the allocation of a demand-zero memory region?
- 18.20 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?
- 18.21 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.
- 18.22 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?
- 18.23 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?
- 18.24 In what ways does the Linux `setuid` feature differ from the `setuid` feature SVR4?
- 18.25 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?

Bibliographical Notes

The Linux system is a product of the Internet; as a result, much of the available documentation on Linux is available in some form on the Internet. The following key sites reference most of the useful information available:

- The *Linux Cross-Reference Page (LXR)* (<http://lxr.linux.no>) maintains current listings of the Linux kernel, browsable via the Web and fully cross-referenced.
- The *Kernel Hackers' Guide* provides a helpful overview of the Linux kernel components and internals and is located at <http://tldp.org/LDP/tlk/tlk.html>.

- The *Linux Weekly News (LWN)* (<http://lwn.net>) provides weekly Linux-related news, including a very well researched subsection on Linux kernel news.

Many mailing lists devoted to Linux are also available. The most important are maintained by a mailing-list manager that can be reached at the e-mail address `majordomo@vger.rutgers.edu`. Send e-mail to this address with the single line “help” in the mail’s body for information on how to access the list server and to subscribe to any lists.

Finally, the Linux system itself can be obtained over the Internet. Complete Linux distributions are available from the home sites of the companies concerned, and the Linux community also maintains archives of current system components at several places on the Internet. The most important is `ftp://ftp.kernel.org/pub/linux`.

In addition to investigating Internet resources, you can read about the internals of the Linux kernel in [Mauerer (2008)] and [Love (2010)].

Bibliography

[Love (2010)] R. Love, *Linux Kernel Development*, Third Edition, Developer’s Library (2010).

[Mauerer (2008)] W. Mauerer, *Professional Linux Kernel Architecture*, John Wiley and Sons (2008).

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 chapter, 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

- To explore the principles underlying Windows 7's design and the specific components of the system.
- To provide a detailed discussion of the Windows 7 file system.
- To illustrate the networking protocols supported in Windows 7.
- To describe the interface available in Windows 7 to system and application programmers.
- To describe the important algorithms implemented with Windows 7.

19.1 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.

19.2 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.

19.2.1 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 classifica-

tion 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**, as described in Section 15.8.) 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 11.6.2). 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.

19.2.2 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.

19.2.3 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 higher 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.

19.2.4 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 6).

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.

19.2.5 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.

19.2.6 Portability

An operating system is **portable** if it can be moved from one CPU architecture to another with relatively 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.

19.2.7 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.

19.2.8 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.

19.2.9 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 PCMCIA 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 enable 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.

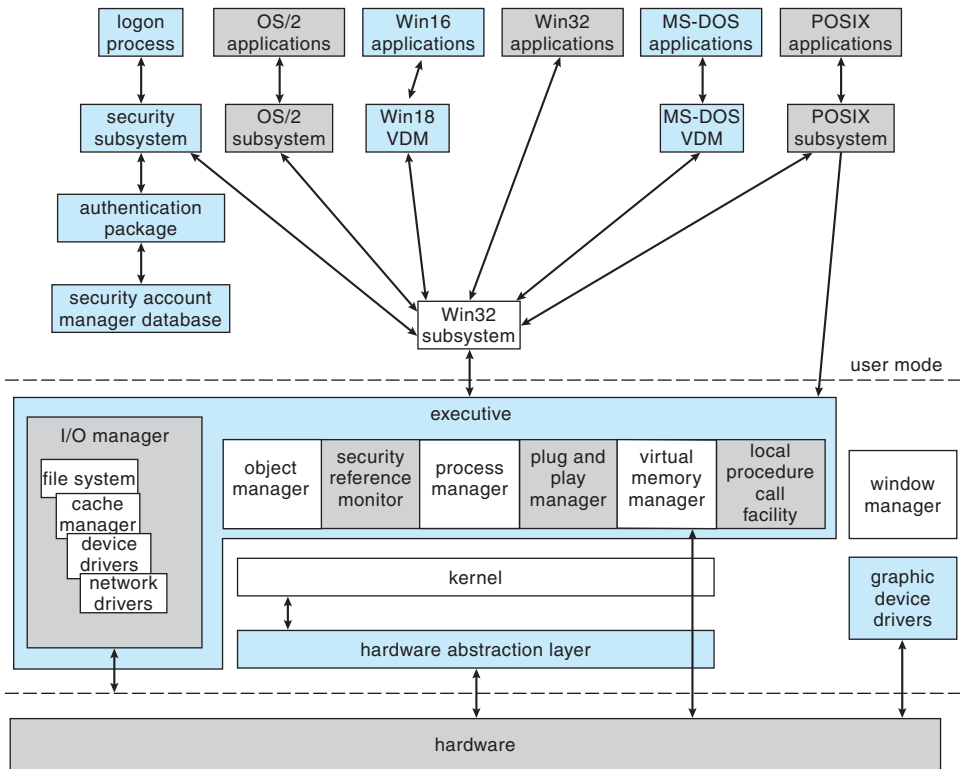


Figure 19.1 Windows block diagram.

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

19.3 System Components

The architecture of Windows is a layered system of modules, as shown in Figure 19.1. 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.

19.3.1 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.

19.3.2 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.

19.3.2.1 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.

19.3.2.2 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 further discussed 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.

19.3.2.3 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 19.7.1).

19.3.2.4 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*.

DPCs are 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.

19.3.2.5 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 19.3.3.4.) 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 19.2.

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 19.2 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.

19.3.2.6 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 19.7.3.7.

19.3.3 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.

19.3.3.1 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.

19.3.3.2 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 19.3 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,

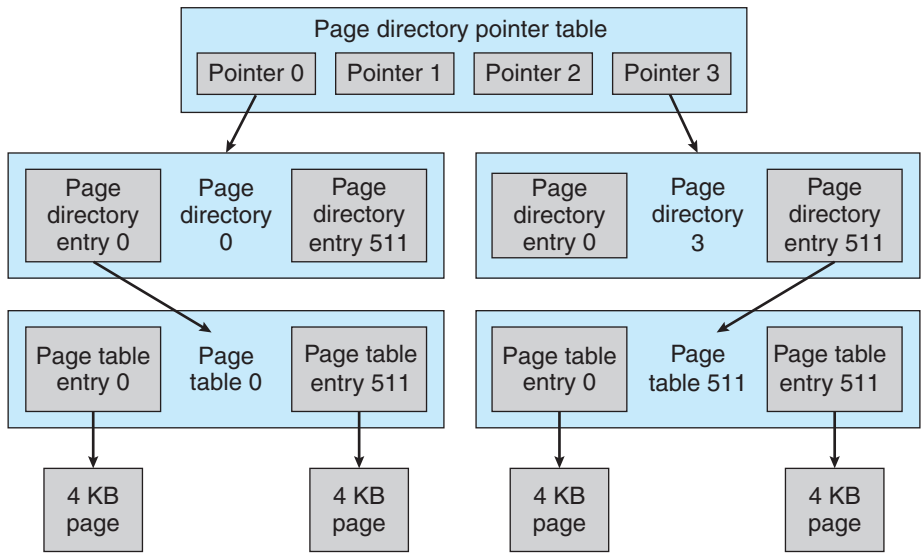


Figure 19.3 Page-table layout.

containing 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 19.4, 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.

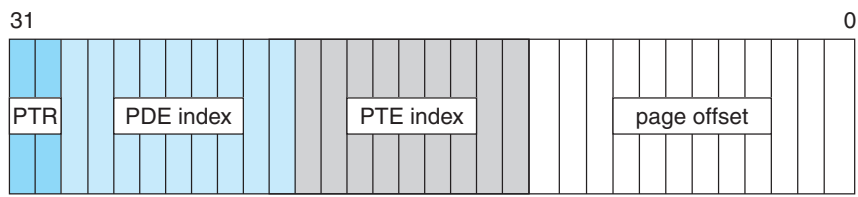


Figure 19.4 Virtual-to-physical address translation on IA-32.

- 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

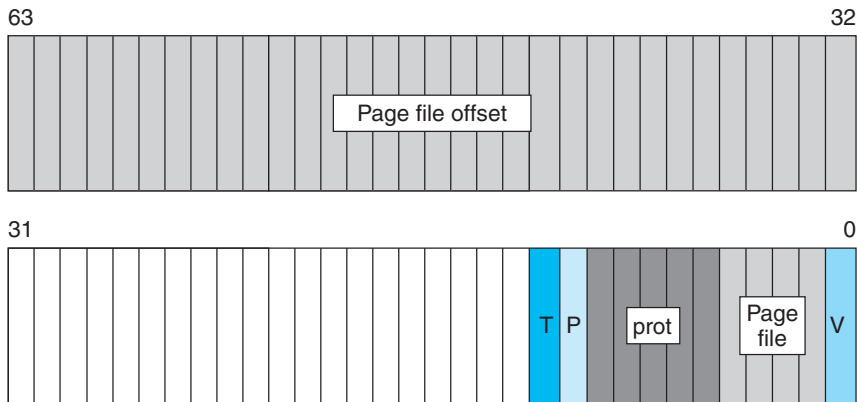


Figure 19.5 Page-file page-table entry. The valid bit is zero.

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 19.5. 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.

19.3.3.3 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.

19.3.3.4 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, network

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 19.3.3.3).

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 19.6.2.7. 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.

19.3.3.5 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.

19.3.3.6 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 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 19.6 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

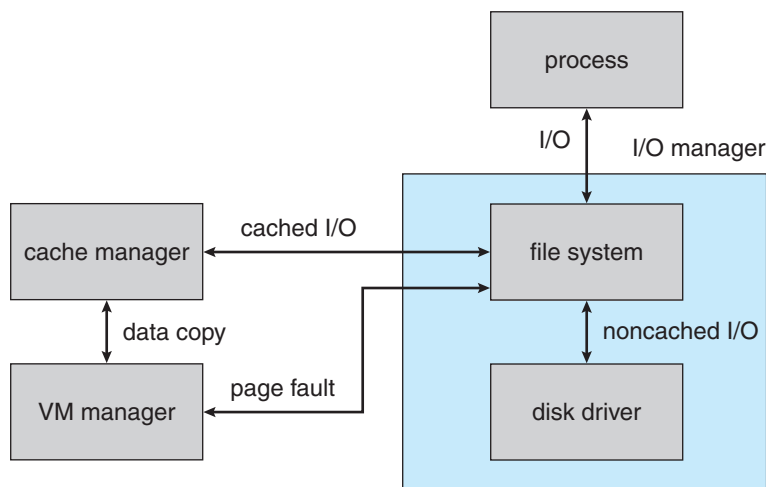


Figure 19.6 File I/O.

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.

19.3.3.7 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.

19.3.3.8 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.

19.3.3.9 Power Manager

Windows works with the hardware to implement sophisticated strategies for energy efficiency, as described in Section 19.2.8. 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.

19.3.3.10 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.

19.3.3.11 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.

19.4 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.

19.5 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.

19.5.1 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.

19.5.1.1 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.

19.5.1.2 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 19.3.3.6, 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.

19.5.2 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.

19.5.3 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.

19.5.4 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.

19.5.4.1 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 19.7. 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 10.7.) 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.

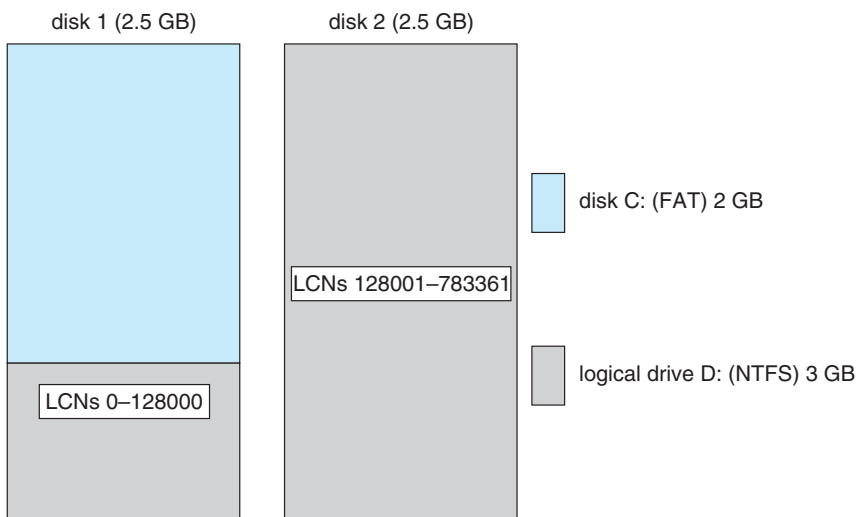


Figure 19.7 Volume set on two drives.

19.5.4.2 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.

19.5.5 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.

19.5.6 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.

19.5.7 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.

19.5.8 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.

19.6 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.

19.6.1 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.

19.6.2 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.

19.6.2.1 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.

19.6.2.2 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 (SNMP), 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.

19.6.2.3 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.

19.6.2.4 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, which Windows makes available as a transport option for implementing RPC.

19.6.2.5 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.

19.6.2.6 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 named pipes 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.

19.6.2.7 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.

19.6.2.8 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.

19.6.3 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 19.3.3.5.
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 19.6.2. The list of redirectors is maintained in the system hive of the registry.

19.6.3.1 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.

19.6.3.2 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.

19.6.4 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.

19.6.5 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, uniformity drastically reduces the cost of computing.

19.7 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.

19.7.1 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.

19.7.2 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 SECURITY_ATTRIBUTES structure with the bInheritHandle field set to TRUE. This field creates an inheritable handle. Next, the child process is created, passing a value of TRUE to the CreateProcess() function's bInheritHandle argument. Figure 19.8 shows a code sample that creates a semaphore handle inherited by a child process.

Assuming the child process knows which handles are shared, the parent and child can achieve interprocess communication through the shared objects. In the example in Figure 19.8, 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


```

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];
ostringstream ostring(command_line, sizeof(command_line));
ostring << a_semaphore << ends;
CreateProcess("another_process.exe", command_line,
             NULL, NULL, TRUE, . . .);

```

Figure 19.8 Code enabling a child to share an object by inheriting a handle.

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 19.9.

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 19.10.

19.7.3 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
. . .
HANDLE a_semaphore = CreateSemaphore(NULL, 1, 1, "MySEM1");
. . .

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

```

Figure 19.9 Code for sharing an object by name lookup.

```

// 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 19.10 Code for sharing an object by passing a handle.

19.7.3.1 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.

19.7.3.2 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 19.3.2.2 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 19.3.2.2, 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.

19.7.3.3 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.

19.7.3.4 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 19.3.2.2. 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.

19.7.3.5 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.

19.7.3.6 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.

19.7.3.7 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 constructs, as well as rudimentary resource management and task synchronization primitives. The key features of UMS are depicted in Figure 19.11.

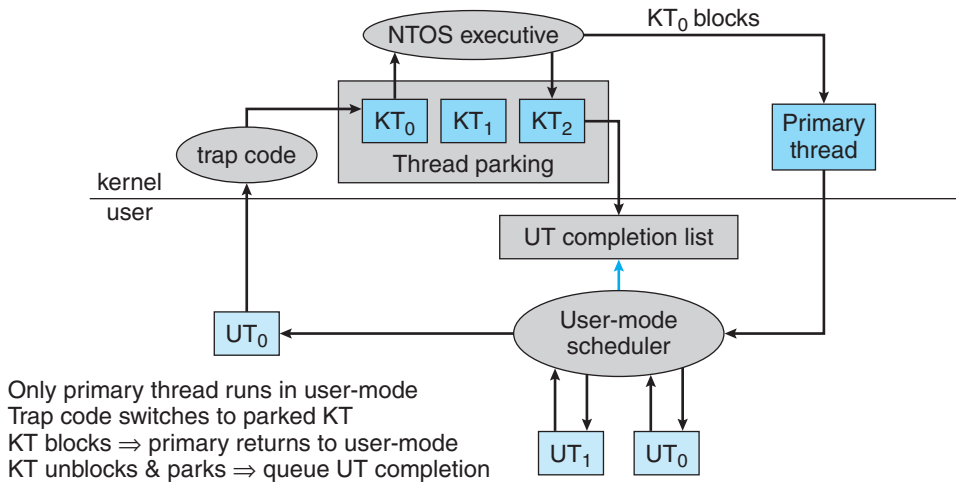


Figure 19.11 User-mode scheduling.

19.7.3.8 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.

19.7.4 Interprocess Communication 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 is actually delivered. The send routines are synchronous: they block the caller until the message has been delivered and processed.


```

// 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 19.12 Code fragments for allocating virtual memory.

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”.

19.7.5 Memory Management

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

19.7.5.1 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 19.12.

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.

19.7.5.2 Memory-Mapping Files

Another way for an application to use memory is by memory-mapping a file into its address space. Memory mapping is also a convenient way for two processes to share memory: both processes map the same file into their virtual memory. Memory mapping is a multistage process, as you can see in the example in Figure 19.13.

```
// 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 19.13 Code fragments for memory mapping of a file.

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.

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

19.7.5.4 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

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

Figure 19.14 Code for dynamic thread-local storage.

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 19.14. 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;
```

19.8 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

- 19.1 What type of operating system is Windows? Describe two of its major features.
- 19.2 List the design goals of Windows. Describe two in detail.

- 19.3 Describe the booting process for a Windows system.
- 19.4 Describe the three main architectural layers of the Windows kernel.
- 19.5 What is the job of the object manager?
- 19.6 What types of services does the process manager provide?
- 19.7 What is a local procedure call?
- 19.8 What are the responsibilities of the I/O manager?
- 19.9 What types of networking does Windows support? How does Windows implement transport protocols? Describe two networking protocols.
- 19.10 How is the NTFS namespace organized?
- 19.11 How does NTFS handle data structures? How does NTFS recover from a system crash? What is guaranteed after a recovery takes place? ‘
- 19.12 How does Windows allocate user memory?
- 19.13 Describe some of the ways in which an application can use memory via the Win32 API.

Exercises

- 19.14 Under what circumstances would one use the deferred procedure calls facility in Windows?
- 19.15 What is a handle, and how does a process obtain a handle?
- 19.16 Describe the management scheme of the virtual memory manager. How does the VM manager improve performance?
- 19.17 Describe a useful application of the no-access page facility provided in Windows.
- 19.18 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?
- 19.19 What manages caching in Windows? How is the cache managed?
- 19.20 How does the NTFS directory structure differ from the directory structure used in UNIX operating systems?
- 19.21 What is a process, and how is it managed in Windows?
- 19.22 What is the fiber abstraction provided by Windows? How does it differ from the thread abstraction?
- 19.23 How does user-mode scheduling (UMS) in Windows 7 differ from fibers? What are some trade-offs between fibers and UMS?
- 19.24 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?
- 19.25 What is the performance trade-off of allowing KTs and UTs to execute on different processors?

- 19.26 Why does the self-map occupy large amounts of virtual address space but no additional virtual memory?
- 19.27 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?
- 19.28 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?
- 19.29 Give an example showing how the use of a suspend count is helpful in suspending and resuming threads in Windows.

Bibliographical Notes

[Russinovich and Solomon (2009)] give 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)]. [Silberschatz et al. (2010)] supply a good discussion of B+ trees.

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 and Solomon (2009)] M. E. Russinovich and D. A. Solomon, *Windows Internals: Including Windows Server 2008 and Windows Vista*, Fifth Edition, Microsoft Press (2009).
- [Silberschatz et al. (2010)] A. Silberschatz, H. F. Korth, and S. Sudarshan, *Database System Concepts*, Sixth Edition, McGraw-Hill (2010).

Influential Operating Systems



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

In the bibliographical notes at the end of the chapter, we include references to further reading about these early systems. The papers, written by the designers of the systems, are important both for their technical content and for their style and flavor.

CHAPTER OBJECTIVES

- To explain how operating-system features migrate over time from large computer systems to smaller ones.
- To discuss the features of several historically important operating systems.

20.1 Feature Migration

One reason to study early architectures and operating systems is that a feature that once ran only on huge systems may eventually make its way into very small systems. Indeed, an examination of operating systems for mainframes and microcomputers shows that many features once available only on mainframes have been adopted for microcomputers. The same operating-system concepts are thus appropriate for various classes of computers: mainframes, minicomputers, microcomputers, and handhelds. To understand modern operating systems, then, you need to recognize the theme of feature migration and the long history of many operating-system features, as shown in Figure 20.1.

A good example of feature migration started with the Multiplexed Information and Computing Services (MULTICS) operating system. MULTICS was

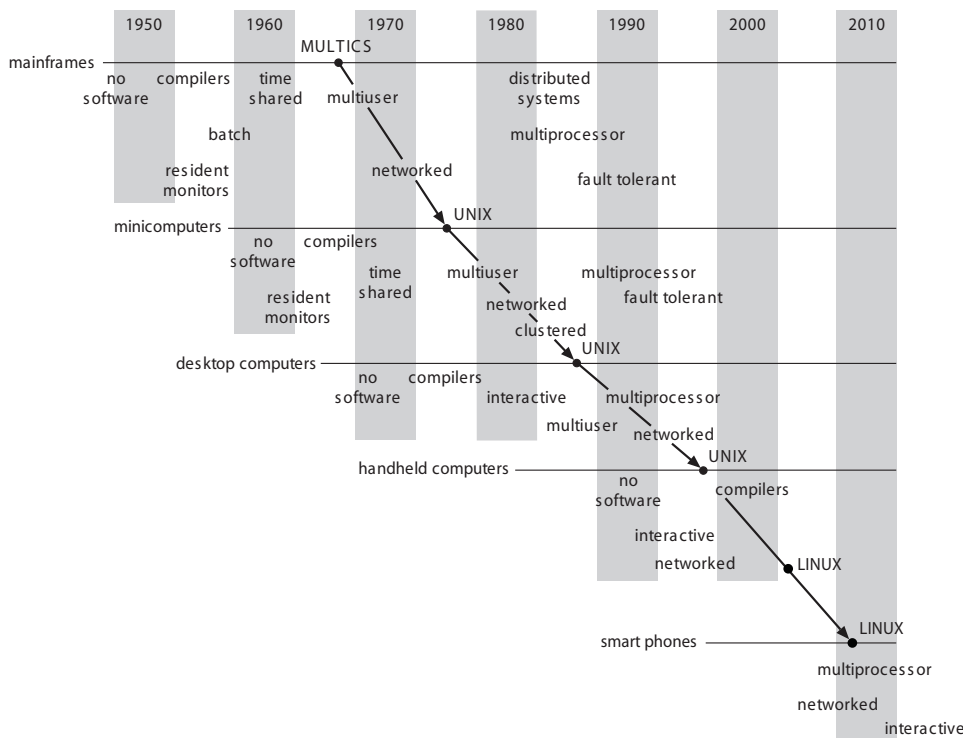


Figure 20.1 Migration of operating-system concepts and features.

developed from 1965 to 1970 at the Massachusetts Institute of Technology (MIT) as a computing **utility**. It ran on a large, complex mainframe computer (the GE 645). Many of the ideas that were developed for MULTICS were subsequently used at Bell Laboratories (one of the original partners in the development of MULTICS) in the design of UNIX. The UNIX operating system was designed around 1970 for a PDP-11 minicomputer. Around 1980, the features of UNIX became the basis for UNIX-like operating systems on microcomputers; and these features are included in several more recent operating systems for microcomputers, such as Microsoft Windows, Windows XP, and the Mac OS X operating system. Linux includes some of these same features, and they can now be found on PDAs.

20.2 Early Systems

We turn our attention now to a historical overview of early computer systems. We should note that the history of computing starts far before “computers” with looms and calculators. We begin our discussion, however, with the computers of the twentieth century.

Before the 1940s, computing devices were designed and implemented to perform specific, fixed tasks. Modifying one of those tasks required a great deal of effort and manual labor. All that changed in the 1940s when Alan Turing and John von Neumann (and colleagues), both separately and together, worked on the idea of a more general-purpose **stored program** computer. Such a machine

has both a program store and a data store, where the program store provides instructions about what to do to the data.

This fundamental computer concept quickly generated a number of general-purpose computers, but much of the history of these machines is blurred by time and the secrecy of their development during World War II. It is likely that the first working stored-program general-purpose computer was the Manchester Mark 1, which ran successfully in 1949. The first commercial computer—the Ferranti Mark 1, which went on sale in 1951—was its offspring.

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

20.2.1 Dedicated Computer Systems

As time went on, additional software and hardware were developed. Card readers, line printers, and magnetic tape became commonplace. Assemblers, loaders, and linkers were designed to ease the programming task. Libraries of common functions were created. Common functions could then be copied into a new program without having to be written again, providing software reusability.

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

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

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

1. Loading the FORTRAN compiler tape
2. Running the compiler
3. Unloading the compiler tape
4. Loading the assembler tape
5. Running the assembler
6. Unloading the assembler tape
7. Loading the object program
8. Running the object program

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

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

20.2.2 Shared Computer Systems

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

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

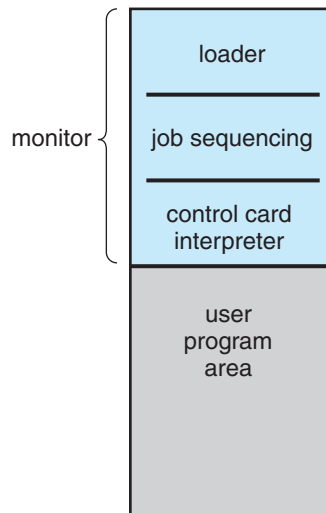


Figure 20.2 Memory layout for a resident monitor.

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

To overcome this idle time, people developed **automatic job sequencing**. With this technique, the first rudimentary operating systems were created. A small program, called a **resident monitor**, was created to transfer control automatically from one job to the next (Figure 20.2). The resident monitor is always in memory (or *resident*).

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

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

\$FTN—Execute the FORTRAN compiler.

\$ASM—Execute the assembler.

\$RUN—Execute the user program.

These cards tell the resident monitor which program to run.

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

\$JOB—First card of a job
\$END—Final card of a job

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

One problem with control cards is how to distinguish them from data or program cards. The usual solution is to identify them by a special character or pattern on the card. Several systems used the dollar-sign character (\$) in the first column to identify a control card. Others used a different code. IBM's Job Control Language (JCL) used slash marks (/ /) in the first two columns. Figure 20.3 shows a sample card-deck setup for a simple batch system.

A resident monitor thus has several identifiable parts:

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

These batch systems work fairly well. The resident monitor provides automatic job sequencing as indicated by the control cards. When a control card indicates that a program is to be run, the monitor loads the program into memory and transfers control to it. When the program completes, it

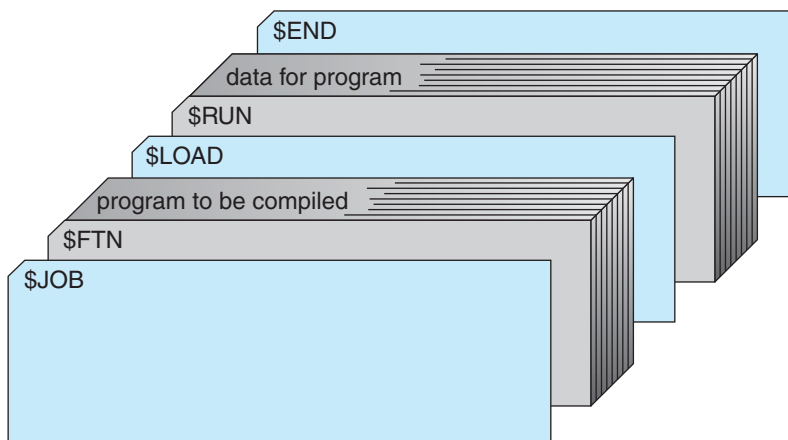


Figure 20.3 Card deck for a simple batch system.

transfers control back to the monitor, which reads the next control card, loads the appropriate program, and so on. This cycle is repeated until all control cards are interpreted for the job. Then the monitor automatically continues with the next job.

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

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

20.2.3 Overlapped I/O

One common solution to the I/O problem was to replace slow card readers (input devices) and line printers (output devices) with magnetic-tape units. Most computer systems in the late 1950s and early 1960s were batch systems reading from card readers and writing to line printers or card punches. The CPU did not read directly from cards, however; instead, the cards were first copied onto a magnetic tape via a separate device. When the tape was sufficiently full, it was taken down and carried over to the computer. When a card was needed for input to a program, the equivalent record was read from the tape. Similarly, output was written to the tape, and the contents of the tape were printed later. The card readers and line printers were operated *off-line*, rather than by the main computer (Figure 20.4).

An obvious advantage of off-line operation was that the main computer was no longer constrained by the speed of the card readers and line printers but was limited only by the speed of the much faster magnetic tape units.

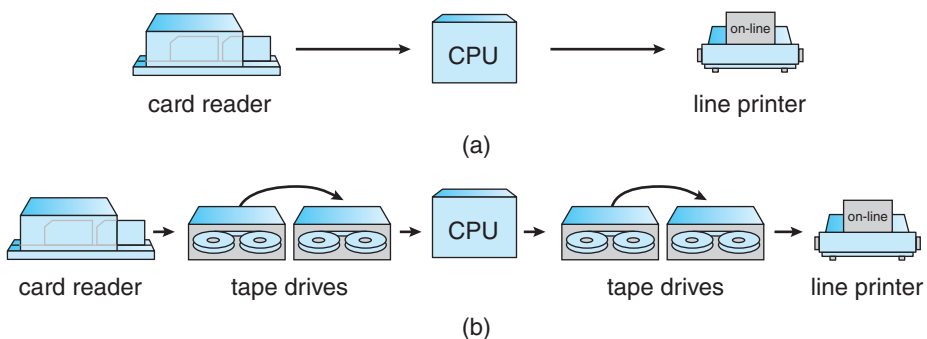


Figure 20.4 Operation of I/O devices (a) on-line and (b) off-line.

The technique of using magnetic tape for all I/O could be applied with any similar equipment (such as card readers, card punches, plotters, paper tape, and printers).

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

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

In a disk system, cards are read directly from the card reader onto the disk. The location of card images is recorded in a table kept by the operating system. When a job is executed, the operating system satisfies its requests for card-reader input by reading from the disk. Similarly, when the job requests the printer to output a line, that line is copied into a system buffer and is written to the disk. When the job is completed, the output is actually printed. This form of processing is called **spooling** (Figure 20.5); the name is an acronym for simultaneous peripheral operation on-line. Spooling, in essence, uses the disk

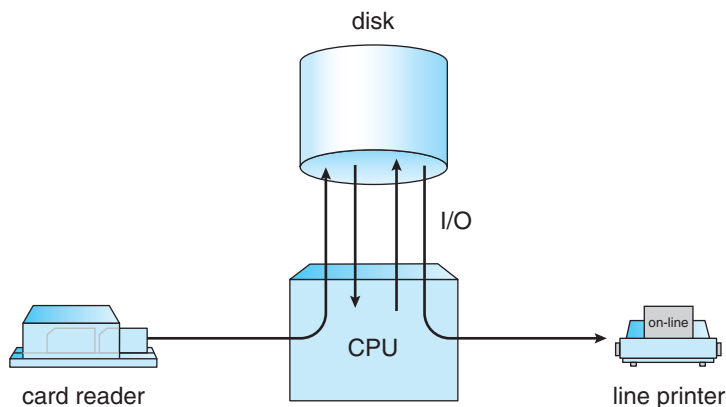


Figure 20.5 Spooling.

as a huge buffer for reading as far ahead as possible on input devices and for storing output files until the output devices are able to accept them.

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

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

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

20.3 Atlas

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

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

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

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

If a page fault occurred, a page-replacement algorithm was invoked. One memory frame was always kept empty, so that a drum transfer could start immediately. The page-replacement algorithm attempted to predict future memory-accessing behavior based on past behavior. A reference bit for each frame was set whenever the frame was accessed. The reference bits were read

into memory every 1,024 instructions, and the last 32 values of these bits were retained. This history was used to define the time since the most recent reference (t_1) and the interval between the last two references (t_2). Pages were chosen for replacement in the following order:

1. Any page with $t_1 > t_2 + 1$ is considered to be no longer in use and is replaced.
2. If $t_1 \leq t_2$ for all pages, then replace the page with the largest value for $t_2 - t_1$.

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

20.4 XDS-940

The XDS-940 operating system was designed at the University of California at Berkeley in the early 1960's. Like the Atlas system, it used paging for memory management. Unlike the Atlas system, it was a time-shared system. The paging was used only for relocation; it was not used for demand paging. The virtual memory of any user process was made up of 16-KB words, whereas the physical memory was made up of 64-KB words. Each page was made up of 2-KB words. The page table was kept in registers. Since physical memory was larger than virtual memory, several user processes could be in memory at the same time. The number of users could be increased by page sharing when the pages contained read-only reentrant code. Processes were kept on a drum and were swapped in and out of memory as necessary.

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

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

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

20.5 THE

The THE operating system was designed at the Technische Hogeschool in Eindhoven in the Netherlands in the mid-1960's. It was a batch system running on a Dutch computer, the EL X8, with 32 KB of 27-bit words. The system was mainly noted for its clean design, particularly its layer structure, and its use of a set of concurrent processes employing semaphores for synchronization.

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

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

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

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

Closely related to the THE system is the Venus system. The Venus system was also a layer-structured design, using semaphores to synchronize processes. The lower levels of the design were implemented in microcode, however, providing a much faster system. Paged-segmented memory was used for memory management. In addition, the system was designed as a time-sharing system, rather than a batch system.

20.6 RC 4000

The RC 4000 system, like the THE system, was notable primarily for its design concepts. It was designed in the late 1960's for the Danish 4000 computer by Regnecentralen, particularly by Brinch-Hansen. The objective was not to design a batch system, or a time-sharing system, or any other specific system. Rather, the goal was to create an operating-system nucleus, or kernel, on which a complete operating system could be built. Thus, the system structure was layered, and only the lower levels—comprising the kernel—were provided.

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

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

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

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

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

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

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

20.7 CTSS

The Compatible Time-Sharing System (CTSS) was designed at MIT as an experimental time-sharing system and first appeared in 1961. It was implemented on an IBM 7090 and eventually supported up to 32 interactive users. The users were provided with a set of interactive commands that allowed them to manipulate files and to compile and run programs through a terminal.

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

CTSS was extremely successful and was in use as late as 1972. Although it was limited, it succeeded in demonstrating that time sharing was a con-

venient and practical mode of computing. One result of CTSS was increased development of time-sharing systems. Another result was the development of MULTICS.

20.8 MULTICS

The MULTICS operating system was designed from 1965 to 1970 at MIT as a natural extension of CTSS. CTSS and other early time-sharing systems were so successful that they created an immediate desire to proceed quickly to bigger and better systems. As larger computers became available, the designers of CTSS set out to create a time-sharing utility. Computing service would be provided like electrical power. Large computer systems would be connected by telephone wires to terminals in offices and homes throughout a city. The operating system would be a time-shared system running continuously with a vast file system of shared programs and data.

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

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

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

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

20.9 IBM OS/360

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

The IBM/360 — which first appeared in the mid 1960's — was designed to alter this situation. The IBM/360 ([Mealy et al. (1966)]) was designed as a family of computers spanning the complete range from small business machines to large scientific machines. Only one set of software would be needed for these systems, which all used the same operating system: OS/360. This arrangement

was intended to reduce maintenance problems for IBM and to allow users to move programs and applications freely from one IBM system to another.

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

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

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

Virtual memory was added to OS/360 with the change to the IBM/370 architecture. The underlying hardware provided a segmented-paged virtual memory. New versions of OS used this hardware in different ways. OS/VS1 created one large virtual address space and ran OS/MFT in that virtual memory. Thus, the operating system itself was paged, as well as user programs. OS/VS2 Release 1 ran OS/MVT in virtual memory. Finally, OS/VS2 Release 2, which is now called MVS, provided each user with his own virtual memory.

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

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

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

Neither TSS/360 nor MULTICS achieved commercial success. What went wrong? Part of the problem was that these advanced systems were too large and too complex to be understood. Another problem was the assumption that computing power would be available from a large, remote source.

Minicomputers came along and decreased the need for large monolithic systems. They were followed by workstations and then personal computers, which put computing power closer and closer to the end users.

20.10 TOPS-20

DEC created many influential computer systems during its history. Probably the most famous operating system associated with DEC is VMS, a popular business-oriented system that is still in use today as OpenVMS, a product of Hewlett-Packard. But perhaps the most influential of DEC's operating systems was TOPS-20.

TOPS-20 started life as a research project at Bolt, Beranek, and Newman (BBN) around 1970. BBN took the business-oriented DEC PDP-10 computer running TOPS-10, added a hardware memory-paging system to implement virtual memory, and wrote a new operating system for that computer to take advantage of the new hardware features. The result was TENEX, a general-purpose timesharing system. DEC then purchased the rights to TENEX and created a new computer with a built-in hardware pager. The resulting system was the DECSYSTEM-20 and the TOPS-20 operating system.

TOPS-20 had an advanced command-line interpreter that provided help as needed to users. That, in combination with the power of the computer and its reasonable price, made the DECSYSTEM-20 the most popular time-sharing system of its time. In 1984, DEC stopped work on its line of 36-bit PDP-10 computers to concentrate on 32-bit VAX systems running VMS.

20.11 CP/M and MS/DOS

Early hobbyist computers were typically built from kits and ran a single program at a time. The systems evolved into more advanced systems as computer components improved. An early “standard” operating system for these computers of the 1970s was **CP/M**, short for Control Program/Monitor, written by Gary Kindall of Digital Research, Inc. CP/M ran primarily on the first “personal computer” CPU, the 8-bit Intel 8080. CP/M originally supported only 64 KB of memory and ran only one program at a time. Of course, it was text-based, with a command interpreter. The command interpreter resembled those in other operating systems of the time, such as the TOPS-10 from DEC.

When IBM entered the personal computer business, it decided to have Bill Gates and company write a new operating system for its 16-bit CPU of choice—the Intel 8086. This operating system, **MS-DOS**, was similar to CP/M but had a richer set of built-in commands, again mostly modeled after TOPS-10. MS-DOS became the most popular personal-computer operating system of its time, starting in 1981 and continuing development until 2000. It supported 640 KB of memory, with the ability to address “extended” and “expanded” memory to get somewhat beyond that limit. It lacked fundamental current operating-system features, however, especially protected memory.

20.12 Macintosh Operating System and Windows

With the advent of 16-bit CPUs, operating systems for personal computers could become more advanced, feature rich, and usable. The **Apple Macintosh** computer was arguably the first computer with a GUI designed for home users. It was certainly the most successful for a while, starting at its launch in 1984. It used a mouse for screen pointing and selecting and came with many utility programs that took advantage of the new user interface. Hard-disk drives were relatively expensive in 1984, so it came only with a 400-KB-capacity floppy drive by default.

The original Mac OS ran only on Apple computers and slowly was eclipsed by Microsoft Windows (starting with Version 1.0 in 1985), which was licensed to run on many different computers from a multitude of companies. As microprocessor CPUs evolved to 32-bit chips with advanced features, such as protected memory and context switching, these operating systems added features that had previously been found only on mainframes and minicomputers. Over time, personal computers became as powerful as those systems and more useful for many purposes. Minicomputers died out, replaced by general and special-purpose “servers.” Although personal computers continue to increase in capacity and performance, servers tend to stay ahead of them in amount of memory, disk space, and number and speed of available CPUs. Today, servers typically run in data centers or machine rooms, while personal computers sit on or next to desks and talk to each other and servers across a network.

The desktop rivalry between Apple and Microsoft continues today, with new versions of Windows and Mac OS trying to outdo each other in features, usability, and application functionality. Other operating systems, such as AmigaOS and OS/2, have appeared over time but have not been long-term competitors to the two leading desktop operating systems. Meanwhile, Linux in its many forms continues to gain in popularity among more technical users—and even with nontechnical users on systems like the **One Laptop per Child (OLPC)** children’s connected computer network (<http://laptop.org/>).

20.13 Mach

The Mach operating system traces its ancestry to the Accent operating system developed at Carnegie Mellon University (CMU). 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 task and thread management) were developed from scratch.

Work on Mach began in the mid 1980’s and the operating system was designed with the following three critical goals in mind:

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

Mach's development followed an evolutionary path from BSD UNIX systems. Mach code was initially developed inside the 4.2BSD kernel, with BSD kernel components replaced by Mach components as the Mach components were completed. The BSD components were updated to 4.3BSD when that became available. By 1986, the virtual memory and communication subsystems were running on the DEC VAX computer family, including multiprocessor versions of the VAX. Versions for the IBM RT/PC and for SUN 3 workstations followed shortly. Then, 1987 saw the completion of the Encore Multimax and Sequent Balance multiprocessor versions, including task and thread support, as well as the first official releases of the system, Release 0 and Release 1.

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

Mach was propelled 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. (Mach 2.5 was also the basis for the operating system on the NeXT workstation, the brainchild of Steve Jobs of Apple Computer fame.) The initial release of OSF/1 occurred a year later, and this system competed with UNIX System V, Release 4, the operating system of choice at that time among UNIX International (UI) members. OSF members included key technological companies such as IBM, DEC, and HP. OSF has since changed its direction, and only DEC UNIX is based on the Mach kernel.

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

while permitting operating-system emulation at the user level, much like IBM's virtual machine systems.

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

20.14 Other Systems

There are, of course, other operating systems, and most of them have interesting properties. The MCP operating system for the Burroughs computer family was the first to be written in a system programming language. It supported segmentation and multiple CPUs. The SCOPE operating system for the CDC 6600 was also a multi-CPU system. The coordination and synchronization of the multiple processes were surprisingly well designed.

History is littered with operating systems that suited a purpose for a time (be it a long or a short time) and then, when faded, were replaced by operating systems that had more features, supported newer hardware, were easier to use, or were better marketed. We are sure this trend will continue in the future.

Exercises

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

Bibliographical Notes

Looms and calculators are described in [Frah (2001)] and shown graphically in [Frauenfelder (2005)].

The Manchester Mark 1 is discussed by [Rojas and Hashagen (2000)], and its offspring, the Ferranti Mark 1, is described by [Ceruzzi (1998)].

[Kilburn et al. (1961)] and [Howarth et al. (1961)] examine the Atlas operating system.

The XDS-940 operating system is described by [Lichtenberger and Pirtle (1965)].

The THE operating system is covered by [Dijkstra (1968)] and by [McKeag and Wilson (1976)].

The Venus system is described by [Liskov (1972)].

[Brinch-Hansen (1970)] and [Brinch-Hansen (1973)] discuss the RC 4000 system.

The Compatible Time-Sharing System (CTSS) is presented by [Corbato et al. (1962)].

The MULTICS operating system is described by [Corbato and Vyssotsky (1965)] and [Organick (1972)].

[Mealy et al. (1966)] presented the IBM/360. [Lett and Konigsford (1968)] cover TSS/360.

CP/67 is described by [Meyer and Seawright (1970)] and [Parmelee et al. (1972)].

DEC VMS is discussed by [Kenah et al. (1988)], and TENEX is described by [Bobrow et al. (1972)].

A description of the Apple Macintosh appears in [Apple (1987)]. For more information on these operating systems and their history, see [Freiberger and Swaine (2000)].

The Mach operating system and its ancestor, the Accent operating system, are described by [Rashid and Robertson (1981)]. Mach's communication system is covered by [Rashid (1986)], [Tevanian et al. (1989)], and [Accetta et al. (1986)]. The Mach scheduler is described in detail by [Tevanian et al. (1987a)] and [Black (1990)]. An early version of the Mach shared-memory and memory-mapping system is presented by [Tevanian et al. (1987b)]. A good resource describing the Mach project can be found at <http://www.cs.cmu.edu/afs/cs/project/mach/public/www/mach.html>.

[McKeag and Wilson (1976)] discuss the MCP operating system for the Burroughs computer family as well as the SCOPE operating system for the CDC 6600.

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.

[Apple (1987)] *Apple Technical Introduction to the Macintosh Family*. Addison-Wesley (1987).

[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.

- [Bobrow et al. (1972)] D. G. Bobrow, J. D. Burchfiel, D. L. Murphy, and R. S. Tomlinson, “TENEX, a Paged Time Sharing System for the PDP-10”, *Communications of the ACM*, Volume 15, Number 3 (1972).
- [Brinch-Hansen (1970)] P. Brinch-Hansen, “The Nucleus of a Multiprogramming System”, *Communications of the ACM*, Volume 13, Number 4 (1970), pages 238–241 and 250.
- [Brinch-Hansen (1973)] P. Brinch-Hansen, *Operating System Principles*, Prentice Hall (1973).
- [Ceruzzi (1998)] P. E. Ceruzzi, *A History of Modern Computing*, MIT Press (1998).
- [Corbato and Vyssotsky (1965)] F. J. Corbato and V. A. Vyssotsky, “Introduction and Overview of the MULTICS System”, *Proceedings of the AFIPS Fall Joint Computer Conference* (1965), pages 185–196.
- [Corbato et al. (1962)] F. J. Corbato, M. Merwin-Daggett, and R. C. Daley, “An Experimental Time-Sharing System”, *Proceedings of the AFIPS Fall Joint Computer Conference* (1962), pages 335–344.
- [Dijkstra (1968)] E. W. Dijkstra, “The Structure of the THE Multiprogramming System”, *Communications of the ACM*, Volume 11, Number 5 (1968), pages 341–346.
- [Frah (2001)] G. Frah, *The Universal History of Computing*, John Wiley and Sons (2001).
- [Frauenfelder (2005)] M. Frauenfelder, *The Computer—An Illustrated History*, Carlton Books (2005).
- [Freiberger and Swaine (2000)] P. Freiberger and M. Swaine, *Fire in the Valley—The Making of the Personal Computer*, McGraw-Hill (2000).
- [Howarth et al. (1961)] D. J. Howarth, R. B. Payne, and F. H. Sumner, “The Manchester University Atlas Operating System, Part II: User’s Description”, *Computer Journal*, Volume 4, Number 3 (1961), pages 226–229.
- [Kenah et al. (1988)] L. J. Kenah, R. E. Goldenberg, and S. F. Bate, *VAX/VMS Internals and Data Structures*, Digital Press (1988).
- [Kilburn et al. (1961)] T. Kilburn, D. J. Howarth, R. B. Payne, and F. H. Sumner, “The Manchester University Atlas Operating System, Part I: Internal Organization”, *Computer Journal*, Volume 4, Number 3 (1961), pages 222–225.
- [Lett and Konigsford (1968)] A. L. Lett and W. L. Konigsford, “TSS/360: A Time-Shared Operating System”, *Proceedings of the AFIPS Fall Joint Computer Conference* (1968), pages 15–28.
- [Lichtenberger and Pirtle (1965)] W. W. Lichtenberger and M. W. Pirtle, “A Facility for Experimentation in Man-Machine Interaction”, *Proceedings of the AFIPS Fall Joint Computer Conference* (1965), pages 589–598.
- [Liskov (1972)] B. H. Liskov, “The Design of the Venus Operating System”, *Communications of the ACM*, Volume 15, Number 3 (1972), pages 144–149.
- [McKeag and Wilson (1976)] R. M. McKeag and R. Wilson, *Studies in Operating Systems*, Academic Press (1976).

- [Mealy et al. (1966)] G. H. Mealy, B. I. Witt, and W. A. Clark, “The Functional Structure of OS/360”, *IBM Systems Journal*, Volume 5, Number 1 (1966), pages 3–11.
- [Meyer and Seawright (1970)] R. A. Meyer and L. H. Seawright, “A Virtual Machine Time-Sharing System”, *IBM Systems Journal*, Volume 9, Number 3 (1970), pages 199–218.
- [Organick (1972)] E. I. Organick, *The Multics System: An Examination of Its Structure*, MIT Press (1972).
- [Parmelee et al. (1972)] R. P. Parmelee, T. I. Peterson, C. C. Tillman, and D. Hatfield, “Virtual Storage and Virtual Machine Concepts”, *IBM Systems Journal*, Volume 11, Number 2 (1972), pages 99–130.
- [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.
- [Rojas and Hashagen (2000)] R. Rojas and U. Hashagen, *The First Computers—History and Architectures*, MIT Press (2000).
- [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).

