48. Suppose a system uses ACLs to maintain its protection matrix. Write a set of management functions to manage the ACLs when (1) a new object is created; (2) an object is deleted; (3) a new domain is created; (4) a domain is deleted; (5) new access rights (a combination of *r*, *w*, *x*) are granted to a domain to access an object; (6) existing access rights of a domain to access an object are revoked; (7) new access rights are granted to all domains to access an object; (8) access rights to access an object are revoked from all domains.

10

CASE STUDY 1: LINUX

In the previous chapters, we examined many operating system principles, abstractions, algorithms, and techniques in general. Now it is time to look at some concrete systems to see how these principles are applied in the real world. We will begin with Linux, a popular variant of UNIX, which runs on a wide variety of computers. It is one of the dominant operating systems on high-end workstations and servers, but it is also used on systems ranging from cell phones to supercomputers. It also illustrates many important design principles well.

Our discussion will start with its history and evolution of UNIX and Linux. Then we will provide an overview of Linux, to give an idea of how it is used. This overview will be of special value to readers familiar only with Windows, since the latter hides virtually all the details of the system from its users. Although graphical interfaces may be easy for beginners, they provide little flexibility and no insight into how the system works.

Next we come to the heart of this chapter, an examination of processes, memory management, I/O, the file system, and security in Linux. For each topic we will first discuss the fundamental concepts, then the system calls, and finally the implementation.

Right off the bat we should address the question: Why Linux? Linux is a variant of UNIX, but there are many other versions and variants of UNIX including AIX, FreeBSD, HP-UX, SCO UNIX, System V, Solaris, and others. Fortunately, the fundamental principles and system calls are pretty much the same for all of them (by design). Furthermore, the general implementation strategies, algorithms,

and data structures are similar, but there are some differences. To make the examples concrete, it is best to choose one of them and describe it consistently. Since most readers are more likely to have encountered Linux than any of the others, we will use it as our running example, but again be aware that except for the information on implementation, much of this chapter applies to all UNIX systems. A large number of books have been written on how to use UNIX, but there are also some about advanced features and system internals (Bovet and Cesati, 2005; Maxwell, 2001; McKusick and Neville-Neil, 2004; Pate, 2003; Stevens and Rago, 2008; and Vahalia, 2007).

10.1 HISTORY OF UNIX AND LINUX

UNIX and Linux have a long and interesting history, so we will begin our study there. What started out as the pet project of one young researcher (Ken Thompson) has become a billion-dollar industry involving universities, multinational corporations, governments, and international standardization bodies. In the following pages we will tell how this story has unfolded.

10.1.1 UNICS

Back in the 1940s and 1950s, all computers were personal computers, at least in the sense that the then-normal way to use a computer was to sign up for an hour of time and take over the entire machine for that period. Of course, these machines were physically immense, but only one person (the programmer) could use them at any given time. When batch systems took over, in the 1960s, the programmer submitted a job on punched cards by bringing it to the machine room. When enough jobs had been assembled, the operator read them all in as a single batch. It usually took an hour or more after submitting a job until the output was returned. Under these circumstances, debugging was a time-consuming process, because a single misplaced comma might result in wasting several hours of the programmer's time.

To get around what almost everyone viewed as an unsatisfactory and unproductive arrangement, timesharing was invented at Dartmouth College and M.I.T. The Dartmouth system ran only BASIC and enjoyed a short-term commercial success before vanishing. The M.I.T. system, CTSS, was general purpose and was an enormous success in the scientific community. Within a short time, researchers at M.I.T. joined forces with Bell Labs and General Electric (then a computer vendor) and began designing a second-generation system, **MULTICS** (**MULTiplexed Information and Computing Service**), as we discussed in Chap. 1.

Although Bell Labs was one of the founding partners in the MULTICS project, it later pulled out, which left one of the Bell Labs researchers, Ken Thompson, looking around for something interesting to work on. He eventually decided

to write a stripped-down MULTICS by himself (in assembler this time) on a discarded PDP-7 minicomputer. Despite the tiny size of the PDP-7, Thompson's system actually worked and could support Thompson's development effort. Consequently, one of the other researchers at Bell Labs, Brian Kernighan, somewhat jokingly called it **UNICS** (**UNiplexed Information and Computing Service**). Despite puns about "EUNUCHS" being a castrated MULTICS, the name stuck, although the spelling was later changed to **UNIX**.

10.1.2 PDP-11 UNIX

Thompson's work so impressed many of his colleagues at Bell Labs that he was soon joined by Dennis Ritchie, and later by his entire department. Two major developments occurred around this time. First, UNIX was moved from the obsolete PDP-7 to the much more modern PDP-11/20 and then later to the PDP-11/45 and PDP-11/70. The latter two machines dominated the minicomputer world for much of the 1970s. The PDP-11/45 and PDP-11/70 were powerful machines with large physical memories for their era (256 KB and 2 MB, respectively). Also, they had memory protection hardware, making it possible to support multiple users at the same time. However, they were both 16-bit machines that limited individual processes to 64 KB of instruction space and 64 KB of data space, even though the machine may have had far more physical memory.

The second development concerned the language in which UNIX was written. By now it was becoming painfully obvious that having to rewrite the entire system for each new machine was no fun at all, so Thompson decided to rewrite UNIX in a high-level language of his own design, called **B**. B was a simplified form of BCPL (which itself was a simplified form of CPL, which, like PL/I, never worked). Due to weaknesses in B, primarily lack of structures, this attempt was not successful. Ritchie then designed a successor to B, (naturally) called **C**, and wrote an excellent compiler for it. Working together, Thompson and Ritchie rewrote UNIX in C. C was the right language at the right time, and has dominated system programming ever since.

In 1974, Ritchie and Thompson published a landmark paper about UNIX (Ritchie and Thompson, 1974). For the work described in this paper they were later given the prestigious ACM Turing Award (Ritchie, 1984; Thompson, 1984). The publication of this paper stimulated many universities to ask Bell Labs for a copy of UNIX. Since Bell Labs' parent company, AT&T, was a regulated monopoly at the time and was not permitted to be in the computer business, it had no objection to licensing UNIX to universities for a modest fee.

In one of those coincidences that often shape history, the PDP-11 was the computer of choice at nearly all university computer science departments, and the operating systems that came with the PDP-11 were widely regarded as dreadful by professors and students alike. UNIX quickly filled the void, not in the least because it was supplied with the complete source code, so that people could, and

did, tinker with it endlessly. Numerous scientific meetings were organized around UNIX, with distinguished speakers getting up in front of the room to tell about some obscure kernel bug they had found and fixed. An Australian professor, John Lions, wrote a commentary on the UNIX source code of the type normally reserved for the works of Chaucer or Shakespeare (reprinted as Lions, 1996). The book described Version 6, so named because it was described in the sixth edition of the UNIX Programmer's Manual. The source code was 8200 lines of C and 900 lines of assembly code. As a result of all this activity, new ideas and improvements to the system spread rapidly.

Within a couple of years, Version 6 was replaced by Version 7, the first portable version of UNIX (it ran on the PDP-11 and the Interdata 8/32), by now 18,800 lines of C and 2100 lines of assembler. A whole generation of students was brought up on Version 7, which contributed to its spread after they graduated and went to work in industry. By the mid-1980s, UNIX was in widespread use on minicomputers and engineering workstations from a variety of vendors. A number of companies even licensed the source code to make their own version of UNIX. One of these was a small startup called Microsoft, which sold Version 7 under the name XENIX for a number of years until its interest turned elsewhere.

10.1.3 Portable UNIX

Now that UNIX was written in C, moving it to a new machine, known as porting it, was much easier than in the early days. A port requires first writing a C compiler for the new machine. Then it requires writing device drivers for the new machine's I/O devices, such as monitors, printers, and disks. Although the driver code is in C, it cannot be moved to another machine, compiled, and run there because no two disks work the same way. Finally, a small amount of machine-dependent code, such as the interrupt handlers and memory management routines, must be rewritten, usually in assembly language.

The first port beyond the PDP-11 was to the Interdata 8/32 minicomputer. This exercise revealed a large number of assumptions that UNIX implicitly made about the machine it was running on, such as the unspoken supposition that integers held 16 bits, pointers also held 16 bits (implying a maximum program size of 64 KB), and that the machine had exactly three registers available for holding important variables. None of these were true on the Interdata, so considerable work was needed to clean UNIX up .

Another problem was that although Ritchie's compiler was fast and produced good object code, it produced only PDP-11 object code. Rather than write a new compiler specifically for the Interdata, Steve Johnson of Bell Labs designed and implemented the **portable C compiler**, which could be retargeted to produce code for any reasonable machine with a only a moderate amount of effort. For years, nearly all C compilers for machines other than the PDP-11 were based on Johnson's compiler, which greatly aided the spread of UNIX to new computers.

The port to the Interdata initially went slowly because all the development work had to be done on the only working UNIX machine, a PDP-11, which happened to be on the fifth floor at Bell Labs. The Interdata was on the first floor. Generating a new version meant compiling it on the fifth floor and then physically carrying a magnetic tape down to the first floor to see if it worked. After several months of tape carrying, an unknown person said: "You know, we're the phone company. Can't we run a wire between these two machines?" Thus was UNIX networking born. After the Interdata port, UNIX was ported to the VAX and other computers.

After AT&T was broken up in 1984 by the U.S. government, the company was legally free to set up a computer subsidiary, and soon did. Shortly thereafter, AT&T released its first commercial UNIX product, System III. It was not well received, so it was replaced by an improved version, System V, a year later. Whatever happened to System IV is one of the great unsolved mysteries of computer science. The original System V has since been replaced by System V, releases 2, 3, and 4, each one bigger and more complicated than its predecessor. In the process, the original idea behind UNIX, of having a simple, elegant system, has gradually diminished. Although Ritchie and Thompson's group later produced an 8th, 9th, and 10th edition of UNIX, these were never widely circulated, as AT&T put all its marketing muscle behind System V. However, some of the ideas from the 8th, 9th, and 10th editions were eventually incorporated into System V. AT&T eventually decided that it wanted to be a telephone company, not a computer company, after all, and sold its UNIX business to Novell in 1993. Novell subsequently sold it to the Santa Cruz Operation in 1995. By then it was almost irrelevant who owned it, since all the major computer companies already had licenses.

10.1.4 Berkeley UNIX

One of the many universities that acquired UNIX Version 6 early on was the University of California at Berkeley. Because the full source code was available, Berkeley was able to modify the system substantially. Aided by grants from ARPA, the U.S. Dept. of Defense's Advanced Research Projects Agency, Berkeley produced and released an improved version for the PDP-11 called **1BSD (First Berkeley Software Distribution)**. This tape was followed quickly by another one, called 2BSD, also for the PDP-11.

More important were 3BSD and especially its successor, 4BSD for the VAX. Although AT&T had a VAX version of UNIX, called **32V**, it was essentially Version 7. In contrast, 4BSD contained a large number of improvements. Foremost among these was the use of virtual memory and paging, allowing programs to be larger than physical memory by paging parts of them in and out as needed. Another change allowed file names to be longer than 14 characters. The implementation of the file system was also changed, making it considerably faster. Signal handling was made more reliable. Networking was introduced, causing the

network protocol that was used, **TCP/IP**, to become a de facto standard in the UNIX world, and later in the Internet, which is dominated by UNIX-based servers.

Berkeley also added a substantial number of utility programs to UNIX, including a new editor (*vi*), a new shell (*csh*), Pascal and Lisp compilers, and many more. All these improvements caused Sun Microsystems, DEC, and other computer vendors to base their versions of UNIX on Berkeley UNIX, rather than on AT&T's "official" version, System V. As a consequence, Berkeley UNIX became well established in the academic, research, and defense worlds. For more information about Berkeley UNIX, see McKusick et al. (1996).

10.1.5 Standard UNIX

By the late 1980s, two different, and somewhat incompatible, versions of UNIX were in widespread use: 4.3BSD and System V Release 3. In addition, virtually every vendor added its own nonstandard enhancements. This split in the UNIX world, together with the fact that there were no standards for binary program formats, greatly inhibited the commercial success of UNIX because it was impossible for software vendors to write and package UNIX programs with the expectation that they would run on any UNIX system (as was routinely done with MS-DOS). Various attempts at standardizing UNIX initially failed. AT&T, for example, issued the **SVID** (**System V Interface Definition**), which defined all the system calls, file formats, and so on. This document was an attempt to keep all the System V vendors in line, but it had no effect on the enemy (BSD) camp, which just ignored it.

The first serious attempt to reconcile the two flavors of UNIX was initiated under the auspices of the IEEE Standards Board, a highly respected and, most important, neutral body. Hundreds of people from industry, academia, and government took part in this work. The collective name for this project was **POSIX**. The first three letters refer to Portable Operating System. The *IX* was added to make the name UNIXish.

After a great deal of argument and counterargument, rebuttal and counterrebuttal, the POSIX committee produced a standard known as **1003.1**. It defines a set of library procedures that every conformant UNIX system must supply. Most of these procedures invoke a system call, but a few can be implemented outside the kernel. Typical procedures are *open*, *read*, and *fork*. The idea of POSIX is that a software vendor who writes a program that uses only the procedures defined by 1003.1 knows that this program will run on every conformant UNIX system.

While it is true that most standards bodies tend to produce a horrible compromise with a few of everyone's pet features in it, 1003.1 is remarkably good considering the large number of parties involved and their respective vested interests. Rather than take the union of all features in System V and BSD as the starting point (the norm for most standards bodies), the IEEE committee took the intersection. Very roughly, if a feature was present in both System V and BSD, it

was included in the standard; otherwise it was not. As a consequence of this algorithm, 1003.1 bears a strong resemblance to the direct ancestor of both System V and BSD, namely Version 7. The 1003.1 document is written in such a way that both operating system implementers and software writers can understand it, another novelty in the standards world, although work is already underway to remedy this.

Although the 1003.1 standard addresses only the system calls, related documents standardize threads, the utility programs, networking, and many other features of UNIX. In addition, the C language has also been standardized by ANSI and ISO.

10.1.6 MINIX

One property that all modern UNIX systems have is that they are large and complicated, in a sense, the antithesis of the original idea behind UNIX. Even if the source code were freely available, which it is not in most cases, it is out of the question that a single person could understand it all any more. This situation led the author of this book to write a new UNIX-like system that was small enough to understand, was available with all the source code, and could be used for educational purposes. That system consisted of 11,800 lines of C and 800 lines of assembly code. It was released in 1987, and was functionally almost equivalent to Version 7 UNIX, the mainstay of most computer science departments during the PDP-11 era.

MINIX was one of the first UNIX-like systems based on a microkernel design. The idea behind a microkernel is to provide minimal functionality in the kernel to make it reliable and efficient. Consequently, memory management and the file system were pushed out into user processes. The kernel handled message passing between the processes and little else. The kernel was 1600 lines of C and 800 lines of assembler. For technical reasons relating to the 8088 architecture, the I/O device drivers (2900 additional lines of C) were also in the kernel. The file system (5100 lines of C) and memory manager (2200 lines of C) ran as two separate user processes.

Microkernels have the advantage over monolithic systems that they are easy to understand and maintain due to their highly modular structure. Also, moving code from the kernel to user mode makes them highly reliable because the crash of a user-mode process does less damage than the crash of a kernel-mode component. Their main disadvantage is a slightly lower performance due to the extra switches between user mode and kernel mode. However, performance is not everything: all modern UNIX systems run X Windows in user mode and simply accept the performance hit to get the greater modularity (in contrast to Windows, where the entire **GUI** (**Graphical User Interface**) is in the kernel). Other well-known microkernel designs of this era were Mach (Accetta et al., 1986) and Chorus (Rozier et al., 1988).

Within a few months of its appearance, MINIX became a bit of a cult item, with its own USENET (now Google) newsgroup, *comp.os.minix*, and over 40,000 users. Many users contributed commands and other user programs, so MINIX became a collective undertaking done by large numbers of users over the Internet. It was a prototype of other collaborative efforts that came later. In 1997, Version 2.0 of MINIX, was released and the base system, now including networking, had grown to 62,200 lines of code.

Around 2004, the direction of MINIX development changed radically, with the focus becoming building an extremely reliable and dependable system that could automatically repair its own faults and become self healing, continuing to function correctly even in the face of repeated software bugs being triggered. As a consequence, the modularization idea present in Version 1 was greatly expanded in MINIX 3.0, with nearly all the device drivers being moved to user space, with each driver running as a separate process. The size of the entire kernel abruptly dropped to under 4000 lines of code, something a single programmer could easily understand. Internal mechanisms were changed to enhance fault tolerance in numerous ways.

In addition, over 500 popular UNIX programs were ported to MINIX 3.0, including the **X Window System** (sometimes just called **X**), various compilers (including *gcc*), text-processing software, networking software, Web browsers, and much more. Unlike previous versions, which were primarily educational in nature, starting with MINIX 3.0, the system was quite usable, with the focus moving toward high dependability. The ultimate goal is: No more reset buttons.

A third edition of the book appeared, describing the new system and giving its source code in an appendix and describing it in detail (Tanenbaum and Woodhull, 2006). The system continues to evolve and has an active user community. For more details and to get the current version for free, you can visit *www.minix3.org*.

10.1.7 Linux

During the early years of MINIX development and discussion on the Internet, many people requested (or in many cases, demanded) more and better features, to which the author often said "No" (to keep the system small enough for students to understand completely in a one-semester university course). This continuous "No" irked many users. At this time, FreeBSD was not available, so that was not an option. After a number of years went by like this, a Finnish student, Linus Torvalds, decided to write another UNIX clone, named **Linux,** which would be a full-blown production system with many features MINIX was initially lacking. The first version of Linux, 0.01, was released in 1991. It was cross-developed on a MINIX machine and borrowed numerous ideas from MINIX, ranging from the structure of the source tree to the layout of the file system. However, it was a monolithic rather than a microkernel design, with the entire operating system in the kernel. The code totaled 9300 lines of C and 950 lines of assembler, roughly

similar to MINIX version in size and also comparable in functionality. De facto, it was a rewrite of MINIX, the only system Torvalds had source code for.

Linux rapidly grew in size and evolved into a full, production UNIX clone as virtual memory, a more sophisticated file system, and many other features were added. Although it originally ran only on the 386 (and even had embedded 386 assembly code in the middle of C procedures), it was quickly ported to other platforms and now runs on a wide variety of machines, just as UNIX does. One difference with UNIX does stand out, however: Linux makes use of many special features of the *gcc* compiler and would need a lot of work before it would compile with an ANSI standard C compiler.

The next major release of Linux was version 1.0, issued in 1994. It was about 165,000 lines of code and included a new file system, memory-mapped files, and BSD-compatible networking with sockets and TCP/IP. It also included many new device drivers. Several minor revisions followed in the next two years.

By this time, Linux was sufficiently compatible with UNIX that a vast amount of UNIX software was ported to Linux, making it far more useful than it would have otherwise been. In addition, a large number of people were attracted to Linux and began working on the code and extending it in many ways under Torvalds' general supervision.

The next major release, 2.0, was made in 1996. It consisted of about 470,000 lines of C and 8000 lines of assembly code. It included support for 64-bit architectures, symmetric multiprogramming, new networking protocols, and numerous other features. A large fraction of the total code mass was taken up by an extensive collection of device drivers. Additional releases followed frequently.

The version numbers of the Linux kernel consist of four numbers, *A.B.C.D*, such as 2.6.9.11. The first number denotes the kernel version. The second number denotes the major revision. Prior to the 2.6 kernel, even revision numbers corresponded to stable kernel releases, whereas odd ones corresponded to unstable revisions, under development. With the 2.6 kernel that is no longer the case. The third number corresponds to minor revisions, such as support for new drivers. The fourth number corresponds to minor bug fixes or security patches.

A large array of standard UNIX software has been ported to Linux, including the X Window System and a great deal of networking software. Two different GUIs (GNOME and KDE) have also been written for Linux. In short, it has grown to a full-blown UNIX clone with all the bells and whistles a UNIX lover might want.

One unusual feature of Linux is its business model: it is free software. It can be downloaded from various sites on the Internet, for example: *www.kernel.org*. Linux comes with a license devised by Richard Stallman, founder of the Free Software Foundation. Despite the fact that Linux is free, this license, the **GPL** (**GNU Public License**), is longer than Microsoft's Windows license and specifies what you can and cannot do with the code. Users may use, copy, modify, and redistribute the source and binary code freely. The main restriction is that all

works derived from the Linux kernel may not be sold or redistributed in binary form only; the source code must either be shipped with the product or be made available on request.

Although Torvalds still controls the kernel fairly closely, a large amount of user-level software has been written by numerous other programmers, many of them originally migrated over from the MINIX, BSD, and GNU online communities. However, as Linux evolves, a steadily smaller fraction of the Linux community want to hack source code (witness the hundreds of books telling how to install and use Linux and only a handful discussing the code or how it works). Also, many Linux users now forgo the free distribution on the Internet to buy one of the many CD-ROM distributions available from numerous competing commercial companies. A popular Website listing the current top-100 Linux distributions is at *www.distrowatch.org*. As more and more software companies start selling their own versions of Linux and more and more hardware companies offer to preinstall it on the computers they ship, the line between commercial software and free software is beginning to blur substantially.

As a footnote to the Linux story, it is interesting to note that just as the Linux bandwagon was gaining steam, it got a big boost from an unexpected source—AT&T. In 1992, Berkeley, by now running out of funding, decided to terminate BSD development with one final release, 4.4BSD, (which later formed the basis of FreeBSD). Since this version contained essentially no AT&T code, Berkeley issued the software under an open source license (not GPL) that let everybody do whatever they wanted with it except one thing—sue the University of California. The AT&T subsidiary controlling UNIX promptly reacted by—you guessed it—suing the University of California. It also sued a company, BSDI, set up by the BSD developers to package the system and sell support, much as Red Hat and other companies now do for Linux. Since virtually no AT&T code was involved, the lawsuit was based on copyright and trademark infringement, including items such as BSDI's 1-800-ITS-UNIX telephone number. Although the case was eventually settled out of court, it kept FreeBSD off the market long enough for Linux to get well established. Had the lawsuit not happened, starting around 1993 there would have been serious competition between two free, open source UNIX systems: the reigning champion, BSD, a mature and stable system with a large academic following dating back to 1977, versus the vigorous young challenger, Linux, just two years old but with a growing following among individual users. Who knows how this battle of the free UNICES would have turned out?

10.2 OVERVIEW OF LINUX

In this section we will provide a general introduction to Linux and how it is used, for the benefit of readers not already familiar with it. Nearly all of this material applies to just about all UNIX variants with only small deviations. Although Linux has several graphical interfaces, the focus here is on how Linux appears to a programmer working in a shell window on X. Subsequent sections will focus on system calls and how it works inside.

10.2.1 Linux Goals

UNIX was always an interactive system designed to handle multiple processes and multiple users at the same time. It was designed by programmers, for programmers, to use in an environment in which the majority of the users are relatively sophisticated and are engaged in (often quite complex) software development projects. In many cases, a large number of programmers are actively cooperating to produce a single system, so UNIX has extensive facilities to allow people to work together and share information in controlled ways. The model of a group of experienced programmers working together closely to produce advanced software is obviously very different from the personal computer model of a single beginner working alone with a word processor, and this difference is reflected throughout UNIX from start to finish. It is only natural that Linux inherited many of these goals, even though the first version was for a personal computer.

What is it that good programmers want in a system? To start with, most like their systems to be simple, elegant, and consistent. For example, at the lowest level, a file should just be a collection of bytes. Having different classes of files for sequential access, random access, keyed access, remote access, and so on, (as mainframes do) just gets in the way. Similarly, if the command

```
ls A*
```

means list all the files beginning with "A" then the command

```
rm A*
```

should mean remove all the files beginning with "A" and not remove the one file whose name consists of an "A" and an asterisk. This characteristic is sometimes called the *principle of least surprise*.

Another thing that experienced programmers generally want is power and flexibility. This means that a system should have a small number of basic elements that can be combined in an infinite variety of ways to suit the application. One of the basic guidelines behind Linux is that every program should do just one thing and do it well. Thus compilers do not produce listings, because other programs can do that better.

Finally, most programmers have a strong dislike for useless redundancy. Why type *copy* when *cp* is enough? To extract all the lines containing the string "ard" from the file *f*, the Linux programmer types

```
grep ard f
```

The opposite approach is to have the programmer first select the *grep* program (with no arguments), and then have *grep* announce itself by saying: "Hi, I'm *grep*,

I look for patterns in files. Please enter your pattern." After getting the pattern, *grep* prompts for a file name. Then it asks if there are any more file names. Finally, it summarizes what it is going to do and asks if that is correct. While this kind of user interface may be suitable for rank novices, it drives skilled programmers up the wall. What they want is a servant, not a nanny.

10.2.2 Interfaces to Linux

A Linux system can be regarded as a kind of pyramid, as illustrated in Fig. 10-1. At the bottom is the hardware, consisting of the CPU, memory, disks, a monitor and keyboard, and other devices. Running on the bare hardware is the operating system. Its function is to control the hardware and provide a system call interface to all the programs. These system calls allow user programs to create and manage processes, files, and other resources.

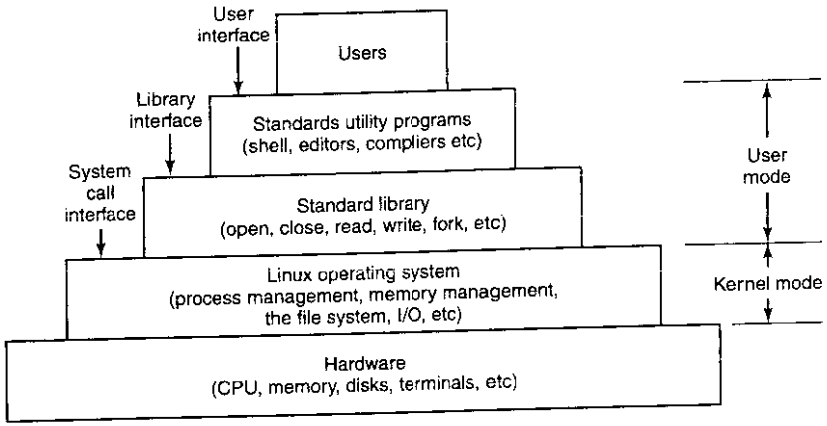

Figure 10-1. The layers in a Linux system.

Programs make system calls by putting the arguments in registers (or sometimes, on the stack), and issuing trap instructions to switch from user mode to kernel mode. Since there is no way to write a trap instruction in C, a library is provided, with one procedure per system call. These procedures are written in assembly language, but can be called from C. Each one first puts its arguments in the proper place, then executes the trap instruction. Thus to execute the read system call, a C program can call the *read* library procedure. As an aside, it is the library interface, and not the system call interface, that is specified by POSIX. In other words, POSIX tells which library procedures a conformant system must supply, what their parameters are, what they must do, and what results they must return. It does not even mention the actual system calls.

In addition to the operating system and system call library, all versions of Linux supply a large number of standard programs, some of which are specified by the POSIX 1003.2 standard, and some of which differ between Linux versions. These include the command processor (shell), compilers, editors, text processing programs, and file manipulation utilities. It is these programs that a user at the keyboard invokes. Thus we can speak of three different interfaces to Linux: the true system call interface, the library interface, and the interface formed by the set of standard utility programs.

Most personal computer distributions of Linux have replaced this keyboard-oriented user interface with a mouse-oriented graphical user interface, without changing the operating system itself at all. It is precisely this flexibility that makes Linux so popular and has allowed it to survive numerous changes in the underlying technology so well.

The GUI for Linux is similar to the first GUIs developed for UNIX systems in the 1970s, and popularized by Macintosh and later Windows for PC platforms. The GUI creates a desktop environment, a familiar metaphor with windows, icons, folders, toolbars, and drag-and-drop capabilities. A full desktop environment contains a window manager, which controls the placement and appearance of windows, as well as various applications, and provides a consistent graphical interface. Popular desktop environments for Linux include GNOME (GNU Network Object Model Environment) and KDE (K Desktop Environment).

GUIs on Linux are supported by the X Windowing System, or commonly X11 or just X, which defines communication and display protocols for manipulating windows on bitmap displays for UNIX and UNIX-like systems. The X server is the main component which controls devices such as keyboards, mouse, screen and is responsible for redirecting input to or accepting output from client programs. The actual GUI environment is typically built on top of a low-level library, *xlib*, which contains the functionality to interact with the X server. The graphical interface extends the basic functionality of X11 by enriching the window view, providing buttons, menus, icons, and other options. The X server can be started manually, from a command line, but is typically started during the boot process by a display manager, which displays the graphical login screen for the user.

When working on Linux systems through a graphical interface, users may use mouse clicks to run applications or open files, drag and drop to copy files from one location to another, and so on. In addition, users may invoke a terminal emulator program, or *xterm*, which provides them with the basic command-line interface to the operating system. Its description is given in the following section.

10.2.3 The Shell

Although Linux systems have a graphical user interface, most programmers and sophisticated users still prefer a command-line interface, called the **shell**. Often they start one or more shell windows from the graphical user interface and

just work in them. The shell command-line interface is much faster to use, more powerful, easily extensible, and does not give the user RSI from having to use a mouse all the time. Below we will briefly describe the bash shell (*bash*). It is heavily based on the original UNIX shell, *Bourne shell*, and in fact its name is an acronym for *Bourne Again SHell*. Many other shells are also in use (*ksh*, *csh*, etc.), but, *bash* is the default shell in most Linux systems.

When the shell starts up, it initializes itself, then types a **prompt** character, often a percent or dollar sign, on the screen and waits for the user to type a command line.

When the user types a command line, the shell extracts the first word from it, assumes it is the name of a program to be run, searches for this program, and if it finds it, runs the program. The shell then suspends itself until the program terminates, at which time it tries to read the next command. What is important here is simply the observation that the shell is an ordinary user program. All it needs is the ability to read from the keyboard and write to the monitor and the power to execute other programs.

Commands may take arguments, which are passed to the called program as character strings. For example, the command line

```
cp src dest
```

invokes the *cp* program with two arguments, *src* and *dest*. This program interprets the first one to be the name of an existing file. It makes a copy of this file and calls the copy *dest*.

Not all arguments are file names. In

```
head –20 file
```

the first argument, *–20*, tells *head* to print the first 20 lines of *file*, instead of the default number of lines, 10. Arguments that control the operation of a command or specify an optional value are called **flags**, and by convention are indicated with a dash. The dash is required to avoid ambiguity, because the command

```
head 20 file
```

is perfectly legal, and tells *head* to first print the initial 10 lines of a file called *20*, and then print the initial 10 lines of a second file called *file*. Most Linux commands accept multiple flags and arguments.

To make it easy to specify multiple file names, the shell accepts **magic characters**, sometimes called **wild cards**. An asterisk, for example, matches all possible strings, so

```
ls *.c
```

tells *ls* to list all the files whose name ends in *.c*. If files named *x.c*, *y.c*, and *z.c* all exist, the above command is equivalent to typing

```
ls x.c y.c z.c
```


Another wild card is the question mark, which matches any one character. A list of characters inside square brackets selects any of them, so

```
ls [ape]*
```

lists all files beginning with "a", "p", or "e".

A program like the shell does not have to open the terminal (keyboard and monitor) in order to read from it or write to it. Instead, when it (or any other program) starts up, it automatically has access to a file called **standard input** (for reading), a file called **standard output** (for writing normal output), and a file called **standard error** (for writing error messages). Normally, all three default to the terminal, so that reads from standard input come from the keyboard and writes to standard output or standard error go to the screen. Many Linux programs read from standard input and write to standard output as the default. For example,

```
sort
```

invokes the *sort* program, which reads lines from the terminal (until the user types a CTRL-D, to indicate end of file), sorts them alphabetically, and writes the result to the screen.

It is also possible to redirect standard input and standard output, as that is often useful. The syntax for redirecting standard input uses a less than sign (<) followed by the input file name. Similarly, standard output is redirected using a greater than sign (>). It is permitted to redirect both in the same command. For example, the command

```
sort <in >out
```

causes *sort* to take its input from the file *in* and write its output to the file *out*. Since standard error has not been redirected, any error messages go to the screen. A program that reads its input from standard input, does some processing on it, and writes its output to standard output is called a **filter**.

Consider the following command line consisting of three separate commands:

```
sort <in >temp; head –30 <temp; rm temp
```

It first runs *sort*, taking the input from *in* and writing the output to *temp*. When that has been completed, the shell runs *head*, telling it to print the first 30 lines of *temp* and print them on standard output, which defaults to the terminal. Finally, the temporary file is removed.

It frequently occurs that the first program in a command line produces output that is used as the input on the next program. In the above example, we used the file *temp* to hold this output. However, Linux provides a simpler construction to do the same thing. In

```
sort <in | head –30
```

the vertical bar, called the **pipe symbol**, says to take the output from *sort* and use

it as the input to *head*, eliminating the need for creating, using, and removing the temporary file. A collection of commands connected by pipe symbols, called a **pipeline**, may contain arbitrarily many commands. A four-component pipeline is shown by the following example:

```
grep ter *.t | sort | head –20 | tail –5 >foo
```

Here all the lines containing the string "ter" in all the files ending in *.t* are written to standard output, where they are sorted. The first 20 of these are selected out by *head*, which passes then to *tail*, which writes the last five (i.e., lines 16 to 20 in the sorted list) to *foo*. This is an example of how Linux provides basic building blocks (numerous filters), each of which does one job, along with a mechanism for them to be put together in almost limitless ways.

Linux is a general-purpose multiprogramming system. A single user can run several programs at once, each as a separate process. The shell syntax for running a process in the background is to follow its command with an ampersand. Thus

```
wc –l <a >b &
```

runs the word-count program, *wc*, to count the number of lines (*–l* flag) in its input, *a*, writing the result to *b*, but does it in the background. As soon as the command has been typed, the shell types the prompt and is ready to accept and handle the next command. Pipelines can also be put in the background, for example, by

```
sort <x | head &
```

Multiple pipelines can run in the background simultaneously.

It is possible to put a list of shell commands in a file and then start a shell with this file as standard input. The (second) shell just processes them in order, the same as it would with commands typed on the keyboard. Files containing shell commands are called **shell scripts**. Shell scripts may assign values to shell variables and then read them later. They may also have parameters, and use if, for, while, and case constructs. Thus a shell script is really a program written in shell language. The Berkeley C shell is an alternative shell that has been designed to make shell scripts (and the command language in general) look like C programs in many respects. Since the shell is just another user program, other people have written and distributed a variety of other shells.

10.2.4 Linux Utility Programs

The command-line (shell) user interface to Linux consists of a large number of standard utility programs. Roughly speaking, these programs can be divided into six categories, as follows:

1. File and directory manipulation commands.
2. Filters.
3. Program development tools, such as editors and compilers.
4. Text processing.
5. System administration.
6. Miscellaneous.

The POSIX 1003.2 standard specifies the syntax and semantics of just under 100 of these, primarily in the first three categories. The idea of standardizing them is to make it possible for anyone to write shell scripts that use these programs and work on all Linux systems.

In addition to these standard utilities, there are many application programs as well, of course, such as Web browsers, image viewers, and so on.

Let us consider some examples of these programs, starting with file and directory manipulation.

```
cp a b
```

copies file *a* to *b*, leaving the original file intact. In contrast,

```
mv a b
```

copies *a* to *b* but removes the original. In effect, it moves the file rather than really making a copy in the usual sense. Several files can be concatenated using *cat*, which reads each of its input files and copies them all to standard output, one after another. Files can be removed by the *rm* command. The *chmod* command allows the owner to change the rights bits to modify access permissions. Directories can be created with *mkdir* and removed with *rmdir*. To see a list of the files in a directory, *ls* can be used. It has a vast number of flags to control how much detail about each file is shown (e.g., size, owner, group, creation date), to determine the sort order (e.g., alphabetical, by time of last modification, reversed), to specify the layout on the screen, and much more.

We have already seen several filters: *grep* extracts lines containing a given pattern from standard input or one or more input files; *sort* sorts its input and writes it on standard output; *head* extracts the initial lines of its input; *tail* extracts the final lines of its input. Other filters defined by 1003.2 are *cut* and *paste*, which allow columns of text to be cut and pasted into files; *od*, which converts its (usually binary) input to ASCII text, in octal, decimal, or hexadecimal; *tr*, which does character translation (e.g., lower case to upper case), and *pr* which formats output for the printer, including options to include running heads, page numbers, and so on.

Compilers and programming tools include *gcc*, which calls the C compiler, and *ar*, which collects library procedures into archive files.

Another important tool is *make*, which is used to maintain large programs whose source code consists of multiple files. Typically, some of these are **header files**, which contain type, variable, macro, and other declarations. Source files often include these using a special *include* directive. This way, two or more source files can share the same declarations. However, if a header file is modified, it is necessary to find all the source files that depend on it and recompile them. The function of *make* is to keep track of which file depends on which header, and similar things, and arrange for all the necessary compilations to occur automatically. Nearly all Linux programs, except the smallest ones, are set up to be compiled with *make*.

A selection of the POSIX utility programs is listed in Fig. 10-2, along with a short description of each. All Linux systems have them and many more.

Program	Typical use
cat	Concatenate multiple files to standard output
chmod	Change file protection mode
cp	Copy one or more files
cut	Cut columns of text from a file
grep	Search a file for some pattern
head	Extract the first lines of a file
ls	List directory
make	Compile files to build a binary
mkdir	Make a directory
od	Octal dump a file
paste	Paste columns of text into a file
pr	Format a file for printing
ps	List running processes
rm	Remove one or more files
rmdir	Remove a directory
sort	Sort a file of lines alphabetically
tail	Extract the last lines of a file
tr	Translate between character sets

Figure 10-2. A few of the common Linux utility programs required by POSIX.

10.2.5 Kernel Structure

In Fig. 10-1 we saw the overall structure of a Linux system. Now let us zoom in and look more closely at the kernel as a whole before examining the various parts, such as process scheduling and the file system.

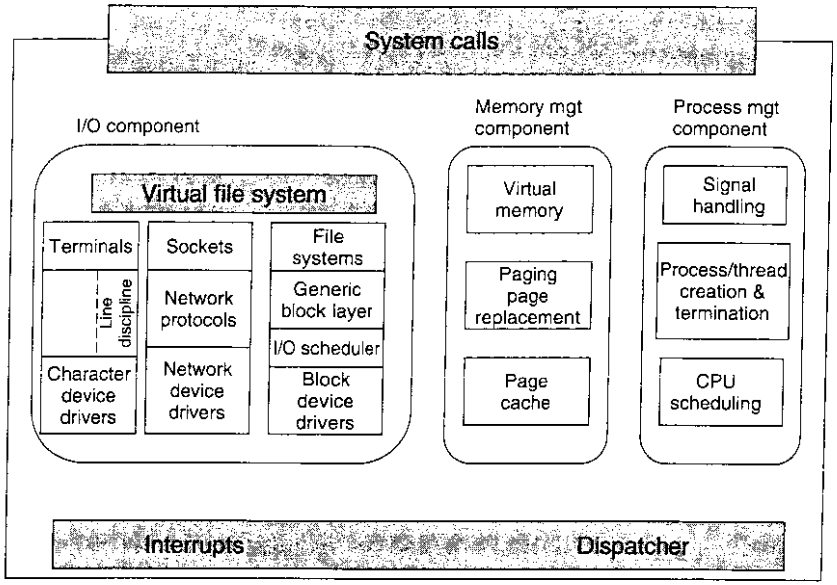

Figure 10-3. Structure of the Linux kernel

The kernel sits directly on the hardware and enables interactions with I/O devices and the memory management unit and controls CPU access to them. At the lowest level, as shown in Fig. 10-3 it contains interrupt handlers, which are the primary way for interacting with devices, and the low-level dispatching mechanism. This dispatching occurs when an interrupt happens. The low-level code here stops the running process, saves its state in the kernel process structures, and starts the appropriate driver. Process dispatching also happens when the kernel completes some operations and it is time to start up a user process again. The dispatching code is in assembler and is quite distinct from scheduling.

Next, we divide the various kernel subsystems into three main components. The I/O component in Fig. 10-3 contains all kernel pieces responsible for interacting with devices and performing network and storage I/O operations. At the highest level, the I/O operations are all integrated under a Virtual File System layer. That is, at the top level, performing a read operation to a file, whether it is in memory or on disk, is the same as performing a read operation to retrieve a character from a terminal input. At the lowest level, all I/O operations pass through some device driver. All Linux drivers are classified as either character device drivers or block device drivers, with the main difference that seeks and random accesses are allowed on block devices and not on character devices. Technically, network devices are really character devices, but they are handled

somewhat differently, so that it is probably clearer to separate them, as has been done in the figure.

Above the device driver level, the kernel code is different for each device type. Character devices may be used in two different ways. Some programs, such as visual editors like *vi* and *emacs*, want every key stroke as it is hit. Raw terminal (tty) I/O makes this possible. Other software, such as the shell, is line oriented, and allows users to edit the whole line before hitting ENTER to send it to the program. In this case the character stream from the terminal device is passed through a so called line discipline, and appropriate formatting is applied.

Networking software is often modular, with different devices and protocols supported. The layer above the network drivers handles a kind of routing function, making sure that the right packet goes to the right device or protocol handler. Most Linux systems contain the full functionality of a hardware router within the kernel, although the performance is less than that of a hardware router. Above the router code is the actual protocol stack, always including IP and TCP, but also many additional protocols. Overlaying all the network is the socket interface, which allows programs to create sockets for particular networks and protocols, getting back a file descriptor for each socket to use later.

On top of the disk drivers is the I/O scheduler, which is responsible for ordering and issuing disk operation requests in a way that tries to conserve wasteful disk head movement or to meet some other system policy.

At the very top of the block device column are the file systems. Linux may have, and it does in fact, multiple file systems coexisting concurrently. In order to hide the gruesome architectural differences of various hardware devices from the file system implementation, a generic block device layer provides an abstraction used by all file systems.

To the right in Fig. 10-3 are the other two key components of the Linux kernel. These are responsible for the memory and process management tasks. Memory management tasks include maintaining the virtual to physical memory mappings, maintaining a cache of recently accessed pages and implementing a good page replacement policy, and on-demand bringing in new pages of needed code and data into memory.

The key responsibility of the process management component is the creation and termination of processes. It also includes the process scheduler, which chooses which process or, rather, thread to run next. As we shall see in the next section, the Linux kernel treats both processes and threads simply as executable entities, and will schedule them based on a global scheduling policy. Finally, code for signal handling also belongs to this component.

While the three components are represented separately in the figure, they are highly interdependent. File systems typically access files through the block devices. However, in order to hide the large latencies of disk accesses, files are copied into the page cache in main memory. Some files may even be dynamically created and may only have an in-memory representation, such as files providing

some runtime resource usage information. In addition, the virtual memory system may rely on a disk partition or in-file swap area to back up parts of the main memory when it needs to free up certain pages, and therefore relies on the I/O component. Numerous other interdependencies exist.

In addition to the static in-kernel components, Linux supports dynamically loadable modules. These modules can be used to add or replace the default device drivers, file system, networking, or other kernel codes. The modules are not shown in Fig. 10-3.

Finally, at the very top is the system call interface into the kernel. All system calls come here, causing a trap which switches the execution from user mode into protected kernel mode and passes control to one of the kernel components described above.

10.3 PROCESSES IN LINUX

In the previous sections, we started out by looking at Linux as viewed from the keyboard, that is, what the user sees in an *xterm* window. We gave examples of shell commands and utility programs that are frequently used. We ended with a brief overview of the system structure. Now it is time to dig deeply into the kernel and look more closely at the basic concepts Linux supports, namely, processes, memory, the file system, and input/output. These notions are important because the system calls—the interface to the operating system itself—manipulate them. For example, system calls exist to create processes and threads, allocate memory, open files, and do I/O.

Unfortunately, with so many versions of Linux in existence, there are some differences between them. In this chapter, we will emphasize the features common to all of them rather than focus on any one specific version. Thus in certain sections (especially implementation sections), the discussion may not apply equally to every version.

10.3.1 Fundamental Concepts

The main active entities in a Linux system are the processes. Linux processes are very similar to the classical sequential processes that we studied in Chap 2. Each process runs a single program and initially has a single thread of control. In other words, it has one program counter, which keeps track of the next instruction to be executed. Linux allows a process to create additional threads once it starts executing.

Linux is a multiprogramming system, so multiple, independent processes may be running at the same time. Furthermore, each user may have several active processes at once, so on a large system, there may be hundreds or even thousands of processes running. In fact, on most single-user workstations, even when the

user is absent, dozens of background processes, called **daemons,** are running. These are started by a shell script when the system is booted. ("Daemon" is a variant spelling of "demon," which is a self-employed evil spirit.)

A typical daemon is the *cron daemon*. It wakes up once a minute to check if there is any work for it to do. If so, it does the work. Then it goes back to sleep until it is time for the next check.

This daemon is needed because it is possible in Linux to schedule activities minutes, hours, days, or even months in the future. For example, suppose a user has a dentist appointment at 3 o'clock next Tuesday. He can make an entry in the cron daemon's database telling the daemon to beep at him at, say, 2:30. When the appointed day and time arrives, the cron daemon sees that it has work to do, and starts up the beeping program as a new process.

The cron daemon is also used to start up periodic activities, such as making daily disk backups at 4 A.M., or reminding forgetful users every year on October 31 to stock up on trick-or-treat goodies for Halloween. Other daemons handle incoming and outgoing electronic mail, manage the line printer queue, check if there are enough free pages in memory, and so forth. Daemons are straightforward to implement in Linux because each one is a separate process, independent of all other processes.

Processes are created in Linux in an especially simple manner. The fork system call creates an exact copy of the original process. The forking process is called the **parent process**. The new process is called the **child process**. The parent and child each have their own, private memory images. If the parent subsequently changes any of its variables, the changes are not visible to the child, and vice versa.

Open files are shared between parent and child. That is, if a certain file was open in the parent before the fork, it will continue to be open in both the parent and the child afterward. Changes made to the file by either one will be visible to the other. This behavior is only reasonable, because these changes are also visible to any unrelated process that opens the file.

The fact that the memory images, variables, registers, and everything else are identical in the parent and child leads to a small difficulty: How do the processes know which one should run the parent code and which one should run the child code? The secret is that the fork system call returns a 0 to the child and a nonzero value, the child's **PID (Process Identifier)**, to the parent. Both processes normally check the return value and act accordingly, as shown in Fig. 10-4.

Processes are named by their PIDs. When a process is created, the parent is given the child's PID, as mentioned above. If the child wants to know its own PID, there is a system call, getpid, that provides it. PIDs are used in a variety of ways. For example, when a child terminates, the parent is given the PID of the child that just finished. This can be important because a parent may have many children. Since children may also have children, an original process can build up an entire tree of children, grandchildren, and further descendants.


```
pid = fork( );                  /* if the fork succeeds, pid > 0 in the parent */
if (pid < 0) {
     handle_error( );           /* fork failed (e.g., memory or some table is full) */
} else if (pid > 0) {
                                /* parent code goes here. /*/
} else {
                                /* child code goes here. /*/
}
```

Figure 10-4. Process creation in Linux.

Processes in Linux can communicate with each other using a form of message passing. It is possible to create a channel between two processes into which one process can write a stream of bytes for the other to read. These channels are called **pipes**. Synchronization is possible because when a process tries to read from an empty pipe it is blocked until data are available.

Shell pipelines are implemented with pipes. When the shell sees a line like

```
sort <f | head
```

it creates two processes, *sort* and *head*, and sets up a pipe between them in such a way that *sort*'s standard output is connected to *head*'s standard input. In this way, all the data that *sort* writes go directly to *head*, instead of going to a file. If the pipe fills, the system stops running *sort* until *head* has removed some data from it.

Processes can also communicate in another way: software interrupts. A process can send what is called a **signal** to another process. Processes can tell the system what they want to happen when a signal arrives. The choices are to ignore it, to catch it, or to let the signal kill the process (the default for most signals). If a process elects to catch signals sent to it, it must specify a signal-handling procedure. When a signal arrives, control will abruptly switch to the handler. When the handler is finished and returns, control goes back to where it came from, analogous to hardware I/O interrupts. A process can only send signals to members of its **process group**, which consists of its parent (and further ancestors), siblings, and children (and further descendants). A process may also send a signal to all members of its process group with a single system call.

Signals are also used for other purposes. For example, if a process is doing floating-point arithmetic, and inadvertently divides by 0, it gets a SIGFPE (floating-point exception) signal. The signals that are required by POSIX are listed in Fig. 10-5. Many Linux systems have additional signals as well, but programs using them may not be portable to other versions of Linux and UNIX in general.

10.3.2 Process Management System Calls in Linux

Let us now look at the Linux system calls dealing with process management. The main ones are listed in Fig. 10-6. Fork is a good place to start the discussion. The Fork system call, supported also by other traditional UNIX systems, is the

Signal	Cause
SIGABRT	Sent to abort a process and force a core dump
SIGALRM	The alarm clock has gone off
SIGFPE	A floating-point error has occurred (e.g., division by 0)
SIGHUP	The phone line the process was using has been hung up
SIGILL	The user has hit the DEL key to interrupt the process
SIGQUIT	The user has hit the key requesting a core dump
SIGKILL	Sent to kill a process (cannot be caught or ignored)
SIGPIPE	The process has written to a pipe which has no readers
SIGSEGV	The process has referenced an invalid memory address
SIGTERM	Used to request that a process terminate gracefully
SIGUSR1	Available for application-defined purposes
SIGUSR2	Available for application-defined purposes

Figure 10-5. The signals required by POSIX.

main way to create a new process in Linux systems (We will discuss another alternative in the following subsection.) It creates an exact duplicate of the original process, including all the file descriptors, registers, and everything else. After the fork, the original process and the copy (the parent and child) go their separate ways. All the variables have identical values at the time of the fork, but since the entire parent address space is copied to create the child, subsequent changes in one of them do not affect the other. The fork call returns a value, which is zero in the child, and equal to the child's PID in the parent. Using the returned PID, the two processes can see which is the parent and which is the child.

In most cases, after a fork, the child will need to execute different code from the parent. Consider the case of the shell. It reads a command from the terminal, forks off a child process, waits for the child to execute the command, and then reads the next command when the child terminates. To wait for the child to finish, the parent executes a waitpid system call, which just waits until the child terminates (any child if more than one exists). Waitpid has three parameters. The first one allows the caller to wait for a specific child. If it is −1, any old child (i.e., the first child to terminate) will do. The second parameter is the address of a variable that will be set to the child's exit status (normal or abnormal termination and exit value). The third one determines whether the caller blocks or returns if no child is already terminated.

In the case of the shell, the child process must execute the command typed by the user. It does this by using the exec system call, which causes its entire core image to be replaced by the file named in its first parameter. A highly simplified shell illustrating the use of fork, waitpid, and exec is shown in Fig. 10-7.

System call	Description
pid = fork( )	Create a child process identical to the parent
pid = waitpid(pid, &statloc, opts)	Wait for a child to terminate
s = execve(name, argv, envp)	Replace a process' core image
exit(status)	Terminate process execution and return status
s = sigaction(sig, &act, &oldact)	Define action to take on signals
s = sigreturn(&context)	Return from a signal
s = sigprocmask(how, &set, &old)	Examine or change the signal mask
s = sigpending(set)	Get the set of blocked signals
s = sigsuspend(sigmask)	Replace the signal mask and suspend the process
s = kill(pid, sig)	Send a signal to a process
residual = alarm(seconds)	Set the alarm clock
s = pause( )	Suspend the caller until the next signal

Figure 10-6. Some system calls relating to processes. The return code *s* is −1 if an error has occurred, *pid* is a process ID, and *residual* is the remaining time in the previous alarm. The parameters are what the names suggest.

In the most general case, exec has three parameters: the name of the file to be executed, a pointer to the argument array, and a pointer to the environment array. These will be described shortly. Various library procedures, such as *execl*, *execv*, *execle*, and *execve*, are provided to allow the parameters to be omitted or specified in various ways. All of these procedures invoke the same underlying system call. Although the system call is exec, there is no library procedure with this name; one of the others must be used.

Let us consider the case of a command typed to the shell, such as

```
cp file1 file2
```

used to copy *file1* to *file2*. After the shell has forked, the child locates and executes the file *cp* and passes it information about the files to be copied.

The main program of *cp* (and many other programs) contains the function declaration

```
main(argc, argv, envp)
```

where *argc* is a count of the number of items on the command line, including the program name. For the example above, *argc* is 3.

The second parameter, *argv*, is a pointer to an array. Element *i* of that array is a pointer to the *i*-th string on the command line. In our example, *argv*[0] would point to the string "cp". Similarly, *argv*[1] would point to the five-character string "file1" and *argv*[2] would point to the five-character string "file2".

The third parameter of *main*, *envp*, is a pointer to the environment, an array of strings containing assignments of the form *name* = *value* used to pass information

```
while (TRUE) {                                  /* repeat forever */
     type_prompt( );                            /* display prompt on the screen */
     read_command(command, params);             /* read input line from keyboard */

     pid = fork( );                             /* fork off a child process */
     if (pid < 0) {                             /* error condition */
          printf("Unable to fork0);             /* repeat the loop */
          continue;
     }

     if (pid != 0) {
          waitpid (-1, &status, 0);             /* parent waits for child */
     } else {
          execve(command, params, 0);           /* child does the work */
     }
}
```

Figure 10-7. A highly simplified shell.

such as the terminal type and home directory name to a program. In Fig. 10-7, no environment is passed to the child, so that the third parameter of *execve* is a zero in this case.

If exec seems complicated, do not despair; it is the most complex system call. All the rest are much simpler. As an example of a simple one, consider exit, which processes should use when they are finished executing. It has one parameter, the exit status (0 to 255), which is returned to the parent in the variable *status* of the waitpid system call. The low-order byte of *status* contains the termination status, with 0 being normal termination and the other values being various error conditions. The high-order byte contains the child's exit status (0 to 255), as specified in the child's call to exit. For example, if a parent process executes the statement

```
n = waitpid(-1, &status, 0);
```

it will be suspended until some child process terminates. If the child exits with, say, 4 as the parameter to *exit*, the parent will be awakened with *n* set to the child's PID and *status* set to 0x0400 (0x as a prefix means hexadecimal in C). The low-order byte of *status* relates to signals; the next one is the value the child returned in its call to exit.

If a process exits and its parent has not yet waited for it, the process enters a kind of suspended animation called the **zombie state**. When the parent finally waits for it, the process terminates.

Several system calls relate to signals, which are used in a variety of ways. For example, if a user accidentally tells a text editor to display the entire contents of a very long file, and then realizes the error, some way is needed to interrupt the editor. The usual choice is for the user to hit some special key (e.g., DEL or CTRL-C), which sends a signal to the editor. The editor catches the signal and stops the print-out.

To announce its willingness to catch this (or any other) signal, the process can use the sigaction system call. The first parameter is the signal to be caught (see Fig. 10-5). The second is a pointer to a structure giving a pointer to the signal handling procedure, as well as some other bits and flags. The third one points to a structure where the system returns information about signal handling currently in effect, in case it must be restored later.

The signal handler may run for as long as it wants to. In practice, though, signal handlers are usually fairly short. When the signal handling procedure is done, it returns to the point from which it was interrupted.

The sigaction system call can also be used to cause a signal to be ignored, or to restore the default action, which is killing the process.

Hitting the DEL key is not the only way to send a signal. The kill system call allows a process to signal another related process. The choice of the name "kill" for this system call is not an especially good one, since most processes send signals to other ones with the intention that they be caught.

For many real-time applications, a process needs to be interrupted after a specific time interval to do something, such as to retransmit a potentially lost packet over an unreliable communication line. To handle this situation, the alarm system call has been provided. The parameter specifies an interval, in seconds, after which a SIGALRM signal is sent to the process. A process may have only one alarm outstanding at any instant. If an alarm call is made with a parameter of 10 seconds, and then 3 seconds later another alarm call is made with a parameter of 20 seconds, only one signal will be generated, 20 seconds after the second call. The first signal is canceled by the second call to alarm. If the parameter to alarm is zero, any pending alarm signal is canceled. If an alarm signal is not caught, the default action is taken and the signaled process is killed. Technically, alarm signals may be ignored, but that is a pointless thing to do.

It sometimes occurs that a process has nothing to do until a signal arrives. For example, consider a computer-aided instruction program that is testing reading speed and comprehension. It displays some text on the screen and then calls alarm to signal it after 30 seconds. While the student is reading the text, the program has nothing to do. It could sit in a tight loop doing nothing, but that would waste CPU time that a background process or other user might need. A better solution is to use the pause system call, which tells Linux to suspend the process until the next signal arrives.

10.3.3 Implementation of Processes and Threads in Linux

A process in Linux is like an iceberg: what you see is the part above the water, but there is also an important part underneath. Every process has a user part that runs the user program. However, when one of its threads makes a system

call, it traps to kernel mode and begins running in kernel context, with a different memory map and full access to all machine resources. It is still the same thread, but now with more power and also its own kernel mode stack and kernel mode program counter. These are important because a system call can block part way through, for example, waiting for a disk operation to complete. The program counter and registers are then saved so the thread can be restarted in kernel mode later.

The Linux kernel internally represents processes as **tasks**, via the structure *task_struct*. Unlike other OS approaches, which make a distinction between a process, lightweight process, and thread), Linux uses the task structure to represent any execution context. Therefore, a single-threaded process will be represented with one task structure and a multithreaded process will have one task structure for each of the user-level threads. Finally, the kernel itself is multi-threaded, and has kernel level threads which are not associated with any user process and are executing kernel code. We will return to the treatment of multi-threaded processes (and threads in general) later in this section.

For each process, a process descriptor of type *task_ struct* is resident in memory at all times. It contains vital information needed for the kernel's management of all processes, including scheduling parameters, lists of open file descriptors, and so on. The process descriptor along with memory for the kernel-mode stack for the process are created upon process creation.

For compatibility with other UNIX systems, Linux identifies processes via the *Process Identifier (PID)*. The kernel organizes all processes in a doubly linked list of task structures. In addition to accessing process descriptors by traversing the linked lists, the PID can be mapped to the address of the task structure, and the process information can be accessed immediately.

The task structure contains a variety of fields. Some of these fields contain pointers to other data structures or segments, such as those containing information about open files. Some of these segments are related to the user-level structure of the process, which is not of interest when the user process is not runnable. Therefore, these may be swapped or paged out, in order not to waste memory on information that is not needed. For example, although it is possible for a process to be sent a signal while it is swapped out, it is not possible for it to read a file. For this reason, information about signals must be in memory all the time, even when the process is not present in memory. On the other hand, information about file descriptors can be kept in the user structure and brought in only when the process is in memory and runnable.

The information in the process descriptor falls into the following broad categories:

1. **Scheduling parameters**. Process priority, amount of CPU time consumed recently, amount of time spent sleeping recently. Together, these are used to determine which process to run next.

2. **Memory image**. Pointers to the text, data, and stack segments, or page tables. If the text segment is shared, the text pointer points to the shared text table. When the process is not in memory, information about how to find its parts on disk is here too.
3. **Signals**. Masks showing which signals are being ignored, which are being caught, which are being temporarily blocked, and which are in the process of being delivered.
4. **Machine registers**. When a trap to the kernel occurs, the machine registers (including the floating-point ones, if used) are saved here.
5. **System call state**. Information about the current system call, including the parameters, and results.
6. **File descriptor table**. When a system call involving a file descriptor is invoked, the file descriptor is used as an index into this table to locate the in-core data structure (i-node) corresponding to this file.
7. **Accounting**. Pointer to a table that keeps track of the user and system CPU time used by the process. Some systems also maintain limits here on the amount of CPU time a process may use, the maximum size of its stack, the number of page frames it may consume, and other items.
8. **Kernel stack**. A fixed stack for use by the kernel part of the process.
9. **Miscellaneous**. Current process state, event being waited for, if any, time until alarm clock goes off, PID, PID of the parent process, and user and group identification.

Keeping this information in mind, it is now easy to explain how processes are created in Linux. The mechanism for creating a new process is actually fairly straightforward. A new process descriptor and user area are created for the child process and filled in largely from the parent. The child is given a PID, its memory map is set up, and it is given shared access to its parent's files. Then its registers are set up and it is ready to run.

When a fork system call is executed, the calling process traps to the kernel and creates a task structure and few other accompanying data structures, such as the kernel mode stack and a *thread_info* structure. This structure is allocated at a fixed offset from the process' end-of-stack, and contains few process parameters, along with the address of the process descriptor. By storing the process descriptor's address at a fixed location, Linux needs only few efficient operations to locate the task structure for a running process.

The majority of the process descriptor contents are filled out based on the parent's descriptor values. Linux then looks for an available PID, and updates the PID hash table entry to point to the new task structure. In case of collisions in the

hash table, process descriptors may be chained. It also sets the fields in the *task_struct* to point to the corresponding previous/next process on the task array.

In principle, it should now allocate memory for the child's data and stack segments, and to make exact copies of the parent's segments, since the semantics of fork say that no memory is shared between parent and child. The text segment may either be copied or shared since it is read only. At this point, the child is ready to run.

However, copying memory is expensive, so all modern Linux systems cheat. They give the child its own page tables, but have them point to the parent's pages, only marked read only. Whenever the child tries to write on a page, it gets a protection fault. The kernel sees this and then allocates a new copy of the page to the child and marks it read/write. In this way, only pages that are actually written have to be copied. This mechanism is called **copy on write**. It has the additional benefit of not requiring two copies of the program in memory, thus saving RAM.

After the child process starts running, the code running there (a copy of the shell) does an exec system call giving the command name as a parameter. The kernel now finds and verifies the executable file, copies the arguments and environment strings to the kernel, and releases the old address space and its page tables.

Now the new address space must be created and filled in. If the system supports mapped files, as Linux and other UNIX-based systems do, the new page tables are set up to indicate that no pages are in memory, except perhaps one stack page, but that the address space is backed by the executable file on disk. When the new process starts running, it will immediately get a page fault, which will cause the first page of code to be paged in from the executable file. In this way, nothing has to be loaded in advance, so programs can start quickly and fault in just those pages they need and no more. (This strategy is demand paging in its purest form, as discussed in Chap. 3.) Finally, the arguments and environment strings are copied to the new stack, the signals are reset, and the registers are initialized to all zeros. At this point, the new command can start running.

Fig. 10-8 illustrates the steps described above through the following example: A user types a command, ls, on the terminal, the shell creates a new process by forking off a clone of itself. The new shell then calls exec to overlay its memory with the contents of the executable file *ls*.

Threads in Linux

We discussed threads in a general way in Chap. 2. Here we will focus on kernel threads in Linux, particularly focusing on the differences in the Linux thread model and other UNIX systems. In order to better understand the unique capabilities provided by the Linux model, we start with a discussion of some of the challenging decisions present in multithreaded systems.

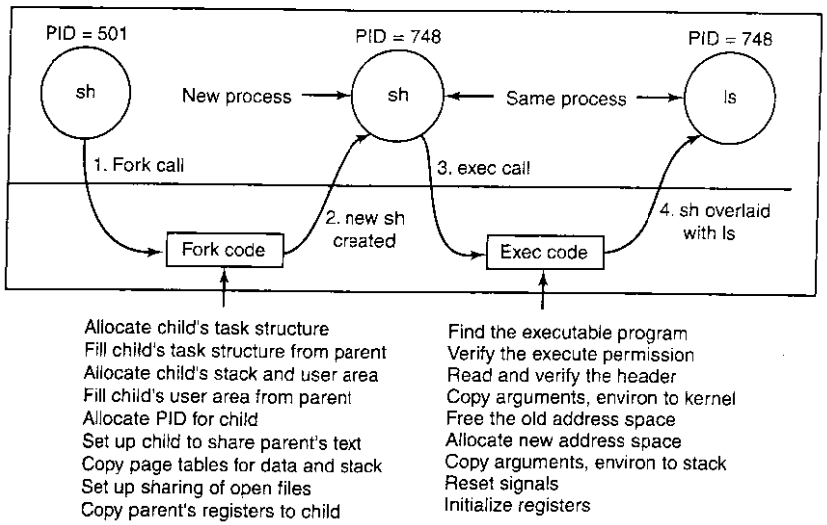

Figure 10-8. The steps in executing the command *ls* typed to the shell.

The main issue in introducing threads is maintaining the correct traditional UNIX semantics. First consider fork. Suppose that a process with multiple (kernel) threads does a fork system call. Should all the other threads be created in the new process? For the moment, let us answer that question with yes. Suppose that one of the other threads was blocked reading from the keyboard. Should the corresponding thread in the new process also be blocked reading from the keyboard? If so, which one gets the next line typed? If not, what should that thread be doing in the new process? The same problem holds for many other things threads can do. In a single-threaded process, the problem does not arise because the one and only thread cannot be blocked when calling fork. Now consider the case that the other threads are not created in the child process. Suppose that one of the not-created threads holds a mutex that the one-and-only thread in the new process tries to acquire after doing the fork. The mutex will never be released and the one thread will hang forever. Numerous other problems exist too. There is no simple solution.

File I/O is another problem area. Suppose that one thread is blocked reading from a file and another thread closes the file or does an lseek to change the current file pointer. What happens next? Who knows?

Signal handling is another thorny issue. Should signals be directed at a specific thread or at the process in general? A SIGFPE (floating-point exception) should probably be caught by the thread that caused it. What if it does not catch it? Should just that thread be killed, or all threads? Now consider the SIGINT

signal, generated by the user at the keyboard. Which thread should catch that? Should all threads share a common set of signal masks? All solutions to these and other problems usually cause something to break somewhere. Getting the semantics of threads right (not to mention the code) is a nontrivial business.

Linux supports kernel threads in an interesting way that is worth looking at. The implementation is based on ideas from 4.4BSD, but kernel threads were not enabled in that distribution because Berkeley ran out of money before the C library could be rewritten to solve the problems discussed above.

Historically, processes were resource containers and threads were the units of execution. A process contained one or more threads that shared the address space, open files, signal handlers, alarms, and everything else. Everything was clear and simple as described above.

In 2000, Linux introduced a powerful new system call, clone, that blurred the distinction between processes and threads and possibly even inverted the primacy of the two concepts. Clone is not present in any other version of UNIX. Classically, when a new thread was created, the original thread(s) and the new one shared everything but their registers. In particular, file descriptors for open files, signal handlers, alarms, and other global properties were per process, not per thread. What clone did was make it possible for each of these aspects and others to be process specific or thread specific. It is called as follows:

```
pid = clone(function, stack_ptr, sharing_flags, arg);
```

The call creates a new thread, either in the current process or in a new process, depending on *sharing_flags*. If the new thread is in the current process, it shares the address space with the existing threads, and every subsequent write to any byte in the address space by any thread is immediately visible to all the other threads in the process. On the other hand, if the address space is not shared, then the new thread gets an exact copy of the address space, but subsequent writes by the new thread are not visible to the old ones. These semantics are the same as POSIX fork.

In both cases, the new thread begins executing at *function*, which is called with *arg* as its only parameter. Also in both cases, the new thread gets its own private stack, with the stack pointer initialized to *stack_ptr*.

The *sharing_flags* parameter is a bitmap that allows a much finer grain of sharing than traditional UNIX systems. Each of the bits can be set independently of the other ones, and each of them determines whether the new thread copies some data structure or shares it with the calling thread. Fig. 10-9 shows some of the items that can be shared or copied according to bits in *sharing_flags*.

The *CLONE_VM* bit determines whether the virtual memory (i.e., address space) is shared with the old threads or copied. If it is set, the new thread just moves in with the existing ones, so the clone call effectively creates a new thread in an existing process. If the bit is cleared, the new thread gets its own private address space. Having its own address space means that the effect of its STORE

Flag	Meaning when set	Meaning when cleared
CLONE_VM	Create a new thread	Create a new process
CLONE_FS	Share umask, root, and working dirs	Do not share them
CLONE_FILES	Share the file descriptors	Copy the file descriptors
CLONE_SIGHAND	Share the signal handler table	Copy the table
CLONE_PID	New thread gets old PID	New thread gets own PID
CLONE_PARENT	New thread has same parent as caller	New thread's parent is caller

Figure 10-9. Bits in the *sharing_flags* bitmap.

instructions is not visible to the existing threads. This behavior is similar to fork, except as noted below. Creating a new address space is effectively the definition of a new process.

The *CLONE_FS* bit controls sharing of the root and working directories and of the umask flag. Even if the new thread has its own address space, if this bit is set, the old and new threads share working directories. This means that a call to chdir by one thread changes the working directory of the other thread, even though the other thread may have its own address space. In UNIX, a call to chdir by a thread always changes the working directory for other threads in its process, but never for threads in another process. Thus this bit enables a kind of sharing not possible in traditional UNIX versions.

The *CLONE_FILES* bit is analogous to the *CLONE_FS* bit. If set, the new thread shares its file descriptors with the old ones, so calls to lseek by one thread are visible to the other ones, again as normally holds for threads within the same process but not for threads in different processes. Similarly, *CLONE_SIGHAND* enables or disables the sharing of the signal handler table between the old and new threads. If the table is shared, even among threads in different address spaces, then changing a handler in one thread affects the handlers in the others. *CLONE_PID* controls whether the new thread gets its own PID or shares its parent's PID. This feature is needed during system booting. User processes are not permitted to enable it.

Finally, every process has a parent. The *CLONE_PARENT* bit controls who the parent of the new thread is. It can either be the same as the calling thread (in which case the new thread is a sibling of the caller) or it can be the calling thread itself, in which case the new thread is a child of the caller. There are a few other bits that control other items, but they are less important.

This fine-grained sharing is possible because Linux maintains separate data structures for the various items listed in Sec. 10.3.3 (scheduling parameters, memory image, and so on). The task structure just points to these data structures, so it is easy to make a new task structure for each cloned thread and have it point either to the old thread's scheduling, memory, and other data structures or to copies of

them. The fact that such fine-grained sharing is possible does not mean that it is useful, however, especially since traditional UNIX versions do not offer this functionality. A Linux program that takes advantage of it is then no longer portable to UNIX.

The Linux thread model raises another difficulty. UNIX systems associate a single PID with a process, independent of whether it is single- or multi-threaded. In order to be compatible with other UNIX systems, Linux distinguishes between a process identifier (PID) and a task identifier (TID). Both fields are stored in the task structure. When clone is used to create a new process that shares nothing with its creator, PID is set to a new value; otherwise, the task receives a new TID, but inherits the PID. In this manner all threads in a process will receive the same PID as the first thread in the process.

10.3.4 Scheduling in Linux

We will now look at the Linux scheduling algorithm. To start with, Linux threads are kernel threads, so scheduling is based on threads, not processes.

Linux distinguishes three classes of threads for scheduling purposes:

1. Real-time FIFO.
2. Real-time round robin.
3. Timesharing.

Real-time FIFO threads are the highest priority and are not preemptable except by a newly readied real-time FIFO thread with higher priority. Real-time round-robin threads are the same as real-time FIFO threads except that they have time quanta associated with them, and are preemptable by the clock. If multiple real-time round-robin threads are ready, each one is run for its quantum, after which it goes to the end of the list of real-time round-robin threads. Neither of these classes is actually real time in any sense. Deadlines cannot be specified and guarantees are not given. These classes are simply higher priority than threads in the standard timesharing class. The reason Linux calls them real time is that Linux is conformant to the P1003.4 standard ("real-time" extensions to UNIX) which uses those names. The real-time threads are internally represented with priority levels from 0 to 99, 0 being the highest and 99 the lowest real-time priority level.

The conventional, non-real-time threads are scheduled according to the following algorithm. Internally, the non-real-time threads are associated with priority levels from 100 to 139, that is, Linux internally distinguishes among 140 priority levels (for real-time and non-real-time tasks). As for the real-time round robin threads, Linux associates time quantum values for each of the nonreal-time priority levels. The quantum is the number of clock ticks the thread may continue to run for. In the current Linux version, the clock runs at 1000Hz and each tick is 1ms, which is called a **jiffy**.

Like most UNIX systems, Linux associates a nice value with each thread. The default is 0, but this can be changed using the nice(value) system call, where value ranges from −20 to +19. This value determines the static priority of each thread. A user computing π to a billion places in the background might put this call in his program to be nice to the other users. Only the system administrator may ask for *better* than normal service (meaning values from −20 to −1). Deducing the reason for this rule is left as an exercise for the reader.

A key data structure used by the Linux scheduler is a **runqueue**. A runqueue is associated with each CPU in the system, and among other information, it maintains two arrays, *active* and *expired*. As shown in Fig. 10-10, each of these fields is a pointer to an array of 140 list heads, each corresponding to a different priority. The list head points to a doubly linked list of processes at a given priority. The basic operation of the scheduler can be described as follows.

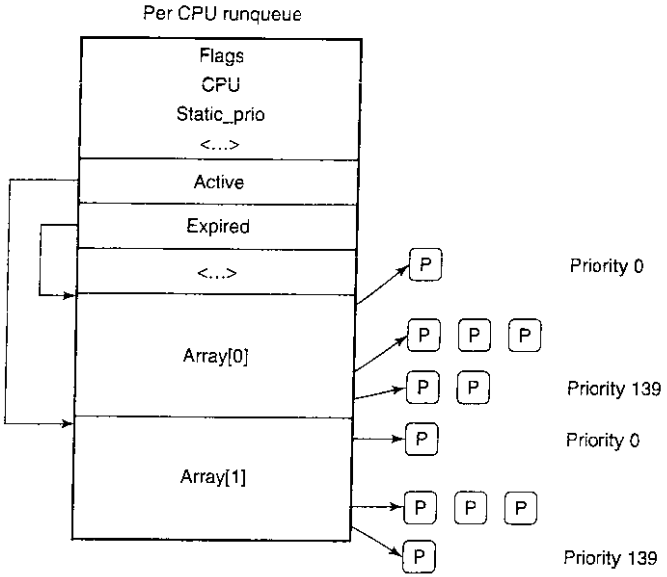

Figure 10-10. Illustration of Linux runqueue and priority arrays.

The scheduler selects a task from the highest-priority active array. If that task's timeslice (quantum) expires, it is moved to an expired list (potentially at a different priority level). If the task blocks, for instance to wait on an I/O event, before its timeslice expires, once the event occurs and its execution can resume, it is placed back on the original active array, and its timeslice is decremented to reflect the CPU time it already consumed. Once its timeslice is fully exhausted, it

too will be placed on an expired array. When there are no more tasks in any of the active arrays, the scheduler simply swaps the pointers, so the expired arrays now become active, and vice versa. This method ensures that low-priority tasks will not starve (except when real-time FIFO threads completely hog the CPU, which is unlikely to happen).

Different priority levels are assigned different timeslice values. Linux assigns higher quanta to higher-priority processes. For instance, tasks running at priority level 100 will receive time quanta of 800 msec, whereas tasks at priority level of 139 will receive 5 msec.

The idea behind this scheme is to get processes out of the kernel fast. If a process is trying to read a disk file, making it wait a second between read calls will slow it down enormously. It is far better to let it run immediately after each request is completed, so that it can make the next one quickly. Similarly, if a process was blocked waiting for keyboard input, it is clearly an interactive process, and as such should be given a high priority as soon as it is ready in order to ensure that interactive processes get good service. In this light, CPU-bound processes basically get any service that is left over when all the I/O bound and interactive processes are blocked.

Since Linux (or any other OS) does not know a priori whether a task is I/O- or CPU-bound, it relies on continuously maintaining interactivity heuristics. In this manner, Linux distinguishes between static and dynamic priority. The threads' dynamic priority is continuously recalculated, so as to (1) reward interactive threads, and (2) punish CPU-hogging threads. The maximum priority bonus is −5, since lower-priority values correspond to higher priority received by the scheduler. The maximum priority penalty is +5.

More specifically, the scheduler maintains a *sleep_avg* variable associated with each task. Whenever a task is awakened, this variable is incremented, whenever a task is preempted or its quantum expires, this variable is decremented by the corresponding value. This value is used to dynamically map the task's bonus to values from −5 to +5. The Linux scheduler recalculates the new priority level as a thread is moved from the active to the expired list.

The scheduling algorithm described in this section refers to the 2.6 kernel, and was first introduced in the unstable 2.5 kernel. Earlier algorithms exhibited poor performance in multiprocessor settings and did not scale well with an increased number of tasks. Since the description presented in the above paragraphs indicates that a scheduling decision can be made through access to the appropriate active list, it can be done in constant O(1) time, independent of the number of processes in the system.

In addition, the scheduler includes features particularly useful for multiprocessor or multicore platforms. First, the runqueue structure is associated with each CPU in the multiprocessing platform. The scheduler tries to maintain benefits from affinity scheduling, and to schedule tasks on the CPU on which they were previously executing. Second, a set of system calls is available to further specify

or modify the affinity requirements of a select thread. Finally, the scheduler performs periodic load balancing across runqueues of different CPUs to ensure that the system load is well balanced, while still meeting certain performance or affinity requirements.

The scheduler considers only runnable tasks, which are placed on the appropriate runqueue. Tasks which are not runnable and are waiting on various I/O operations or other kernel events are placed on another data structure, **waitqueue**. A waitqueue is associated with each event that tasks may wait on. The head of the waitqueue includes a pointer to a linked list of tasks and a spinlock. The spinlock is necessary so as to ensure that the waitqueue can be concurrently manipulated through both the main kernel code and interrupt handlers or other asynchronous invocations.

In fact, the kernel code contains synchronization variables in numerous locations. Earlier Linux kernels had just one **big kernel lock (BLK)**. This proved highly inefficient, particularly on multiprocessor platforms, since it prevented processes on different CPUs from executing kernel code concurrently. Hence, many new synchronization points were introduced at much finer granularity.

10.3.5 Booting Linux

Details vary from platform to platform, but in general the following steps represent the boot process. When the computer starts, the BIOS performs Power-On-Self-Test (POST) and initial device discovery and initialization, since the OS' boot process may rely on access to disks, screens, keyboards, and so on. Next, the first sector of the boot disk, the **MBR (Master Boot Record)**, is read into a fixed memory location and executed. This sector contains a small (512-byte) program that loads a standalone program called **boot** from the boot device, usually an IDE or SCSI disk. The *boot* program first copies itself to a fixed high-memory address to free up low memory for the operating system.

Once moved, *boot* reads the root directory of the boot device. To do this, it must understand the file system and directory format, which is the case with some bootloaders such as **GRUB (GRand Unified Bootloader)**. Other popular bootloaders, such as Intel's LILO, do not rely on any specific filesystem. Instead, they need a block map and low-level addresses, which describe physical sectors, heads, and cylinders, to find the relevant sectors to be loaded.

Then *boot* reads in the operating system kernel and jumps to it. At this point, it has finished its job and the kernel is running.

The kernel start-up code is written in assembly language and is highly machine dependent. Typical work includes setting up the kernel stack, identifying the CPU type, calculating the amount of RAM present, disabling interrupts, enabling the MMU, and finally calling the C-language *main* procedure to start the main part of the operating system.

The C code also has considerable initialization to do, but this is more logical than physical. It starts out by allocating a message buffer to help debug boot problems. As initialization proceeds, messages are written here about what is happening, so that they can be fished out after a boot failure by a special diagnostic program. Think of this as the operating system's cockpit flight recorder (the black box investigators look for after a plane crash).

Next the kernel data structures are allocated. Most are fixed size, but a few, such as the page cache and certain page table structures, depend on the amount of RAM available.

At this point the system begins autoconfiguration. Using configuration files telling what kinds of I/O devices might be present, it begins probing the devices to see which ones actually are present. If a probed device responds to the probe, it is added to a table of attached devices. If it fails to respond, it is assumed to be absent and ignored henceforth. Unlike traditional UNIX versions, Linux device drivers do not need to be statically linked and may be loaded dynamically (as can all versions of MS-DOS and Windows, incidentally).

The arguments for and against dynamically loading drivers are interesting and worth stating briefly. The main argument for dynamic loading is that a single binary can be shipped to customers with divergent configurations and have it automatically load the drivers it needs, possibly even over a network. The main argument against dynamic loading is security. If you are running a secure site, such as a bank's database or a corporate Web server, you probably want to make it impossible for anyone to insert random code into the kernel. The system administrator may keep the operating system sources and object files on a secure machine, do all system builds there, and ship the kernel binary to other machines over a local area network. If drivers cannot be loaded dynamically, this scenario prevents machine operators and others who know the superuser password from injecting malicious or buggy code into the kernel. Furthermore, at large sites, the hardware configuration is known exactly at the time the system is compiled and linked. Changes are sufficiently rare that having to relink the system when a new hardware device is added is not an issue.

Once all the hardware has been configured, the next thing to do is to carefully handcraft process 0, set up its stack, and run it. Process 0 continues initialization, doing things like programming the real-time clock, mounting the root file system, and creating *init* (process 1) and the page daemon (process 2).

Init checks its flags to see if it is supposed to come up single user or multiuser. In the former case, it forks off a process that executess the shell and waits for this process to exit. In the latter case, it forks off a process that executes the system initialization shell script, */etc/rc*, which can do file system consistency checks, mount additional file systems, start daemon processes, and so on. Then it reads */etc/ttys*, which lists the terminals and some of their properties. For each enabled terminal, it forks off a copy of itself, which does some housekeeping and then executess a program called *getty*.

Getty sets the line speed and other properties for each line (some of which may be modems, for example), and then types

```
login:
```

on the terminal's screen and tries to read the user's name from the keyboard. When someone sits down at the terminal and provides a login name, *getty* terminates by executing */bin/login*, the login program. *Login* then asks for a password, encrypts it, and verifies it against the encrypted password stored in the password file, */etc/passwd*. If it is correct, *login* replaces itself with the user's shell, which then waits for the first command. If it is incorrect, *login* just asks for another user name. This mechanism is shown in Fig. 10-11 for a system with three terminals.

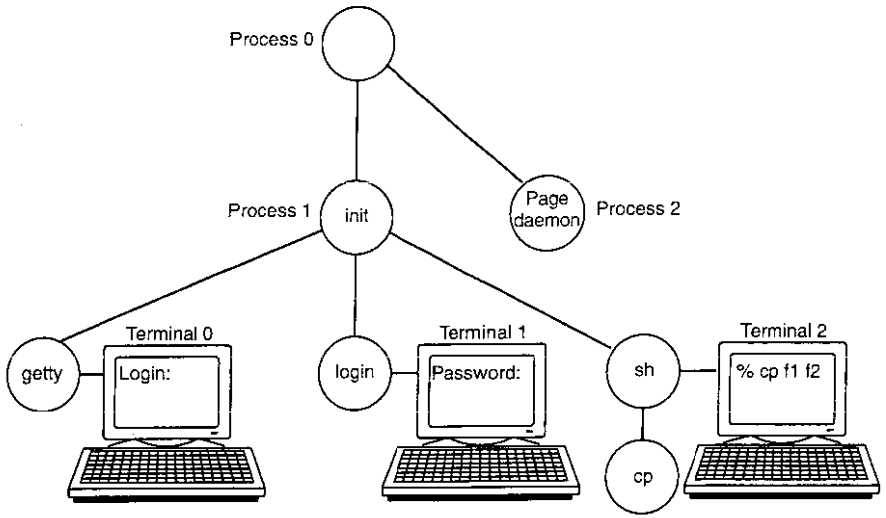

Figure 10-11. The sequence of processes used to boot some Linux systems.

In the figure, the *getty* process running for terminal 0 is still waiting for input. On terminal 1, a user has typed a login name, so *getty* has overwritten itself with *login*, which is asking for the password. A successful login has already occurred on terminal 2, causing the shell to type the prompt (%). The user then typed

```
cp f1 f2
```

which has caused the shell to fork off a child process and have that process execute the *cp* program. The shell is blocked, waiting for the child to terminate, at which time the shell will type another prompt and read from the keyboard. If the user at terminal 2 had typed *cc* instead of *cp*, the main program of the C compiler would have been started, which in turn would have forked off more processes to run the various compiler passes.

10.4 MEMORY MANAGEMENT IN LINUX

The Linux memory model is straightforward, to make programs portable and to make it possible to implement Linux on machines with widely differing memory management units, ranging from essentially nothing (e.g., the original IBM PC) to sophisticated paging hardware. This is an area of the design that has barely changed in decades. It has worked well so it has not needed much revision. We will now examine the model and how it is implemented.

10.4.1 Fundamental Concepts

Every Linux process has an address space logically consisting of three segments: text, data, and stack. An example process' address space is depicted in Fig. 10-12(a) as process *A*. The **text segment** contains the machine instructions that form the program's executable code. It is produced by the compiler and assembler by translating the C, C++, or other program into machine code. The text segment is normally read-only. Self-modifying programs went out of style in about 1950 because they were too difficult to understand and debug. Thus the text segment neither grows nor shrinks nor changes in any other way.

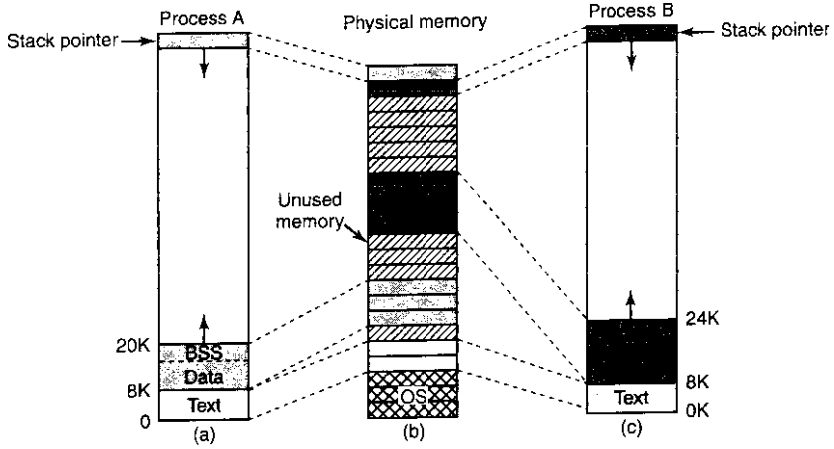

Figure 10-12. (a) Process *A*'s virtual address space. (b) Physical memory. (c) Process *B*'s virtual address space.

The **data segment** contains storage for all the program's variables, strings, arrays, and other data. It has two parts, the initialized data and the uninitialized data. For historical reasons, the latter is known as the **BSS** (historically called **Block Started by Symbol**). The initialized part of the data segment contains variables and compiler constants that need an initial value when the program is started. All the variables in the BSS part are initialized to zero after loading.

For example, in C it is possible to declare a character string and initialize it at the same time. When the program starts up, it expects that the string has its initial value. To implement this construction, the compiler assigns the string a location in the address space, and ensures that when the program is started up, this location contains the proper string. From the operating system's point of view, initialized data are not all that different from program text—both contain bit patterns produced by the compiler that must be loaded into memory when the program starts.

The existence of uninitialized data is actually just an optimization. When a global variable is not explicitly initialized, the semantics of the C language say that its initial value is 0. In practice, most global variables are not initialized explicitly, and are thus 0. This could be implemented by simply having a section of the executable binary file exactly equal to the number of bytes of data, and initializing all of them, including the ones that have defaulted to 0.

However, to save space in the executable file, this is not done. Instead, the file contains all the explicitly initialized variables following the program text. The uninitialized variables are all gathered together after the initialized ones, so all the compiler has to do is put a word in the header telling how many bytes to allocate.

To make this point more explicit, consider Fig. 10-12(a) again. Here the program text is 8 KB and the initialized data is also 8 KB. The uninitialized data (BSS) is 4 KB. The executable file is only 16 KB (text + initialized data), plus a short header that tells the system to allocate another 4 KB after the initialized data and zero it before starting the program. This trick avoids storing 4 KB of zeros in the executable file.

In order to avoid allocating a physical page frame full of zeros, during initialization Linux allocates a static *zero page*, a write-protected page full of zeros. When a process is loaded, its uninitialized data region is set to point to the zero page. Whenever a process actually attempts to write in this area, the copy-on-write mechanism kicks in, and an actual page frame is allocated to the process.

Unlike the text segment, which cannot change, the data segment can change. Programs modify their variables all the time. Furthermore, many programs need to allocate space dynamically, during execution. Linux handles this by permitting the data segment to grow and shrink as memory is allocated and deallocated. A system call, brk, is available to allow a program to set the size of its data segment. Thus to allocate more memory, a program can increase the size of its data segment. The C library procedure *malloc*, commonly used to allocate memory, makes heavy use of this system call. The process address space descriptor contains information on the range of dynamically allocated memory areas in the process, typically called **heap**.

The third segment is the stack segment. On most machines, it starts at or near the top of the virtual address space and grows down toward 0. For instance, on 32bit x86 platforms, the stack starts at address 0xC0000000, which is the 3-GB virtual address limit visible to the process in user mode. If the stack grows below the bottom of the stack segment, a hardware fault occurs and the operating system

lowers the bottom of the stack segment by one page. Programs do not explicitly manage the size of the stack segment.

When a program starts up, its stack is not empty. Instead, it contains all the environment (shell) variables as well as the command line typed to the shell to invoke it. In this way a program can discover its arguments. For example, when the command

```
cp src dest
```

is typed, the *cp* program is run with the string "cp src dest" on the stack, so it can find out the names of the source and destination files. The string is represented as an array of pointers to the symbols in the string, to make parsing easier.

When two users are running the same program, such as the editor, it would be possible, but inefficient, to keep two copies of the editor's program text in memory at once. Instead, most Linux systems support **shared text segments**. In Fig. 10-12(a) and Fig. 10-12(c) we see two processes, *A* and *B*, that have the same text segment. In Fig. 10-12(b) we see a possible layout of physical memory, in which both processes share the same piece of text. The mapping is done by the virtual memory hardware.

Data and stack segments are never shared except after a fork, and then only those pages that are not modified. If either one needs to grow and there is no room adjacent to it to grow into, there is no problem since adjacent virtual pages do not have to map onto adjacent physical pages.

On some computers, the hardware supports separate address spaces for instructions and data. When this feature is available, Linux can use it. For example, on a computer with 32-bit addresses, if this feature is available, there would be 2^{32} bits of address space for instructions and an additional 2^{32} bits of address space for the data and stack segments to share. A jump to 0 goes to address 0 of text space, whereas a move from 0 uses address 0 in data space. This feature doubles the address space available.

In addition to dynamically allocating more memory, processes in Linux can access file data through **memory-mapped files**. This feature makes it possible to map a file onto a portion of a process' address space so that the file can be read and written as if it were a byte array in memory. Mapping a file in makes random access to it much easier than using I/O system calls such as read and write. Shared libraries are accessed by mapping them in using this mechanism. In Fig. 10-13 we see a file that is mapped into two processes at the same time, at different virtual addresses.

An additional advantage of mapping a file in is that two or more processes can map in the same file at the same time. Writes to the file by any one of them are then instantly visible to the others. In fact, by mapping in a scratch file (which will be discarded after all the processes exit), this mechanism provides a high-bandwidth way for multiple processes to share memory. In the most extreme case, two (or more) processes could map in a file that covers the entire address

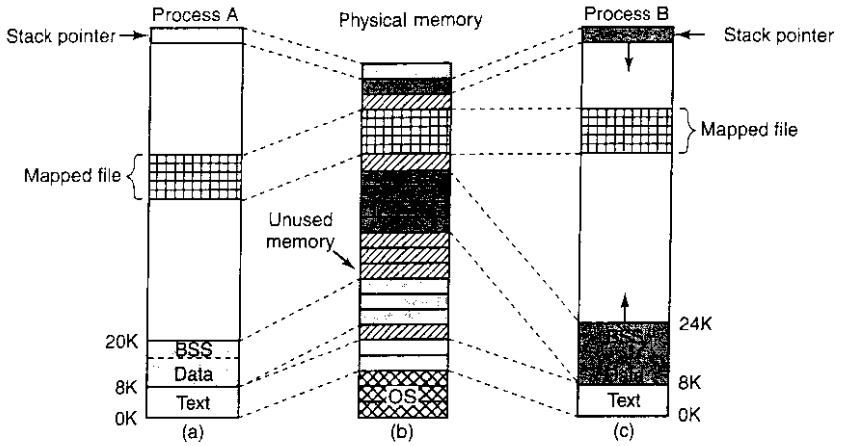

Figure 10-13. Two processes can share a mapped file.

space, giving a form of sharing that is partway between separate processes and threads. Here the address space is shared (like threads), but each process maintains its own open files and signals, for example, which is not like threads. In practice, making two address spaces exactly correspond is never done, however.

10.4.2 Memory Management System Calls in Linux

POSIX does not specify any system calls for memory management. This topic was considered too machine dependent for standardization. Instead, the problem was swept under the rug by saying that programs needing dynamic memory management can use the *malloc* library procedure (defined by the ANSI C standard). How *malloc* is implemented is thus moved outside the scope of the POSIX standard. In some circles, this approach is known as passing the buck.

In practice, most Linux systems have system calls for managing memory. The most common ones are listed in Fig. 10-14. Brk specifies the size of the data segment by giving the address of the first byte beyond it. If the new value is greater than the old one, the data segment becomes larger; otherwise it shrinks.

The mmap and munmap system calls control memory-mapped files. The first parameter to mmap, *addr*, determines the address at which the file (or portion thereof) is mapped. It must be a multiple of the page size. If this parameter is 0, the system determines the address itself and returns it in *a*. The second parameter, *len*, tells how many bytes to map. It, too, must be a multiple of the page size. The third parameter, *prot*, determines the protection for the mapped file. It can be marked readable, writable, executable, or some combination of these. The fourth

System call	Description
s = brk(addr)	Change data segment size
a = mmap(addr, len, prot, flags, fd, offset)	Map a file in
s = unmap(addr, len)	Unmap a file

Figure 10-14. Some system calls relating to memory management. The return code *s* is −1 if an error has occurred; *a* and *addr* are memory addresses, *len* is a length, *prot* controls protection, *flags* are miscellaneous bits, *fd* is a file descriptor, and *offset* is a file offset.

parameter, *flags*, controls whether the file is private or sharable, and whether *addr* is a requirement or merely a hint. The fifth parameter, *fd*, is the file descriptor for the file to be mapped. Only open files can be mapped, so to map a file in, it must first be opened. Finally, *offset* tells where in the file to begin the mapping. It is not necessary to start the mapping at byte 0; any page boundary will do.

The other call, unmap, removes a mapped file. If only a portion of the file is unmapped, the rest remains mapped.

10.4.3 Implementation of Memory Management in Linux

Each Linux process on a 32-bit machine typically gets 3 GB of virtual address space for itself, with the remaining 1 GB reserved for its page tables and other kernel data. The kernel's 1 GB is not visible when running in user mode, but becomes accessible when the process traps into the kernel. The kernel memory typically resides in low physical memory but it is mapped in the top 1 GB of each process virtual address space, between addresses 0xC0000000 and 0xFFFFFFFF (3–4 GB). The address space is created when the process is created and is overwritten on an exec system call.

In order to allow multiple processes to share the underlying physical memory, Linux monitors the use of the physical memory, allocates more memory as needed by user processes or kernel components, dynamically maps portions of the physical memory into the address space of different processes, and dynamically brings in and out of memory program executables, files and other state information as necessary to utilize the platform resources efficiently and to ensure execution progress. The remainder of this chapter describes the implementation of various mechanisms in the Linux kernel which are responsible for these operations.

Physical Memory Management

Due to idiosyncratic hardware limitations on many systems, not all physical memory can be treated identically, especially with respect to I/O and virtual memory. Linux distinguishes between three memory zones:

1. **ZONE_DMA** - pages that can be used for DMA operations.
2. **ZONE_NORMAL** - normal, regularly mapped pages.
3. **ZONE_HIGHMEM** - pages with high-memory addresses, which are not permanently mapped.

The exact boundaries and layout of the memory zones is architecture dependent. On x86 hardware, certain devices can perform DMA operations only in the first 16 MB of address space, hence ZONE_DMA is in the range 0–16 MB. In addition, the hardware cannot directly map memory addresses above 896 MB, hence ZONE_HIGHMEM is anything above this mark. ZONE_NORMAL is anything in between. Therefore, on x86 platforms, the first 896 MB of the Linux address space are directly mapped, whereas the remaining 128 MB of the kernel address space are used to access high memory regions. The kernel maintains a *zone* structure for each of the three zones, and can perform memory allocations for the three zones separately.

Main memory in Linux consists of three parts. The first two parts, the kernel and memory map, are **pinned** in memory (i.e., never paged out). The rest of memory is divided into page frames, each of which can contain a text, data, or stack page, a page table page, or be on the free list.

The kernel maintains a map of the main memory which contains all information about the use of the physical memory in the system, such as its zones, free page frames, and so forth. The information, illustrated in Fig. 10-15, is organized as follows.

First of all, Linux maintains an array of **page descriptors**, of type *page* for each physical page frame in the system, called *mem_map*. Each page descriptor contains a pointer to the address space it belongs to, in case the page is not free, a pair of pointers which allow it to form doubly linked lists with other descriptors, for instance to keep together all free page frames, and few other fields. In Fig. 10-15 the page descriptor for page 150 contains a mapping to the address space the page belongs to. Pages 70, 80 and 200 are free, and they are linked together. The size of the page descriptor is 32 bytes, therefore the entire mem_map can consume less than 1% of the physical memory (for a page frame of 4 KB).

Since the physical memory is divided into zones, for each zone Linux maintains a *zone descriptor*. The zone descriptor contains information about the memory utilization within each zone, such as number of active or inactive pages, low and high watermarks to be used by the page replacement algorithm described later in this chapter, as well as many other fields.

In addition, a zone descriptor contains an array of free areas. The i-th element in this array identifies the first page descriptor of the first block of 2^i free pages. Since there may be multiple blocks of 2^i free pages, Linux uses the pair of page descriptor pointers in each page element to link these together. This information is used in the memory allocation operations supported in Linux. In Fig. 10-15

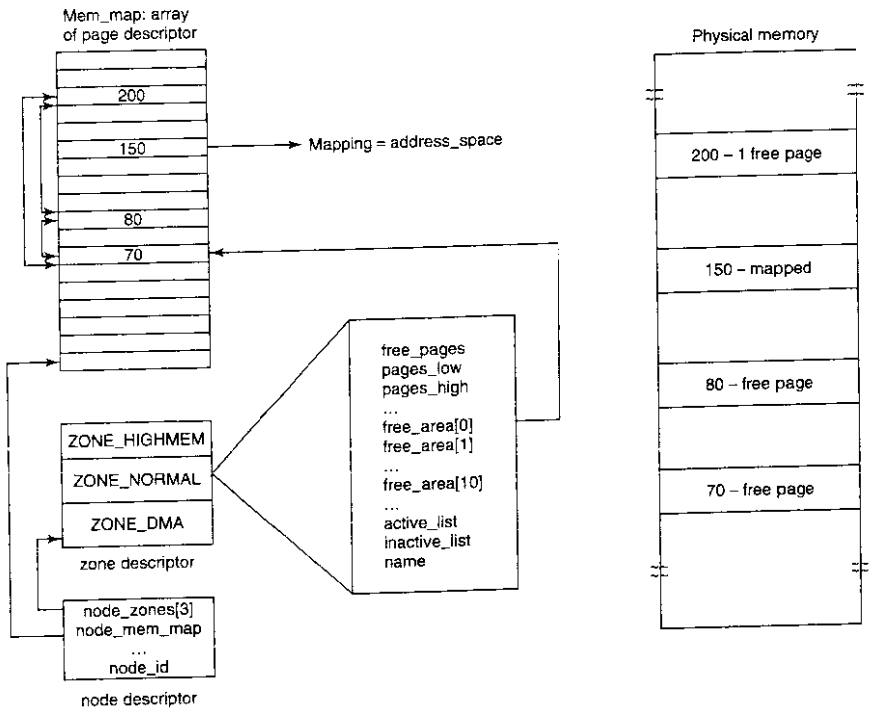

Figure 10-15. Linux main memory representation.

free_area[0], which identifies all free areas of main memory consisting of only one page frame (since 2^0 is one), points to page 70, the first of the three free areas. The other free blocks of size one can be reached through the links in each of the page descriptors.

Finally, since Linux is portable to NUMA architectures (where different memory addresses have very different access times), in order to differentiate between physical memory on different nodes (and avoid allocating data structures across nodes), a *node descriptor* is used. Each node descriptor contains information about the memory usage and zones on that particular node. On UMA platforms, Linux describes all memory via one node descriptor. The first few bits within each page descriptor are used to identify the node and the zone that the page frame belongs to.

In order for the paging mechanism to be efficient on 32- and 64-bit architecture, Linux uses a four-level paging scheme. A three-level paging scheme, originally put into the system for the Alpha, was expanded after Linux 2.6.10, and as of version 2.6.11 a four-level paging scheme is used. Each virtual address is

broken up into five fields, as shown in Fig. 10-16. The directory fields are used as an index into the appropriate page directory, of which there is a private one for each process. The value found is a pointer to one of the next-level directories, which are again indexed by a field from the virtual address. The selected entry in the middle page directory points to the final page table, which is indexed by the page field of the virtual address. The entry found here points to the page needed. On the Pentium, which uses two-level paging, each page's upper and middle directories have only one entry, so the global directory entry effectively chooses the page table to use. Similarly, three-level paging can be used when needed, by setting the size of the upper page directory field to zero.

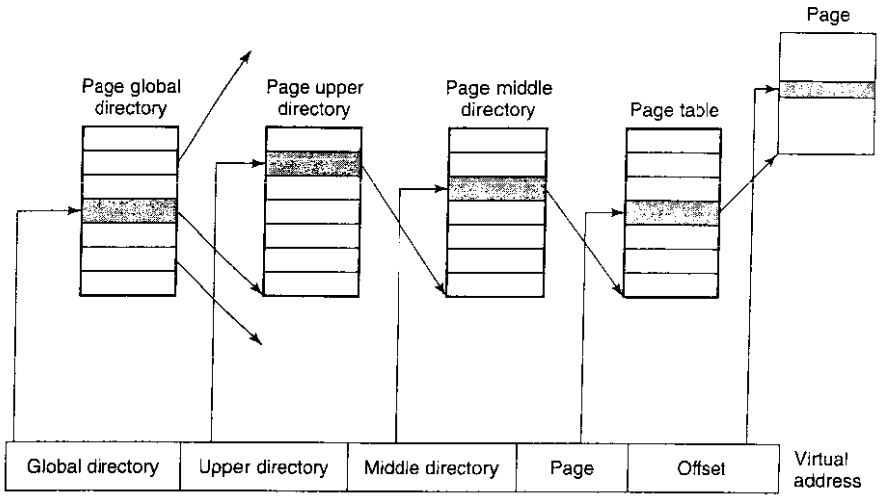

Figure 10-16. Linux uses four-level page tables.

Physical memory is used for various purposes. The kernel itself is fully hardwired; no part of it is ever paged out. The rest of memory is available for user pages, the paging cache, and other purposes. The page cache holds pages containing file blocks that have recently been read or have been read in advance in expectation of being used in the near future, or pages of file blocks which need to be written to disk, such as those which have been created from user mode processes which have been swapped out to disk. It is dynamic in size and competes for the same pool of pages as the user processes. The paging cache is not really a separate cache, but simply the set of user pages that are no longer needed and are waiting around to be paged out. If a page in the paging cache is reused before it is evicted from memory, it can be reclaimed quickly.

In addition, Linux supports dynamically loaded modules, most commonly device drivers. These can be of arbitrary size and each one must be allocated a

contiguous piece of kernel memory. As a direct consequence of these requirements, Linux manages physical memory in such a way that it can acquire an arbitrary-sized piece of memory at will. The algorithm it uses is known as the buddy algorithm and is described below.

Memory Allocation Mechanisms

Linux supports several mechanisms for memory allocation. The main mechanism for allocating new page frames of physical memory is the **page allocator**, which operates using the well-known **buddy algorithm**.

The basic idea for managing a chunk of memory is as follows. Initially memory consists of a single contiguous piece, 64 pages in the simple example of Fig. 10-17(a). When a request for memory comes in, it is first rounded up to a power of 2, say eight pages. The full memory chunk is then divided in half, as shown in (b). Since each of these pieces is still too large, the lower piece is divided in half again (c) and again (d). Now we have a chunk of the correct size, so it is allocated to the caller, as shown shaded in (d).

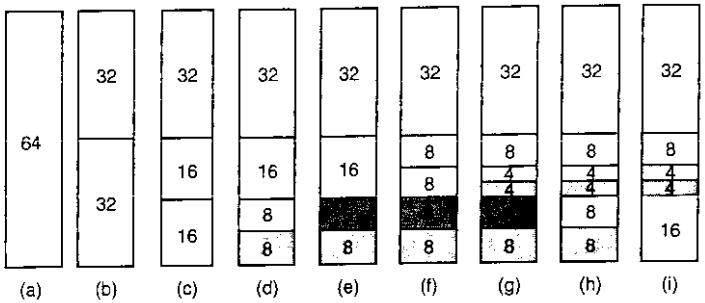

Figure 10-17. Operation of the buddy algorithm.

Now suppose that a second request comes in for eight pages. This can be satisfied directly now (e). At this point a third request comes in for four pages. The smallest available chunk is split (f) and half of it is claimed (g). Next, the second of the 8-page chunks is released (h). Finally, the other eight-page chunk is released. Since the two adjacent just-freed eight-page chunks came from the same 16-page chunk, they are merged to get the 16-page chunk back (i).

Linux manages memory using the buddy algorithm, with the additional feature of having an array in which the first element is the head of a list of blocks of size 1 unit, the second element is the head of a list of blocks of size 2 units, the next element points to the 4-unit blocks, and so on. In this way, any power-of-2 block can be found quickly.

This algorithm leads to considerable internal fragmentation because if you want a 65-page chunk, you have to ask for and get a 128-page chunk.

To alleviate this problem, Linux has a second memory allocation, the **slab allocator**, which takes chunks using the buddy algorithm but then carves slabs (smaller units) from them and manages the smaller units separately.

Since the kernel frequently creates and destroys objects of certain type (e.g., *task_struct*), it relies on so-called **object caches**. These caches consist of pointers to one or more slab which can store a number of objects of the same type. Each of the slabs may be full, partially full, or empty.

For instance, when the kernel needs to allocate a new process descriptor, that is, a new *task_struct* it looks in the object cache for task structures, and first tries to find a partially full slab and allocate a new *task_struct* object there. If no such slab is available, it looks through the list of empty slabs. Finally, if necessary, it will allocate a new slab, place the new task structure there, and link this slab with the task structure object cache. The kmalloc kernel service, which allocates physically contiguous memory regions in the kernel address space, is in fact built on top of the slab and object cache interface described here.

A third memory allocator, vmalloc, is also available and is used when the requested memory need only be contiguous in virtual space, but not in physical memory. In practice, this is true for most of the requested memory. One exception consists of devices, which live on the other side of the memory bus and the memory management unit, and therefore do not understand virtual addresses. However, the use of vmalloc results in some performance degradation, and is used primarily for allocating large amounts of contiguous virtual address space, such as for dynamically inserting kernel modules. All these memory allocators are derived from those in System V.

Virtual Address Space Representation

The virtual address space is divided into homogeneous, contiguous, page-aligned areas or regions. That is to say, each area consists of a run of consecutive pages with the same protection and paging properties. The text segment and mapped files are examples of areas (see Fig. 10-15). There can be holes in the virtual address space between the areas. Any memory reference to a hole results in a fatal page fault. The page size is fixed, for example, 4 KB for the Pentium and 8 KB for the Alpha. Starting with the Pentium, which supports page frames of 4 MB, Linux can support jumbo page frames of 4 MB each. In addition, in a **PAE (Physical Address Extension)** mode, which is used on certain 32-bit architectures to increase the process address space beyond 4 GB, page sizes of 2 MB are supported.

Each area is described in the kernel by a *vm_area_struct* entry. All the *vm_area_structs* for a process are linked together in a list sorted on virtual address so that all the pages can be found. When the list gets too long (more than 32 entries), a tree is created to speed up searching it. The *vm_area_struct* entry lists the area's properties. These properties include the protection mode (e.g., read only

or read/write), whether it is pinned in memory (not pageable), and which direction it grows in (up for data segments, down for stacks).

The *vm_area_struct* also records whether the area is private to the process or shared with one or more other processes. After a fork, Linux makes a copy of the area list for the child process, but sets up the parent and child to point to the same page tables. The areas are marked as read/write, but the pages are marked as read only. If either process tries to write on a page, a protection fault occurs and the kernel sees that the area is logically writable but the page is not, so it gives the process a copy of the page and marks it read/write. This mechanism is how copy on write is implemented.

The *vm_area_struct* also records whether the area has backing storage on disk assigned, and if so, where. Text segments use the executable binary as backing storage and memory-mapped files use the disk file as backing storage. Other areas, such as the stack, do not have backing storage assigned until they have to be paged out.

A top-level memory descriptor, *mm_struct*, gathers information about all virtual memory areas belonging to an address space, information about the different segments (text, data, stack), about users sharing this address space, and so on. All *vm_area_struct* elements of an address space can be accessed through their memory descriptor in two ways. First, they are organized in linked lists ordered by virtual memory addresses. This way is useful when all virtual memory areas need to be accessed, or when the kernel is searching to allocated a virtual memory region of a specific size. In addition, the *vm_area_struct* entries are organized in a binary "red-black" tree, a data structure optimized for fast lookups. This method is used when a specific virtual memory needs to be accessed. By enabling access to elements of the process address space via these two methods, Linux uses more state per process, but allows different kernel operations to use the access method which is more efficient for the task at hand.

10.4.4 Paging in Linux

Early UNIX systems relied on a **swapper process** to move entire processes between memory and disk whenever not all active processes could fit in the physical memory. Linux, like other modern UNIX versions, no longer moves entire processes. The main memory management unit is a page, and almost all memory management components operate on a page granularity. The swapping subsystem also operates on page granularity and is tightly coupled with the **Page Frame Reclaiming Algorithm**, described later in this section.

The basic idea behind paging in Linux is simple: a process need not be entirely in memory in order to run. All that is actually required is the user structure and the page tables. If these are swapped in, the process is deemed "in memory" and can be scheduled to run. The pages of the text, data, and stack segments are

brought in dynamically, one at a time, as they are referenced. If the user structure and page table are not in memory, the process cannot be run until the swapper brings them in.

Paging is implemented partly by the kernel and partly by a new process called the **page daemon**. The page daemon is process 2 (process 0 is the idle process—traditionally called the swapper—and process 1 is *init*, as shown in Fig. 10-11). Like all daemons, the page daemon runs periodically. Once awake, it looks around to see if there is any work to do. If it sees that the number of pages on the list of free memory pages is too low, it starts freeing up more pages.

Linux is a demand-paged system with no prepaging and no working set concept (although there is a system call in which a user can give a hint that a certain page may be needed soon, in the hope it will be there when needed). Text segments and mapped files are paged to their respective files on disk. Everything else is paged to either the paging partition (if present) or one of the fixed-length paging files, called the **swap area**. Paging files can be added and removed dynamically and each one has a priority. Paging to a separate partition, accessed as a raw device, is more efficient than paging to a file for several reasons. First, the mapping between file blocks and disk blocks is not needed (saves disk I/O reading indirect blocks). Second, the physical writes can be of any size, not just the file block size. Third, a page is always written contiguously to disk; with a paging file, it may or may not be.

Pages are not allocated on the paging device or partition until they are needed. Each device and file starts with a bitmap telling which pages are free. When a page without backing store has to be tossed out of memory, the highest-priority paging partition or file that still has space is chosen and a page allocated on it. Normally, the paging partition, if present, has higher priority than any paging file. The page table is updated to reflect that the page is no longer present in memory (e.g., the page-not-present bit is set) and the disk location is written into the page table entry.

The Page Replacement Algorithm

Page replacement works as follows. Linux tries to keep some pages free so that they can be claimed as needed. Of course, this pool must be continually replenished. The **PFRA** (**Page Frame Reclaiming Algorithm**) algorithm is how this happens.

First of all, Linux distinguishes between four different types of pages: *unreclaimable*, *swappable*, *syncable*, and *discardable* . Unreclaimable pages, which include reserved or locked pages, kernel mode stacks, and the like, may not be paged out. Swappable pages must be written back to the swap area or the paging disk partition before the page can be reclaimed. Syncable pages must be written back to disk if they have been marked as dirty. Finally, discardable pages can be reclaimed immediately.

At boot time, *init* starts up a page daemon, *kswapd*, one for each memory node, and configures them to run periodically. Each time *kswapd* awakens, it checks to see if there are enough free pages available, by comparing the low and high watermarks with the current memory usage for each memory zone. If there is enough memory, it goes back to sleep, although it can be awakened early if more pages are suddenly needed. If the available memory for any of the zones falls below a threshold, *kswapd* initiates the page frame reclaiming algorithm. During each run, only a certain target number of pages is reclaimed, typically 32. This number is limited to control the I/O pressure (the number of disk writes, created during the PFRA operations). Both, the number of reclaimed pages and the total number of scanned pages are configurable parameters.

Each time PFRA executes, it first tries to reclaim easy pages, then proceeds with the harder ones. Discardable and unreferenced pages can be reclaimed immediately by moving them onto the zone's freelist. Next it looks for pages with backing store which have not been referenced recently, using a clock-like algorithm. Following are shared pages that none of the users seems to be using much. The challenge with shared pages is that, if a page entry is reclaimed, the page tables of all address spaces originally sharing that page must be updated in a synchronous manner. Linux maintains efficient tree-like data structures to easily find all users of a shared page. Ordinary user pages are searched next, and if chosen to be evicted, they must be scheduled for write in the swap area. The **swappiness** of the system, that is, the ratio of pages with backing store versus pages which need to be swapped out selected during PFRA, is a tunable parameter of the algorithm. Finally, if a page is invalid, absent from memory, shared, locked in memory, or being used for DMA, it is skipped.

PFRA uses a clock-like algorithm to select old pages for eviction within a certain category. At the core of this algorithm is a loop which scans through each zone's active and inactive lists, trying to reclaim different kinds of pages, with different urgencies. The urgency value is passed as a parameter telling the procedure how much effort to expend to reclaim some pages. Usually, this means how many pages to inspect before giving up.

During PFRA, pages are moved between the active and inactive list in the manner described in Fig. 10-18. To maintain some heuristics and try to find pages which have not been referenced and are unlikely to be needed in the near future, PFRA maintains two flags per page: active/inactive, and referenced or not. These two flags encode four states, as shown in Fig. 10-18. During the first scan of a set of pages, PFRA first clears their reference bits. If during the second run over the page it is determined that it has been referenced, it is advanced to another state, from which it is less likely to be reclaimed. Otherwise, the page is moved to a state from where it is more likely to be evicted.

Pages on the inactive list, which have not been referenced since the last time they were inspected, are the best candidates for eviction. They are pages with both *PG_active* and *PG_referenced* set to zero in Fig. 10-18. However, if necessary,

pages may be reclaimed even if they are in some of the other states. The *refill* arrows in Fig. 10-18 illustrate this fact.

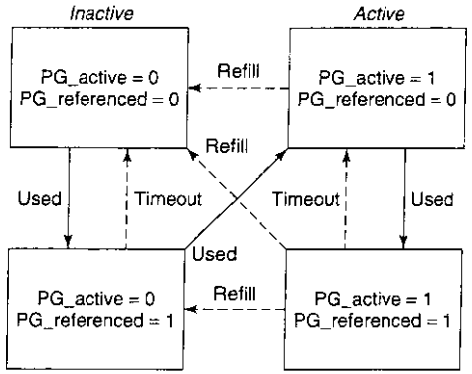

Figure 10-18. Page states considered in the page frame replacement algorithm.

The reason PRFA maintains pages in the inactive list although they might have been referenced, is to prevent situations such as the following. Consider a process which makes periodic accesses to different pages, with a 1-hour period. A page accessed since the last loop will have its reference flag set. However, since it will not be needed again for the next hour, there is no reason not to consider it as a candidate for reclamation.

One aspect of the memory management system that we have not yet mentioned is a second daemon, *pdflush*, actually a set of background daemon threads. The *pdflush* threads either (1) wake up periodically, typically each 500 msec, to write back to disk very old dirty pages, or (2) are explicitly awakened by the kernel when available memory levels fall below a certain threshold, to write back dirty pages from the page cache to disk. In **laptop mode**, in order to conserve battery life, dirty pages are written to disk whenever *pdflush* threads wake up. Dirty pages may also be written out to disk on explicit requests for synchronization, via systems calls such as sync, orfsync, fdatasync. Older Linux versions used two separate daemons: *kupdate*, for old page write back, and *bdflush*, for page write back under low memory conditions. In the 2.4 kernel this functionality was integrated in the *pdflush* threads. The choice of multiple threads was made in order to hide long disk latencies.

10.5 INPUT/OUTPUT IN LINUX

The I/O system in Linux is fairly straightforward and the same as other UNICES. Basically, all I/O devices are made to look like files and are accessed as such with the same read and write system calls that are used to access all ordinary

files. In some cases, device parameters must be set, and this is done using a special system call. We will study these issues in the following sections.

10.5.1 Fundamental Concepts

Like all computers, those running Linux have I/O devices such as disks, printers, and networks connected to them. Some way is needed to allow programs to access these devices. Although various solutions are possible, the Linux one is to integrate the devices into the file system as what are called **special files**. Each I/O device is assigned a path name, usually in */dev*. For example, a disk might be */dev/hd1*, a printer might be */dev/lp*, and the network might be */dev/net*.

These special files can be accessed the same way as any other files. No special commands or system calls are needed. The usual open, read, and write system calls will do just fine. For example, the command

```
cp file /dev/lp
```

copies the *file* to printer, causing it to be printed (assuming that the user has permission to access */dev/lp*). Programs can open, read, and write special files the same way as they do regular files. In fact, *cp* in the above example is not even aware that it is printing. In this way, no special mechanism is needed for doing I/O.

Special files are divided into two categories, block and character. A **block special file** is one consisting of a sequence of numbered blocks. The key property of the block special file is that each block can be individually addressed and accessed. In other words, a program can open a block special file and read, say, block 124 without first having to read blocks 0 to 123. Block special files are typically used for disks.

Character special files are normally used for devices that input or output a character stream. Keyboards, printers, networks, mice, plotters, and most other I/O devices that accept or produce data for people use character special files. It is not possible (or even meaningful) to seek to block 124 on a mouse.

Associated with each special file is a device driver that handles the corresponding device. Each driver has what is called a **major device** number that serves to identify it. If a driver supports multiple devices, say, two disks of the same type, each disk has a **minor device** number that identifies it. Together, the major and minor device numbers uniquely specify every I/O device. In few cases, a single driver handles two closely related devices. For example, the driver corresponding to */dev/tty* controls both the keyboard and the screen, which is often thought of as a single device, the terminal.

Although most character special files cannot be randomly accessed, they often need to be controlled in ways that block special files do not. Consider, for example, input typed on the keyboard and displayed on the screen. When a user makes a typing error and wants to erase the last character typed, he presses some

key. Some people prefer to use backspace, and others prefer DEL. Similarly, to erase the entire line just typed, many conventions abound. Traditionally @ was used, but with the spread of e-mail (which uses @ within e-mail address), many systems have adopted CTRL-U or some other character. Likewise, to interrupt the running program, some special key must be hit. Here, too, different people have different preferences. CTRL-C is a common choice, but it is not universal.

Rather than making a choice and forcing everyone to use it, Linux allows all these special functions and many others to be customized by the user. A special system call is generally provided for setting these options. This system call also handles tab expansion, enabling and disabling of character echoing, conversion between carriage return and line feed, and similar items. The system call is not permitted on regular files or block special files.

10.5.2 Networking

Another example of I/O is networking, as pioneered by Berkeley UNIX and taken over by Linux more or less verbatim. The key concept in the Berkeley design is the **socket**. Sockets are analogous to mailboxes and telephone wall sockets in that they allow users to interface to the network, just as mailboxes allow people to interface to the postal system and telephone wall sockets allow them to plug in telephones and connect to the telephone system. The sockets' position is shown in Fig. 10-19.

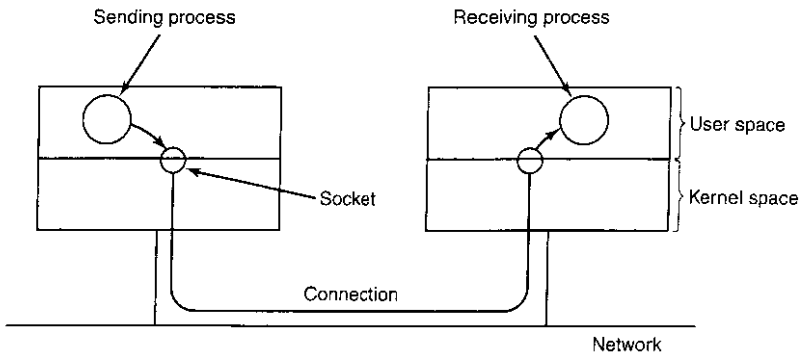

Figure 10-19. The uses of sockets for networking.

Sockets can be created and destroyed dynamically. Creating a socket returns a file descriptor, which is needed for establishing a connection, reading data, writing data, and releasing the connection.

Each socket supports a particular type of networking, specified when the socket is created. The most common types are

1. Reliable connection-oriented byte stream.
2. Reliable connection-oriented packet stream.
3. Unreliable packet transmission.

The first socket type allows two processes on different machines to establish the equivalent of a pipe between them. Bytes are pumped in at one end and they come out in the same order at the other. The system guarantees that all bytes that are sent arrive and in the same order they were sent.

The second type is similar to the first one, except that it preserves packet boundaries. If the sender makes five separate calls to write, each for 512 bytes, and the receiver asks for 2560 bytes, with a type 1 socket all 2560 bytes will be returned. Four returned at once. With a type 2 socket, only 512 bytes will be returned. Four more calls are needed to get the rest. The third type of socket is used to give the user access to the raw network. This type is especially useful for real-time applications, and for those situations in which the user wants to implement a specialized error-handling scheme. Packets may be lost or reordered by the network. There are no guarantees, as in the first two cases. The advantage of this mode is higher performance, which sometimes outweighs reliability (e.g., for multimedia delivery, in which being fast counts for more than being right).

When a socket is created, one of the parameters specifies the protocol to be used for it. For reliable byte streams, the most popular protocol is **TCP** (**Transmission Control Protocol**). For unreliable packet-oriented transmission, **UDP** (**User Datagram Protocol**) is the usual choice. Both are these are layered on top of **IP** (**Internet Protocol**). All of these protocols originated with the U.S. Dept. of Defense's ARPANET, and now form the basis of the Internet. There is no common protocol for reliable packet streams.

Before a socket can be used for networking, it must have an address bound to it. This address can be in one of several naming domains. The most common domain is the Internet naming domain, which uses 32-bit integers for naming endpoints in Version 4 and 128-bit integers in Version 6 (Version 5 was an experimental system that never made it to the major leagues).

Once sockets have been created on both the source and destination computers, a connection can be established between them (for connection-oriented communication). One party makes a listen system call on a local socket, which creates a buffer and blocks until data arrive. The other makes a connect system call, giving as parameters the file descriptor for a local socket and the address of a remote socket. If the remote party accepts the call, the system then establishes a connection between the sockets.

Once a connection has been established, it functions analogously to a pipe. A process can read and write from it using the file descriptor for its local socket. When the connection is no longer needed, it can be closed in the usual way, via the close system call.

10.5.3 Input/Output System Calls in Linux

Each I/O device in a Linux system generally has a special file associated with it. Most I/O can be done by just using the proper file, eliminating the need for special system calls. Nevertheless, sometimes there is a need for something that is device specific. Prior to POSIX most UNIX systems had a system call ioctl that performed a large number of device-specific actions on special files. Over the course of the years, it had gotten to be quite a mess. POSIX cleaned it up by splitting its functions into separate function calls primarily for terminal devices. In Linux and modern UNIX systems, whether each one is a separate system call or they share a single system call or something else is implementation dependent.

The first four calls listed in Fig. 10-20 are used to set and get the terminal speed. Different calls are provided for input and output because some modems operate at split speed. For example, old videotex systems allowed people to access public databases with short requests from the home to the server at 75 bits/sec with replies coming back at 1200 bits/sec. This standard was adopted at a time when 1200 bits/sec both ways was too expensive for home use. Times change in the networking world. This asymmetry still persists, with some telephone companies offering inbound service at 8 Mbps and outbound service at 512 kbps, often under the name of **ADSL** (**Asymmetric Digital Subscriber Line**).

Function call	Description
s = cfsetospeed(&termios, speed)	Set the output speed
s = cfsetispeed(&termios, speed)	Set the input speed
s = cfgetospeed(&termios, speed)	Get the output speed
s = cfgetispeed(&termios, speed)	Get the input speed
s = tcsetattr(fd, opt, &termios)	Set the attributes
s = tcgetattr(fd, &termios)	Get the attributes

Figure 10-20. The main POSIX calls for managing the terminal.

The last two calls in the list are for setting and reading back all the special characters used for erasing characters and lines, interrupting processes, and so on. In addition, they enable and disable echoing, handle flow control, and other related functions. Additional I/O function calls also exist, but they are somewhat specialized, so we will not discuss them further. In addition, ioctl is still available.

10.5.4 Implementation of Input/Output in Linux

I/O in Linux is implemented by a collection of device drivers, one per device type. The function of the drivers is to isolate the rest of the system from the idiosyncracies of the hardware. By providing standard interfaces between the

drivers and the rest of the operating system, most of the I/O system can be put into the machine-independent part of the kernel.

When the user accesses a special file, the file system determines the major and minor device numbers belonging to it and whether it is a block special file or a character special file. The major device number is used to index into one of two internal hash tables containing data structures for character or block devices. The structure thus located contains pointers to the procedures to call to open the device, read the device, write the device, and so on. The minor device number is passed as a parameter. Adding a new device type to Linux means adding a new entry to one of these tables and supplying the corresponding procedures to handle the various operations on the device.

Some of the operations which may be associated with different character devices are shown in Fig. 10-21. Each row refers to a single I/O device (i.e., a single driver). The columns represent the functions that all character drivers must support. Several other functions also exist. When an operation is performed on a character special file, the system indexes into the hash table of character devices to select the proper structure, then calls the corresponding function to have the work performed. Thus each of the file operations contains a pointer to a function contained in the corresponding driver.

Device	Open	Close	Read	Write	Ioctl	Other
Null	null	null	null	null	null	...
Memory	null	null	mem_read	mem_write	null	...
Keyboard	k_open	k_close	k_read	error	k_ioctl	...
Tty	tty_open	tty_close	tty_read	tty_write	tty_ioctl	...
Printer	lp_open	lp_close	error	lp_write	lp_ioctl	...

Figure 10-21. Some of the file operations supported for typical character devices.

Each driver is split into two parts, both of which are part of the Linux kernel and both of which run in kernel mode. The top half runs in the context of the caller and interfaces to the rest of Linux. The bottom half runs in kernel context and interacts with the device. Drivers are allowed to make calls to kernel procedures for memory allocation, timer management, DMA control, and other things. The set of kernel functions that may be called is defined in a document called the **Driver-Kernel Interface**. Writing device drivers for Linux is covered in detail in (Egan and Teixeira, 1992; Rubini et al., 2005).

The I/O system is split into two major components: the handling of block special files and the handling of character special files. We will now look at each of these components in turn.

The goal of the part of the system that does I/O on block special files (e.g., disks) is to minimize the number of transfers that must be done. To accomplish

this goal, Linux systems have a **cache** between the disk drivers and the file system, as illustrated in Fig. 10-22. Prior to the 2.2 kernel, Linux maintained completely separate page and buffer caches, so a file residing in a disk block could be cached in both caches. Newer versions of Linux have a unified cache. A *generic block layer* holds these components together, performs the necessary translations between disk sectors, blocks, buffers and pages of data, and enables the operations on them.

The cache is a table in the kernel for holding thousands of the most recently used blocks. When a block is needed from a disk for any purpose (i-node, directory, or data), a check is first made to see if it is in the cache. If so, it is taken from there and a disk access is avoided, thereby resulting in great improvements in system performance.

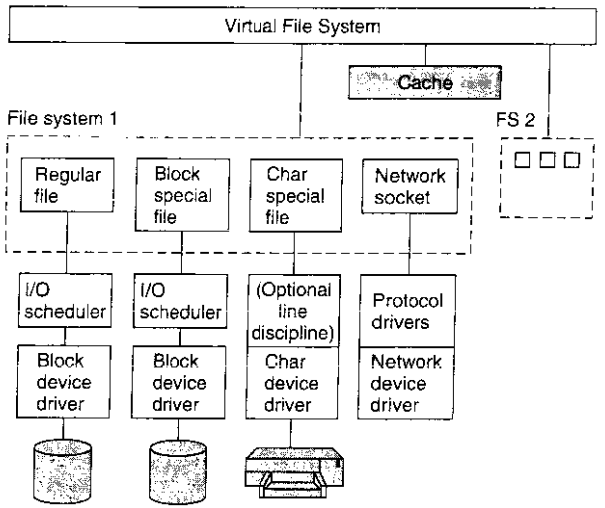

Figure 10-22. The Linux I/O system showing one file system in detail.

If the block is not in the page cache, it is read from the disk into the cache and from there copied to where it is needed. Since the page cache has room for only a fixed number of blocks, the page replacement algorithm described in the previous section is invoked.

The page cache works for writes as well as for reads. When a program writes a block, it goes to the cache, not to the disk. The *pdflush* daemon will flush the block to disk in the event the cache grows above a specified value. In addition, to avoid having blocks stay too long in the cache before being written to the disk, all the dirty blocks are written to the disk every 30 seconds.

In order to minimize the latency of repetitive disk head movements, Linux relies on an **I/O scheduler**. The purpose of the I/O scheduler is to reorder or

bundle read/write requests to block devices. There are many scheduler variants, optimized for different types of workloads. The basic Linux scheduler is based on the original **Linus Elevator scheduler**. The operations of the elevator scheduler can be summarized as follows: Disk operations are sorted in a doubly linked list, ordered by the address of the sector of the disk request. New requests are inserted in this list in a sorted manner. This prevents repeated costly disk head movements. The request list is than *merged* so that adjacent operations are issued via a single disk request. The basic elevator scheduler can lead to starvation. Therefore, the revised version of the Linux disk scheduler includes two additional lists, maintaining read or write operations ordered by their deadlines. The default deadlines are 0.5 sec for read requests and 5 sec for write requests. If a system-defined deadline for the oldest write operation is about to expire, that write request will be serviced before any of the requests on the main doubly linked list.

In addition to regular disk files, there are also block special files, also called **raw block files**. These files allow programs to access the disk using absolute block numbers, without regard to the file system. They are most often used for things like paging and system maintenance.

The interaction with character devices is simple. Since character devices produce or consume streams of characters, or bytes of data, support for random access makes little sense. One exception is the use of **line disciplines**. A line discipline can be associated with a terminal device, represented via the structure *tty_struct*, and it represents an interpreter for the data exchanged with the terminal device. For instance, local line editing can be done (i.e., erased characters and lines can be removed), carriage returns can be mapped onto line feeds, and other special processing can be completed. However, if a process wants to interact on every character, it can put the line in raw mode, in which case the line discipline will be bypassed. Not all devices have line disciplines.

Output works in a similar way, expanding tabs to spaces, converting line feeds to carriage returns + line feeds, adding filler characters following carriage returns on slow mechanical terminals, and so on. Like input, output can go through the line discipline (cooked mode) or bypass it (raw mode). Raw mode is especially useful when sending binary data to other computers over a serial line and for GUIs. Here, no conversions are desired.

The interaction with **network devices** is somewhat different. While network devices also produce/consume streams of characters, their asynchronous nature makes them less suitable for easy integration under the same interface as other character devices. The networking device driver produces packets consisting of multiple bytes of data, along with network headers. These packets are then routed through a series of network protocol drivers, and ultimately are passed to the user space application. A key data structure is the socket buffer structure, *skbuff*, which is used to represent portions of memory filled with packet data. The data in an *skbuff* buffer does not always start at the start of the buffer. As they are being processed by various protocols in the networking stack, protocol headers may be

removed, or added. The user processes interact with networking devices via sockets, which in Linux support the original BSD socket API. The protocol drivers can be bypassed and direct access to the underlying network device is enabled via *raw_sockets*. Only superusers are allowed to create raw sockets.

10.5.5 Modules in Linux

For decades, UNIX device drivers have been statically linked into the kernel so they were all present in memory whenever the system was booted. Given the environment in which UNIX grew up, mostly departmental minicomputers and then high-end workstations, with their small and unchanging sets of I/O devices, this scheme worked well. Basically, a computer center built a kernel containing drivers for the I/O devices and that was it. If next year the center bought a new disk, it relinked the kernel. No big deal.

With the arrival of Linux on the PC platform, suddenly all that changed. The number of I/O devices available on the PC is orders of magnitude larger than on any minicomputer. In addition, although all Linux users have (or can easily get) the full source code, probably the vast majority would have considerable difficulty adding a driver, updating all the device-driver related data structures, relinking the kernel, and then installing it as the bootable system (not to mention dealing with the aftermath of building a kernel that does not boot).

Linux solved this problem with the concept of **loadable modules**. These are chunks of code that can be loaded into the kernel while the system is running. Most commonly these are character or block device drivers, but they can also be entire file systems, network protocols, performance monitoring tools, or anything else desired.

When a module is loaded, several things have to happen. First, the module has to be relocated on the fly, during loading. Second, the system has to check to see if the resources the driver needs are available (e.g., interrupt request levels) and if so, mark them as in use. Third, any interrupt vectors that are needed must be set up. Fourth, the appropriate driver switch table has to be updated to handle the new major device type. Finally, the driver is allowed to run to perform any device-specific initialization it may need. Once all these steps are completed, the driver is fully installed, the same as any statically installed driver. Other modern UNIX systems now also support loadable modules.

10.6 THE LINUX FILE SYSTEM

The most visible part of any operating system, including Linux, is the file system. In the following sections we will examine the basic ideas behind the Linux file system, the system calls, and how the file system is implemented. Some of

these ideas derive from MULTICS, and many of them have been copied by MS-DOS, Windows, and other systems, but others are unique to UNIX-based systems. The Linux design is especially interesting because it clearly illustrates the principle of *Small is Beautiful*. With minimal mechanism and a very limited number of system calls, Linux nevertheless provides a powerful and elegant file system.

10.6.1 Fundamental Concepts

The initial Linux file system was the MINIX 1 file system. However, due to the fact that it limited file names to 14 characters (in order to be compatible with UNIX Version 7) and its maximum file size was 64 MB (which was overkill on the 10-MB hard disks of its era), there was interest in better file systems almost from the beginning of the Linux development, which began about 5 years after MINIX 1 was released. The first improvement was the ext file system, which allowed file names of 255 characters and files of 2 GB, but it was slower than the MINIX 1 file system, so the search continued for a while. Eventually, the ext2 file system was invented, with long file names, long files, and better performance, and it has become the main file system. However, Linux supports several dozen file systems using the Virtual File System (VFS) layer (described in the next section). When Linux is linked, a choice is offered of which file systems should be built into the kernel. Others can be dynamically loaded as modules during execution, if need be.

A Linux file is a sequence of 0 or more bytes containing arbitrary information. No distinction is made between ASCII files, binary files, or any other kinds of files. The meaning of the bits in a file is entirely up to the file's owner. The system does not care. File names are limited to 255 characters, and all the ASCII characters except NUL are allowed in file names, so a file name consisting of three carriage returns is a legal file name (but not an especially convenient one).

By convention, many programs expect file names to consist of a base name and an extension, separated by a dot (which counts as a character). Thus *prog.c* is typically a C program, *prog.f90* is typically a FORTRAN 90 program, and *prog.o* is usually an object file (compiler output). These conventions are not enforced by the operating system but some compilers and other programs expect them. Extensions may be of any length, and files may have multiple extensions, as in *prog.java.gz*, which is probably a *gzip* compressed Java program.

Files can be grouped together in directories for convenience. Directories are stored as files and to a large extent can be treated like files. Directories can contain subdirectories, leading to a hierarchical file system. The root directory is called / and usually contains several subdirectories. The / character is also used to separate directory names, so that the name */usr/ast/x* denotes the file *x* located in the directory *ast*, which itself is in the */usr* directory. Some of the major directories near the top of the tree are shown in Fig. 10-23.

Directory	Contents
bin	Binary (executable) programs
dev	Special files for I/O devices
etc	Miscellaneous system files
lib	Libraries
usr	User directories

Figure 10-23. Some important directories found in most Linux systems.

There are two ways to specify file names in Linux, both to the shell and when opening a file from within a program. The first way is using an **absolute path**, which means telling how to get to the file starting at the root directory. An example of an absolute path is */usr/ast/books/mos3/chap-10*. This tells the system to look in the root directory for a directory called *usr*, then look there for another directory, *ast*. In turn, this directory contains a directory *books*, which contains the directory *mos3*, which contains the file *chap-10*.

Absolute path names are often long and inconvenient. For this reason, Linux allows users and processes to designate the directory in which they are currently working as the **working directory**. Path names can also be specified relative to the working directory. A path name specified relative to the working directory is a **relative path**. For example, if */usr/ast/books/mos3* is the working directory, then the shell command

```
cp chap-10 backup-10
```

has exactly the same effect as the longer command

```
cp /usr/ast/books/mos3/chap-10 /usr/ast/books/mos3/backup-10
```

It frequently occurs that a user needs to refer to a file that belongs to another user, or at least is located elsewhere in the file tree. For example, if two users are sharing a file, it will be located in a directory belonging to one of them, so the other will have to use an absolute path name to refer to it (or change the working directory). If this is long enough, it may become irritating to have to keep typing it. Linux provides a solution to this problem by allowing users to make a new directory entry that points to an existing file. Such an entry is called a **link**.

As an example, consider the situation of Fig. 10-24(a). Fred and Lisa are working together on a project, and each of them needs access to the other's files. If Fred has */usr/fred* as his working directory, he can refer to the file *x* in Lisa's directory as */usr/lisa/x*. Alternatively, Fred can create a new entry in his directory, as shown in Fig. 10-24(b), after which he can use *x* to mean */usr/lisa/x*.

In the example just discussed, we suggested that before linking, the only way for Fred to refer to Lisa's file *x* was by using its absolute path. Actually, this is not really true. When a directory is created, two entries, . and .., are automatically

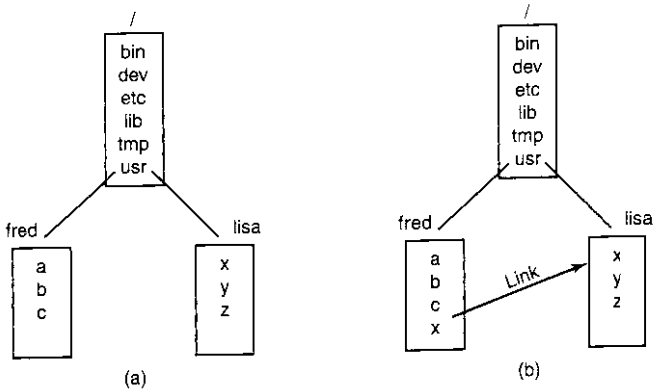

Figure 10-24. (a) Before linking. (b) After linking.

made in it. The former refers to the working directory itself. The latter refers to the directory's parent, that is, the directory in which it itself is listed. Thus from */usr/fred*, another path to Lisa's file *x* is *../lisa/x*.

In addition to regular files, Linux also supports character special files and block special files. Character special files are used to model serial I/O devices, such as keyboards and printers. Opening and reading from */dev/tty* reads from the keyboard; opening and writing to */dev/lp* writes to the printer. Block special files, often with names like */dev/hd1*, can be used to read and write raw disk partitions without regard to the file system. Thus a seek to byte *k* followed by a read will begin reading from the *k*-th byte on the corresponding partition, completely ignoring the i-node and file structure. Raw block devices are used for paging and swapping by programs that lay down file systems (e.g., *mkfs*), and by programs that fix sick file systems (e.g., *fsck*), for example.

Many computers have two or more disks. On mainframes at banks, for example, it is frequently necessary to have 100 or more disks on a single machine, in order to hold the huge databases required. Even personal computers normally have at least two disks—a hard disk and an optical (e.g., DVD) drive. When there are multiple disk drives, the question arises of how to handle them.

One solution is to put a self-contained file system on each one and just keep them separate. Consider, for example, the situation depicted in Fig. 10-25(a). Here we have a hard disk, which we will call *C:*, and a DVD, which we will call *D:*. Each has its own root directory and files. With this solution, the user has to specify both the device and the file when anything other than the default is needed. For example, to copy the file *x* to the directory *d* (assuming *C:* is the default), one would type

```
cp D:/x /a/d/x
```


This is the approach taken by a number of systems, including MS-DOS, Windows 98, and VMS.

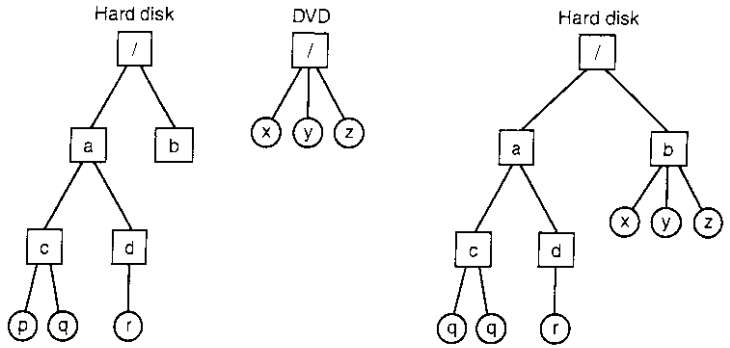

Figure 10-25. (a) Separate file systems. (b) After mounting.

The Linux solution is to allow one disk to be mounted in another disk's file tree. In our example, we could mount the DVD on the directory */b*, yielding the file system of Fig. 10-25(b). The user now sees a single file tree, and no longer has to be aware of which file resides on which device. The above copy command now becomes

```
cp /b/x /a/d/x
```

exactly the same as it would have been if everything had been on the hard disk in the first place.

Another interesting property of the Linux file system is **locking**. In some applications, two or more processes may be using the same file at the same time, which may lead to race conditions. One solution is to program the application with critical regions. However, if the processes belong to independent users who do not even know each other, this kind of coordination is generally inconvenient.

Consider, for example, a database consisting of many files in one or more directories that are accessed by unrelated users. It is certainly possible to associate a semaphore with each directory or file and achieve mutual exclusion by having processes do a down operation on the appropriate semaphore before accessing the data. The disadvantage, however, is that a whole directory or file is then made inaccessible, even though only one record may be needed.

For this reason, POSIX provides a flexible and fine-grained mechanism for processes to lock as little as a single byte and as much as an entire file in one indivisible operation. The locking mechanism requires the caller to specify the file to be locked, the starting byte, and the number of bytes. If the operation succeeds, the system makes a table entry noting that the bytes in question (e.g., a database record) are locked.

Two kinds of locks are provided, **shared locks** and **exclusive locks**. If a portion of a file already contains a shared lock, a second attempt to place a shared lock on it is permitted, but an attempt to put an exclusive lock on it will fail. If a portion of a file contains an exclusive lock, all attempts to lock any part of that portion will fail until the lock has been released. In order to successfully place a lock, every byte in the region to be locked must be available.

When placing a lock, a process must specify whether it wants to block or not in the event that the lock cannot be placed. If it chooses to block, when the existing lock has been removed, the process is unblocked and the lock is placed. If the process chooses not to block when it cannot place a lock, the system call returns immediately, with the status code telling whether the lock succeeded or not. If it did not, the caller has to decide what to do next (e.g., wait and try again).

Locked regions may overlap. In Fig. 10-26(a) we see that process *A* has placed a shared lock on bytes 4 through 7 of some file. Later, process *B* places a shared lock on bytes 6 through 9, as shown in Fig. 10-26(b). Finally, *C* locks bytes 2 through 11. As long as all these locks are shared, they can co-exist.

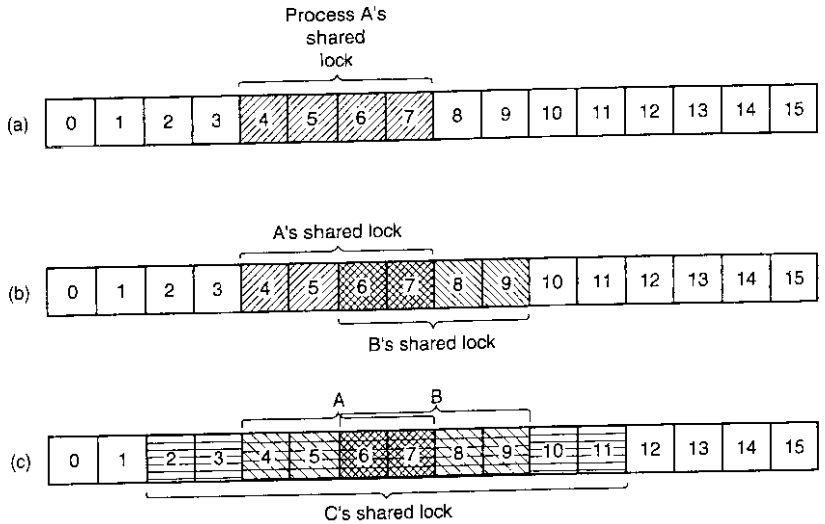

Figure 10-26. (a) A file with one lock. (b) Addition of a second lock. (c) A third lock.

Now consider what happens if a process tries to acquire an exclusive lock to byte 9 of the file of Fig. 10-26(c), with a request to block if the lock fails. Since two previous locks cover this block, the caller will block and will remain blocked until both *B* and *C* release their locks.

10.6.2 File System Calls in Linux

Many system calls relate to files and the file system. First we will look at the system calls that operate on individual files. Later we will examine those that involve directories or the file system as a whole. To create a new file, the creat call can be used. (When Ken Thompson was once asked what he would do differently if he had the chance to reinvent UNIX, he replied that he would spell creat as create this time.) The parameters provide the name of the file and the protection mode. Thus

```
fd = creat("abc", mode);
```

creates a file called *abc* with the protection bits taken from *mode*. These bits determine which users may access the file and how. They will be described later.

The creat call not only creates a new file, but also opens it for writing. To allow subsequent system calls to access the file, a successful creat returns as its result a small nonnegative integer called a **file descriptor**, *fd* in the example above. If a creat is done on an existing file, that file is truncated to length 0 and its contents are discarded. Files can also be created using the open call with appropriate arguments.

Now let us continue looking at the principal file system calls, which are listed in Fig. 10-27. To read or write an existing file, the file must first be opened using open. This call specifies the file name to be opened and how it is to be opened: for reading, writing, or both. Various options can be specified as well. Like creat, the call to open returns a file descriptor that can be used for reading or writing. Afterward, the file can be closed by close, which makes the file descriptor available for reuse on a subsequent creat or open. Both the creat and open calls always return the lowest-numbered file descriptor not currently in use.

When a program starts executing in the standard way, file descriptors 0, 1, and 2 are already opened for standard input, standard output, and standard error, respectively. In this way, a filter, such as the *sort* program, can just read its input from file descriptor 0 and write its output to file descriptor 1, without having to know what files they are. This mechanism works because the shell arranges for these values to refer to the correct (redirected) files before the program is started.

The most heavily used calls are undoubtedly read and write. Each one has three parameters: a file descriptor (telling which open file to read or write), a buffer address (telling where to put the data or get the data from), and a count (telling how many bytes to transfer). That is all there is. It is a very simple design. A typical call is

```
n = read(fd, buffer, nbytes);
```

Although nearly all programs read and write files sequentially, some programs need to be able to access any part of a file at random. Associated with each file is a pointer that indicates the current position in the file. When reading (or

System call	Description
fd = creat(name, mode)	One way to create a new file
fd = open(file, how, ...)	Open a file for reading, writing, or both
s = close(fd)	Close an open file
n = read(fd, buffer, nbytes)	Read data from a file into a buffer
n = write(fd, buffer, nbytes)	Write data from a buffer into a file
position = lseek(fd, offset, whence)	Move the file pointer
s = stat(name, &buf)	Get a file's status information
s = fstat(fd, &buf)	Get a file's status information
s = pipe(&fd[0])	Create a pipe
s = fcntl(fd, cmd, ...)	File locking and other operations

Figure 10-27. Some system calls relating to files. The return code *s* is −1 if an error has occurred; *fd* is a file descriptor, and *position* is a file offset. The parameters should be self explanatory.

writing) sequentially, it normally points to the next byte to be read (written). If the pointer is at, say, 4096, before 1024 bytes are read, it will automatically be moved to 5120 after a successful read system call. The lseek call changes the value of the position pointer, so that subsequent calls to read or write can begin anywhere in the file, or even beyond the end of it. It is called lseek to avoid conflicting with seek, a now-obsolete call that was formerly used on 16-bit computers for seeking.

Lseek has three parameters: the first is the file descriptor for the file; the second is a file position; the third tells whether the file position is relative to the beginning of the file, the current position, or the end of the file. The value returned by lseek is the absolute position in the file after the file pointer is changed. Slightly ironically, lseek is the only file system call that can never cause an actual disk seek because all it does is update the current file position, which is a number in memory.

For each file, Linux keeps track of the file mode (regular, directory, special file), size, time of last modification, and other information. Programs can ask to see this information via the stat system call. The first parameter is the file name. The second is a pointer to a structure where the information requested is to be put. The fields in the structure are shown in Fig. 10-28. The fstat call is the same as stat except that it operates on an open file (whose name may not be known) rather than on a path name.

The pipe system call is used to create shell pipelines. It creates a kind of pseudofile, which buffers the data between the pipeline components, and returns file descriptors for both reading and writing the buffer. In a pipeline such as

```
sort <in | head -30
```


Device the file is on
I-node number (which file on the device)
File mode (includes protection information)
Number of links to the file
Identity of the file's owner
Group the file belongs to
File size (in bytes)
Creation time
Time of last access
Time of last modification

Figure 10-28. The fields returned by the stat system call.

file descriptor 1 (standard output) in the process running *sort* would be set (by the shell) to write to the pipe, and file descriptor 0 (standard input) in the process running *head* would be set to read from the pipe. In this way, *sort* just reads from file descriptor 0 (set to the file *in*) and writes to file descriptor 1 (the pipe) without even being aware that these have been redirected. If they have not been redirected, *sort* will automatically read from the keyboard and write to the screen (the default devices). Similarly, when *head* reads from file descriptor 0, it is reading the data *sort* put into the pipe buffer without even knowing that a pipe is in use. This is a clear example of how a simple concept (redirection) with a simple implementation (file descriptors 0 and 1) can lead to a powerful tool (connecting programs in arbitrary ways without having to modify them at all).

The last system call in Fig. 10-27 is fcntl. It is used to lock and unlock files, apply shared or exclusive locks, and perform a few other file-specific operations.

Now let us look at some system calls that relate more to directories or the file system as a whole, rather than just to one specific file. Some common ones are listed in Fig. 10-29. Directories are created and destroyed using mkdir and rmdir, respectively. A directory can only be removed if it is empty.

As we saw in Fig. 10-24, linking to a file creates a new directory entry that points to an existing file. The link system call creates the link. The parameters specify the original and new names, respectively. Directory entries are removed with unlink. When the last link to a file is removed, the file is automatically deleted. For a file that has never been linked, the first unlink causes it to disappear.

The working directory is changed by the chdir system call. Doing so has the effect of changing the interpretation of relative path names.

The last four calls of Fig. 10-29 are for reading directories. They can be opened, closed, and read, analogous to ordinary files. Each call to readdir returns exactly one directory entry in a fixed format. There is no way for users to write in a directory (in order to maintain the integrity of the file system). Files can be added

System call	Description
s = mkdir(path, mode)	Create a new directory
s = rmdir(path)	Remove a directory
s = link(oldpath, newpath)	Create a link to an existing file
s = unlink(path)	Unlink a file
s = chdir(path)	Change the working directory
dir = opendir(path)	Open a directory for reading
s = closedir(dir)	Close a directory
dirent = readdir(dir)	Read one directory entry
rewinddir(dir)	Rewind a directory so it can be reread

Figure 10-29. Some system calls relating to directories. The return code *s* is −1 if an error has occurred; *dir* identifies a directory stream, and *dirent* is a directory entry. The parameters should be self explanatory.

to a directory using creat or link and removed using unlink. There is also no way to seek to a specific file in a directory, but rewinddir allows an open directory to be read again from the beginning.

10.6.3 Implementation of the Linux File System

In this section we will first look at the abstractions supported by the Virtual File System layer. The VFS hides from higher-level processes and applications the differences among many types of file systems supported by Linux, whether they are residing on local devices or are stored remotely and need to be accessed over the network. Devices and other special files are also accessed through the VFS layer. Next, we will describe the implementation of the first widespread Linux file system, **ext2**, or the second **extended file system**. Afterward, we will discuss the improvements in the ext3 file system. A wide variety of other file systems are also in use. All Linux systems can handle multiple disk partitions, each with a different file system on it.

The Linux Virtual File System

In order to enable applications to interact with different file systems, implemented on different types of local or remote devices, Linux adopts an approach used in other UNIX systems: the Virtual File System (VFS). VFS defines a set of basic file system abstractions and the operations which are allowed on these abstractions. Invocations of the system calls described in the previous section access the VFS data structures, determine the exact file system where the accessed

file belongs, and via function pointers stored in the VFS data structures invoke the corresponding operation in the specified file system.

Fig. 10-30 summarizes the four main file system structures supported by VFS. The **superblock** contains critical information about the layout of the file system. Destruction of the superblock will render the file system unreadable. The **i-nodes** (short for index-nodes, but never called that, although some lazy people drop the hyphen and call them **inodes**) each describe exactly one file. Note that in Linux, directories and devices are also represented as files, thus they will have corresponding i-nodes. Both superblocks and i-nodes have a corresponding structure maintained on the physical disk where the file system resides.

Object	Description	Operation
Superblock	specific filesystem	read_inode, sync_fs
Dentry	directory entry, single component of a path	create, link
i-node	specific file	d_compare, d_delete
File	open file associated with a process	read, write

Figure 10-30. File system abstractions supported by the VFS.

In order to facilitate certain directory operations and traversals of paths, such as */usr/ast/bin*, VFS supports a **dentry** data structure which represents a directory entry. This data structure is created by the file system on the fly. Directory entries are cached in a *dentry_cache*. For instance, the *dentry_cache* would contain entries for */*, */usr*, */usr/ast*, and the like. If multiple processes access the same file through the same hard link (i.e., same path), their file object will point to the same entry in this cache.

Finally, the **file** data structure is an in-memory representation of an open file, and is created in response to the open system call. It supports operations such as read, write, sendfile, lock, and other system calls described in the previous section.

The actual file systems implemented underneath VFS need not use the exact same abstractions and operations internally. They must, however, implement semantically equivalent file system operations as the ones specified with the VFS objects. The elements of the *operations* data structures for each of the four VFS objects are pointers to functions in the underlying file system.

The Linux Ext2 File System

We next describe the most popular on-disk file system used in Linux: **ext2**. The first Linux release used the MINIX file system, and was limited by short filenames and 64-MB file sizes. The MINIX file system was eventually replaced by the first extended file system, **ext**, which permitted both longer file names and

larger file sizes. Due to its performance inefficiencies, ext was replaced by its successor, **ext2**, which is still in widespread use.

An ext2 Linux disk partition contains a file system with the layout illustrated in Fig. 10-31. Block 0 is not used by Linux and often contains code to boot the computer. Following block 0, the disk partition is divided into groups of blocks, without regard to where the disk cylinder boundaries fall. Each group is organized as follows.

The first block is the **superblock**. It contains information about the layout of the file system, including the number of i-nodes, the number of disk blocks, and the start of the list of free disk blocks (typically a few hundred entries). Next comes the group descriptor, which contains information about the location of the bitmaps, the number of free blocks and i-nodes in the group, and the number of directories in the group. This information is important since ext2 attempts to spread directories evenly over the disk.

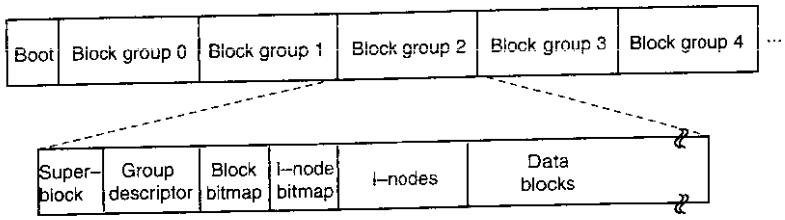

Figure 10-31. Disk layout of the Linux ext2 file system.

Two bitmaps keep track of the free blocks and free i-nodes, respectively, a choice inherited from the MINIX 1 file system (and in contrast to most UNIX file systems, which use a free list). Each map is one block long. With a 1-KB block, this design limits a block group to 8192 blocks and 8192 i-nodes. The former is a real restriction but, in practice, the latter is not.

Following the superblock are the i-nodes themselves. They are numbered from 1 up to some maximum. Each i-node is 128 bytes long and describes exactly one file. An i-node contains accounting information (including all the information returned by stat, which simply takes it from the i-node), as well as enough information to locate all the disk blocks that hold the file's data.

Following the i-nodes are the data blocks. All the files and directories are stored here. If a file or directory consists of more than one block, the blocks need not be contiguous on the disk. In fact, the blocks of a large file are likely to be spread all over the disk.

I-nodes corresponding to directories are dispersed throughout the disk block groups. Ext2 attempts to collocate ordinary files in the same block group as the parent directory, and data files in the same block as the original file i-node, provided that there is sufficient space. This idea was taken from the Berkeley Fast

File System (McKusick et al., 1984). The bitmaps are used to make quick decisions regarding where to allocate new file system data. When new file blocks are allocated, ext2 also *preallocates* a number (eight) of additional blocks for that file, so as to minimize the file fragmentation due to future write operations. This scheme balances the file system load across the entire disk. It also performs well due to its tendencies for collocation and reduced fragmentation.

To access a file, it must first use one of the Linux system calls, such as open, which requires the file's pathname. The pathname is parsed to extract individual directories. If a relative path is specified, the lookup starts from the process' current directory, otherwise it starts from the root directory. In either case, the i-node for the first directory can easily be located: there is a pointer to it in the process descriptor, or, in the case of a root directory, it is typically stored in a predetermined block on disk.

The directory file allows file names up to 255 characters and is illustrated in Fig. 10-32. Each directory consists of some integral number of disk blocks so that directories can be written atomically to the disk. Within a directory, entries for files and directories are in unsorted order, with each entry directly following the one before it. Entries may not span disk blocks, so often there are some number of unused bytes at the end of each disk block.

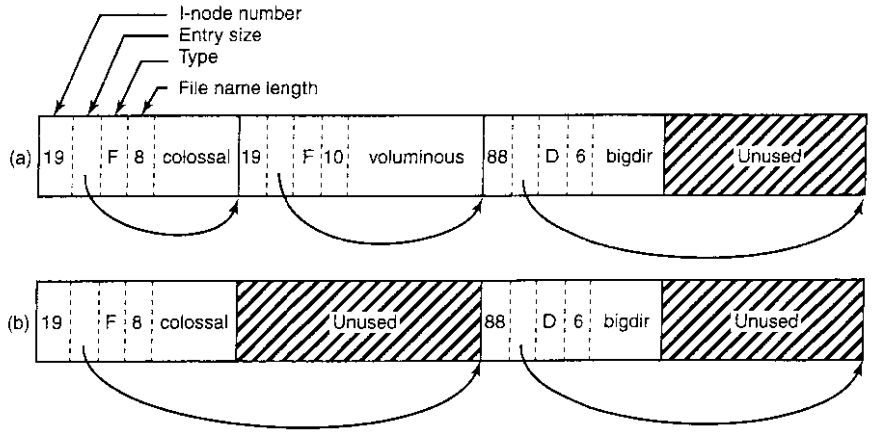

Figure 10-32. (a) A Linux directory with three files. (b) The same directory after the file *voluminous* has been removed.

Each directory entry in Fig. 10-32 consists of four fixed-length fields and one variable-length field. The first field is the i-node number, 19 for the file *colossal*, 42 for the file *voluminous*, and 88 for the directory *bigdir*. Next comes a field rec_len, telling how big the entry is (in bytes), possibly including some padding after the name. This field is needed to find the next entry for the case that the file

name is padded by an unknown length. That is the meaning of the arrow in Fig. 10-32. Then comes the type field: file, directory, and so on. The last fixed field is the length of the actual file name in bytes, 8, 10, and 6 in this example. Finally, comes the file name itself, terminated by a 0 byte and padded out to a 32-bit boundary. Additional padding may follow that.

In Fig. 10-32(b) we see the same directory after the entry for *voluminous* has been removed. All that is done is increase the size of the total entry field for *colossal*, turning the former field for *voluminous* into padding for the first entry. This padding can be used for a subsequent entry, of course.

Since directories are searched linearly, it can take a long time to find an entry at the end of a large directory. Therefore, the system maintains a cache of recently accessed directories. This cache is searched using the name of the file, and if a hit occurs, the costly linear search is avoided. A *dentry* object is entered in the dentry cache for each of the path components, and, through its i-node, the directory is searched for the subsequent path element entry, until the actual file i-node is reached.

For instance, to look up a file specified with an absolute path name, such as */usr/ast/file* the following steps are required. First, the system locates the root directory, which generally uses i-node 2, especially when i-node 1 is reserved for bad block handling. It places an entry in the dentry cache for future lookups of the root directory. Then it looks up the string "usr" in the root directory, to get the i-node number of the */usr* directory, which is also entered in the dentry cache. This i-node is then fetched, and the disk blocks are extracted from it, so the */usr* directory can be read and searched for the string "ast". Once this entry is found, the i-node number for the */usr/ast* directory can be taken from it. Armed with the i-node number of the */usr/ast* directory, this i-node can be read and the directory blocks located. Finally, "file" is looked up and its i-node number found. Thus the use of a relative path name is not only more convenient for the user, but it also saves a substantial amount of work for the system.

If the file is present, the system extracts the i-node number and uses it as an index into the i-node table (on disk) to locate the corresponding i-node and bring it into memory. The i-node is put in the **i-node table**, a kernel data structure that holds all the i-nodes for currently open files and directories. The format of the i-node entries, as a bare minimum, must contain all the fields returned by the stat system call so as to make stat work (see Fig. 10-28). In Fig. 10-33 we show some of the fields included in the i-node structure supported by the Linux file system layer. The actual i-node structure contains many more fields, since the same structure is also used to represent directories, devices, and other special files. The i-node structure also contains fields reserved for future use. History has shown that unused bits do not remain that way for long.

Let us now see how the system reads a file. Remember that a typical call to the library procedure for invoking the read system call looks like this:

```
n = read(fd, buffer, nbytes);
```


Field	Bytes	Description
Mode	2	File type, protection bits, setuid, setgid bits
Nlinks	2	Number of directory entries pointing to this i-node
Uid	2	UID of the file owner
Gid	2	GID of the file owner
Size	4	File size in bytes
Addr	60	Address of first 12 disk blocks, then 3 indirect blocks
Gen	1	Generation number (incremented every time i-node is reused)
Atime	4	Time the file was last accessed
Mtime	4	Time the file was last modified
Ctime	4	Time the i-node was last changed (except the other times)

Figure 10-33. Some fields in the i-node structure in Linux

When the kernel gets control, all it has to start with are these three parameters and the information in its internal tables relating to the user. One of the items in the internal tables is the file descriptor array. It is indexed by a file descriptor and contains one entry for each open file (up to the maximum number, usually defaults to 32).

The idea is to start with this file descriptor and end up with the corresponding i-node. Let us consider one possible design: just put a pointer to the i-node in the file descriptor table. Although simple, unfortunately this method does not work. The problem is as follows. Associated with every file descriptor is a file position that tells at which byte the next read (or write) will start. Where should it go? One possibility is to put it in the i-node table. However, this approach fails if two or more unrelated processes happen to open the same file at the same time because each one has its own file position.

A second possibility is to put the file position in the file descriptor table. In that way, every process that opens a file gets its own private file position. Unfortunately this scheme fails too, but the reasoning is more subtle and has to do with the nature of file sharing in Linux. Consider a shell script, *s*, consisting of two commands, *p1* and *p2*, to be run in order. If the shell script is called by the command line

```
s >x
```

it is expected that *p1* will write its output to *x*, and then *p2* will write its output to *x* also, starting at the place where *p1* stopped.

When the shell forks off *p1*, *x* is initially empty, so *p1* just starts writing at file position 0. However, when *p1* finishes, some mechanism is needed to make sure that the initial file position that *p2* sees is not 0 (which it would be if the file position were kept in the file descriptor table), but the value *p1* ended with.

The way this is achieved is shown in Fig. 10-34. The trick is to introduce a new table, the **open file description table**, between the file descriptor table and the i-node table, and put the file position (and read/write bit) there. In this figure, the parent is the shell and the child is first *p1* and later *p2*. When the shell forks off *p1*, its user structure (including the file descriptor table) is an exact copy of the shell's, so both of them point to the same open file description table entry. When *p1* finishes, the shell's file descriptor is still pointing to the open file description containing *p1*'s file position. When the shell now forks off *p2*, the new child automatically inherits the file position, without either it or the shell even having to know what that position is.

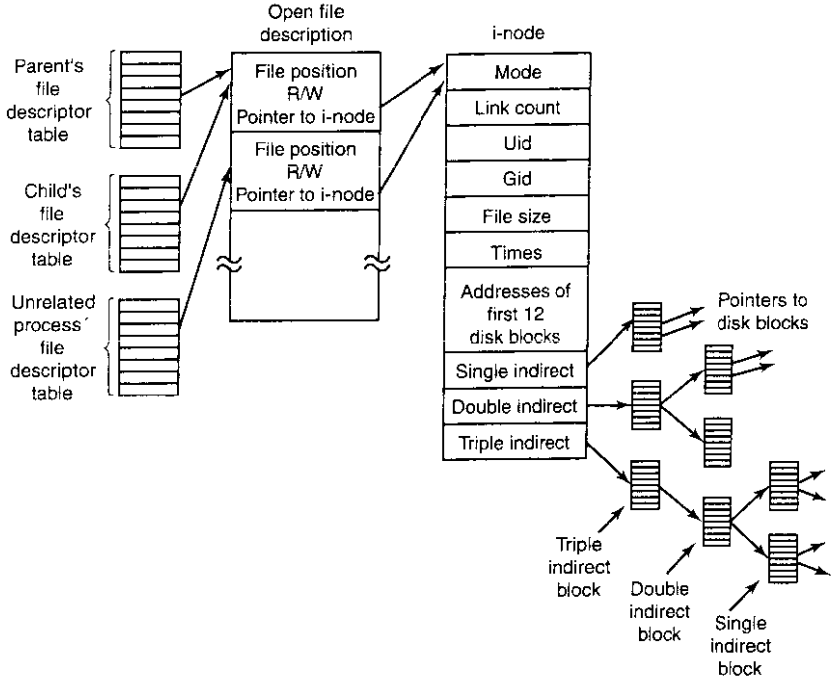

Figure 10-34. The relation between the file descriptor table, the open file description table, and the i-node table.

However, if an unrelated process opens the file, it gets its own open file description entry, with its own file position, which is precisely what is needed. Thus the whole point of the open file description table is to allow a parent and child to share a file position, but to provide unrelated processes with their own values.

Getting back to the problem of doing the read, we have now shown how the file position and i-node are located. The i-node contains the disk addresses of the first 12 blocks of the file. If the file position falls in the first 12 blocks, the block is read and the data are copied to the user. For files longer than 12 blocks, a field in the i-node contains the disk address of a **single indirect block**, as shown in Fig. 10-34. This block contains the disk addresses of more disk blocks. For example, if a block is 1 KB and a disk address is 4 bytes, the single indirect block can hold 256 disk addresses. Thus this scheme works for files of up to 268 KB in total.

Beyond that, a **double indirect block** is used. It contains the addresses of 256 single indirect blocks, each of which holds the addresses of 256 data blocks. This mechanism is sufficient to handle files up to $10 + 2^{16}$ blocks (67,119,104 bytes). If even this is not enough, the i-node has space for a **triple indirect block**. Its pointers point to many double indirect blocks. This addressing scheme can handle file sizes of 2^{24} 1 KB blocks (16 GB). For 8-KB block sizes, the addressing scheme can support file sizes up to 64 TB.

The Linux Ext3 File System

In order to prevent all data loss after system crashes and power failures, the ext2 file system would have to write out each data block to disk as soon as it was created. The latency incurred during the required disk head seek operation would be so high that the performance would be intolerable. Therefore, writes are delayed, and changes may not be committed to disk for up to 30 sec, which is a very long time interval in the context of modern computer hardware.

To improve the robustness of the file system, Linux relies on **journaling file systems**. **Ext3**, a follow-on of the ext2 file system, is an example of a journaling file system.

The basic idea behind this type of file system is to maintain a *journal*, which describes all file system operations in sequential order. By sequentially writing out changes to the file system data or metadata (i-nodes, superblock, etc.), the operations do not suffer from the overheads of disk head movement during random disk accesses. Eventually, the changes will be written out, committed, to the appropriate disk location, and the corresponding journal entries can be discarded. If a system crash or power failure occurs before the changes are committed, during restart the system will detect that the file system was not unmounted properly, traverse the journal, and apply the file system changes described in the journal log.

Ext3 is designed to be highly compatible with ext2, and in fact, all core data structures and disk layout are the same in both systems. Furthermore, a file system which has been unmounted as an ext2 system can be subsequently mounted as an ext3 system and offer the journaling capability.

The journal is a file managed as a circular buffer. The journal may be stored on the same or a separate device from the main file system. Since the journal operations are not "journaled" themselves, these are not handled by the same ext3

file system. Instead, a separate **JBD (Journaling Block Device)** is used to perform the journal read/write operations.

JBD supports three main data structures: *log record*, *atomic operation handle*, and *transaction*. A log record describes a low-level file system operation, typically resulting in changes within a block. Since a system call such as write includes changes at multiple places—i-nodes, existing file blocks, new file blocks, list of free blocks, etc.—related log records are grouped in atomic operations. Ext3 notifies JBD of the start and end of a system call processing, so that JBD can ensure that either all log records in an atomic operation are applied, or none of them. Finally, primarily for efficiency reasons, JBD treats collections of atomic operations as transactions. Log records are stored consecutively within a transaction. JBD will allow portions of the journal file to be discarded only after all log records belonging to a transaction are safely committed to disk.

Since writing out a log entry for each disk change may be costly, ext3 may be configured to keep a journal of all disk changes, or only of changes related to the file system metadata (the i-nodes, superblocks, bitmaps, etc.). Journaling only metadata gives less system overhead and results in better performance but does not make any guarantees against corruption of file data. Several other journaling file systems maintain logs of only metadata operations (e.g., SGI's XFS).

The /proc File System

Another Linux file system is the **/proc** (process) file system, an idea originally devised in the 8th edition of UNIX from Bell Labs and later copied in 4.4BSD and System V. However, Linux extends the idea in several ways. The basic concept is that for every process in the system, a directory is created in */proc*. The name of the directory is the process PID expressed as a decimal number. For example, */proc/619* is the directory corresponding to the process with PID 619. In this directory are files that appear to contain information about the process, such as its command line, environment strings, and signal masks. In fact, these files do not exist on the disk. When they are read, the system retrieves the information from the actual process as needed and returns it in a standard format.

Many of the Linux extensions relate to other files and directories located in */proc*. They contain a wide variety of information about the CPU, disk partitions, devices, interrupt vectors, kernel counters, file systems, loaded modules, and much more. Unprivileged user programs may read much of this information to learn about system behavior in a safe way. Some of these files may be written to in order to change system parameters.

10.6.4 NFS: The Network File System

Networking has played a major role in Linux, and UNIX in general, right from the beginning (the first UNIX network was built to move new kernels from the PDP-11/70 to the Interdata 8/32 during the port to the latter). In this section we will examine Sun Microsystem's **NFS (Network File System)**, which is used on all modern Linux systems to join the file systems on separate computers into one logical whole. Currently, the dominant NSF implementation is version 3, introduced in 1994. NSFv4 was introduced in 2000 and provides several enhancements over the previous NFS architecture. Three aspects of NFS are of interest: the architecture, the protocol, and the implementation. We will now examine these in turn, first in the context of the simpler NFS version 3, then we will briefly discuss the enhancements included in v4.

NFS Architecture

The basic idea behind NFS is to allow an arbitrary collection of clients and servers to share a common file system. In many cases, all the clients and servers are on the same LAN, but this is not required. It is also possible to run NFS over a wide area network if the server is far from the client. For simplicity we will speak of clients and servers as though they were on distinct machines, but in fact, NFS allows every machine to be both a client and a server at the same time.

Each NFS server exports one or more of its directories for access by remote clients. When a directory is made available, so are all of its subdirectories, so in fact, entire directory trees are normally exported as a unit. The list of directories a server exports is maintained in a file, often */etc/exports*, so these directories can be exported automatically whenever the server is booted. Clients access exported directories by mounting them. When a client mounts a (remote) directory, it becomes part of its directory hierarchy, as shown in Fig. 10-35.

In this example, client 1 has mounted the *bin* directory of server 1 on its own *bin* directory, so it can now refer to the shell as */bin/sh* and get the shell on server 1. Diskless workstations often have only a skeleton file system (in RAM) and get all their files from remote servers like this. Similarly, client 1 has mounted server 2's directory */projects* on its directory */usr/ast/work* so it can now access file *a* as */usr/ast/work/proj1/a*. Finally, client 2 has also mounted the *projects* directory and can also access file *a*, only as */mnt/proj1/a*. As seen here, the same file can have different names on different clients due to its being mounted in a different place in the respective trees. The mount point is entirely local to the clients; the server does not know where it is mounted on any of its clients.

NFS Protocols

Since one of the goals of NFS is to support a heterogeneous system, with clients and servers possibly running different operating systems on different hardware, it is essential that the interface between the clients and servers be well defined. Only then is it possible for anyone to be able to write a new client implementation and expect it to work correctly with existing servers, and vice versa.

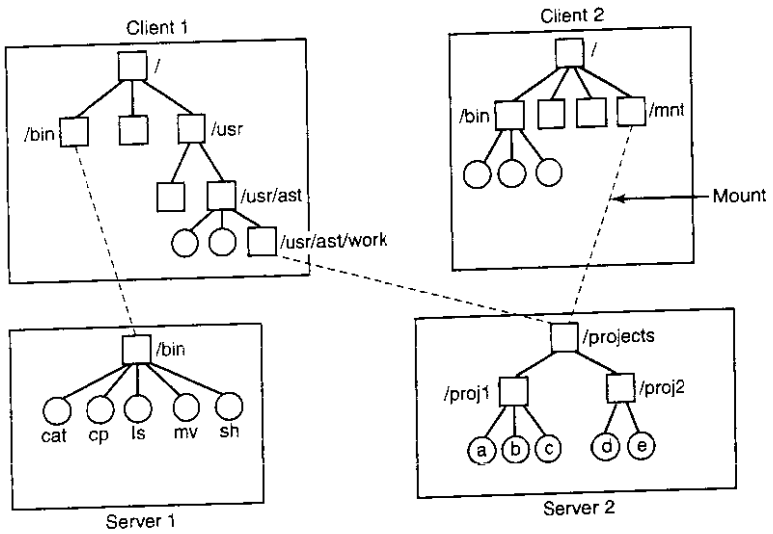

Figure 10-35. Examples of remote mounted file systems. Directories are shown as squares and files are shown as circles.

NFS accomplishes this goal by defining two client-server protocols. A **protocol** is a set of requests sent by clients to servers, along with the corresponding replies sent by the servers back to the clients.

The first NFS protocol handles mounting. A client can send a path name to a server and request permission to mount that directory somewhere in its directory hierarchy. The place where it is to be mounted is not contained in the message, as the server does not care where it is to be mounted. If the path name is legal and the directory specified has been exported, the server returns a **file handle** to the client. The file handle contains fields uniquely identifying the file system type, the disk, the i-node number of the directory, and security information. Subsequent calls to read and write files in the mounted directory or any of its subdirectories use the file handle.

When Linux boots, it runs the */etc/rc* shell script before going multiuser. Commands to mount remote file systems can be placed in this script, thus automatically mounting the necessary remote file systems before allowing any logins. Alternatively, most versions of Linux also support **automounting**. This feature allows a set of remote directories to be associated with a local directory. None of these remote directories are mounted (or their servers even contacted) when the client is booted. Instead, the first time a remote file is opened, the operating system sends a message to each of the servers. The first one to reply wins, and its directory is mounted.

Automounting has two principal advantages over static mounting via the */etc/rc* file. First, if one of the NFS servers named in */etc/rc* happens to be down, it is impossible to bring the client up, at least not without some difficulty, delay, and quite a few error messages. If the user does not even need that server at the moment, all that work is wasted. Second, by allowing the client to try a set of servers in parallel, a degree of fault tolerance can be achieved (because only one of them needs to be up), and the performance can be improved (by choosing the first one to reply—presumably the least heavily loaded).

On the other hand, it is tacitly assumed that all the file systems specified as alternatives for the automount are identical. Since NFS provides no support for file or directory replication, it is up to the user to arrange for all the file systems to be the same. Consequently, automounting is most often used for read-only file systems containing system binaries and other files that rarely change.

The second NFS protocol is for directory and file access. Clients can send messages to servers to manipulate directories and read and write files. They can also access file attributes, such as file mode, size, and time of last modification. Most Linux system calls are supported by NFS, with the perhaps surprising exceptions of open and close.

The omission of open and close is not an accident. It is fully intentional. It is not necessary to open a file before reading it, nor to close it when done. Instead, to read a file, a client sends the server a lookup message containing the file name, with a request to look it up and return a file handle, which is a structure that identifies the file (i.e., contains a file system identifier and i-node number, among other data). Unlike an open call, this lookup operation does not copy any information into internal system tables. The read call contains the file handle of the file to read, the offset in the file to begin reading, and the number of bytes desired. Each such message is self-contained. The advantage of this scheme is that the server does not have to remember anything about open connections in between calls to it. Thus if a server crashes and then recovers, no information about open files is lost, because there is none. A server like this that does not maintain state information about open files is said to be **stateless**.

Unfortunately, the NFS method makes it difficult to achieve the exact Linux file semantics. For example, in Linux a file can be opened and locked so that other processes cannot access it. When the file is closed, the locks are released. In a stateless server such as NFS, locks cannot be associated with open files, because the server does not know which files are open. NFS therefore needs a separate, additional mechanism to handle locking.

NFS uses the standard UNIX protection mechanism, with the *rwx* bits for the owner, group, and others (mentioned in Chap. 1 and discussed in detail below). Originally, each request message simply contained the user and group IDs of the caller, which the NFS server used to validate the access. In effect, it trusted the clients not to cheat. Several years' experience abundantly demonstrated that such an assumption was—how shall we put it?—rather naive. Currently, public key

cryptography can be used to establish a secure key for validating the client and server on each request and reply. When this option is used, a malicious client cannot impersonate another client because it does not know that client's secret key.

NFS Implementation

Although the implementation of the client and server code is independent of the NFS protocols, most Linux systems use a three-layer implementation similar to that of Fig. 10-36. The top layer is the system call layer. This handles calls like open, read, and close. After parsing the call and checking the parameters, it invokes the second layer, the Virtual File System (VFS) layer.

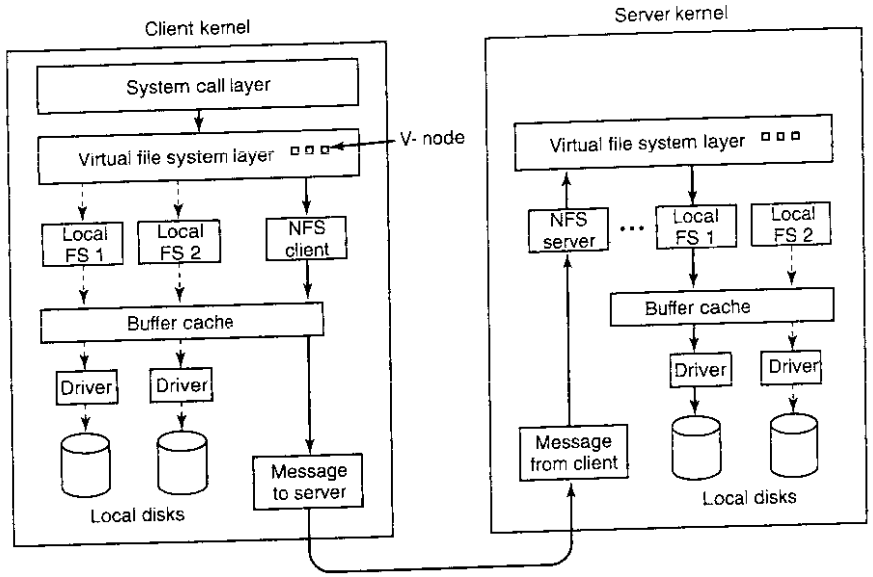

Figure 10-36. The NFS layer structure

The task of the VFS layer is to maintain a table with one entry for each open file. The VFS layer has an entry, a **virtual i-node**, or **v-node**, for every open file. V-nodes are used to tell whether the file is local or remote. For remote files, enough information is provided to be able to access them. For local files, the file system and i-node are recorded because modern Linux systems can support multiple file systems (e.g., ext2fs, /proc, FAT, etc.). Although VFS was invented to support NFS, most modern Linux systems now support it as an integral part of the operating system, even if NFS is not used.

To see how v-nodes are used, let us trace a sequence of mount, open, and read system calls. To mount a remote file system, the system administrator (or

/etc/rc) calls the *mount* program specifying the remote directory, the local directory on which it is to be mounted, and other information. The *mount* program parses the name of the remote directory to be mounted and discovers the name of the NFS server on which the remote directory is located. It then contacts that machine, asking for a file handle for the remote directory. If the directory exists and is available for remote mounting, the server returns a file handle for the directory. Finally, it makes a mount system call, passing the handle to the kernel.

The kernel then constructs a v-node for the remote directory and asks the NFS client code in Fig. 10-36 to create an **r-node (remote i-node)** in its internal tables to hold the file handle. The v-node points to the r-node. Each v-node in the VFS layer will ultimately contain either a pointer to an r-node in the NFS client code, or a pointer to an i-node in one of the local file systems (shown as dashed lines in Fig. 10-36). Thus from the v-node it is possible to see if a file or directory is local or remote. If it is local, the correct file system and i-node can be located. If it is remote, the remote host and file handle can be located.

When a remote file is opened on the client, at some point during the parsing of the path name, the kernel hits the directory on which the remote file system is mounted. It sees that this directory is remote and in the directory's v-node finds the pointer to the r-node. It then asks the NFS client code to open the file. The NFS client code looks up the remaining portion of the path name on the remote server associated with the mounted directory and gets back a file handle for it. It makes an r-node for the remote file in its tables and reports back to the VFS layer, which puts in its tables a v-node for the file that points to the r-node. Again here we see that every open file or directory has a v-node that points to either an r-node or an i-node.

The caller is given a file descriptor for the remote file. This file descriptor is mapped onto the v-node by tables in the VFS layer. Note that no table entries are made on the server side. Although the server is prepared to provide file handles upon request, it does not keep track of which files happen to have file handles outstanding and which do not. When a file handle is sent to it for file access, it checks the handle, and if it is valid, uses it. Validation can include verifying an authentication key contained in the RPC headers, if security is enabled.

When the file descriptor is used in a subsequent system call, for example, read, the VFS layer locates the corresponding v-node, and from that determines whether it is local or remote and also which i-node or r-node describes it. It then sends a message to the server containing the handle, the file offset (which is maintained on the client side, not the server side), and the byte count. For efficiency reasons, transfers between client and server are done in large chunks, normally 8192 bytes, even if fewer bytes are requested.

When the request message arrives at the server, it is passed to the VFS layer there, which determines which local file system holds the requested file. The VFS layer then makes a call to that local file system to read and return the bytes. These data are then passed back to the client. After the client's VFS layer has gotten the

8-KB chunk it asked for, it automatically issues a request for the next chunk, so it will have it should it be needed shortly. This feature, known as **read ahead**, improves performance considerably.

For writes an analogous path is followed from client to server. Also, transfers are done in 8-KB chunks here too. If a write system call supplies fewer than 8 KB bytes of data, the data are just accumulated locally. Only when the entire 8-KB chunk is full is it sent to the server. However, when a file is closed, all of its data are sent to the server immediately.

Another technique used to improve performance is caching, as in ordinary UNIX. Servers cache data to avoid disk accesses, but this is invisible to the clients. Clients maintain two caches, one for file attributes (i-nodes) and one for file data. When either an i-node or a file block is needed, a check is made to see if it can be satisfied out of the cache. If so, network traffic can be avoided.

While client caching helps performance enormously, it also introduces some nasty problems. Suppose that two clients are both caching the same file block and that one of them modifies it. When the other one reads the block, it gets the old (stale) value. The cache is not coherent.

Given the potential severity of this problem, the NFS implementation does several things to mitigate it. For one, associated with each cache block is a timer. When the timer expires, the entry is discarded. Normally, the timer is 3 sec for data blocks and 30 sec for directory blocks. Doing this reduces the risk somewhat. In addition, whenever a cached file is opened, a message is sent to the server to find out when the file was last modified. If the last modification occurred after the local copy was cached, the cache copy is discarded and the new copy fetched from the server. Finally, once every 30 sec a cache timer expires, and all the dirty (i.e., modified) blocks in the cache are sent to the server. While not perfect, these patches make the system highly usable in most practical circumstances.

NFS Version 4

Version 4 of the Network File System was designed to simplify certain operations from its predecessor. In contrast to NSFv3, which is described above, NFSv4 is a **stateful** file system. This permits open operations to be invoked on remote files, since the remote NFS server will maintain all file-system-related structures, including the file pointer. Read operations then need not include absolute read ranges, but can be incrementally applied from the previous file pointer position. This results in shorter messages, and also in the ability to bundle multiple NFSv3 operations in one network transaction.

The stateful nature of NFSv4 makes it easy to integrate the variety of NFSv3 protocols described earlier in this section into one coherent protocol. There is no need to support separate protocols for mounting, caching, locking, or secure operations. NFSv4 also works better with both Linux (and UNIX in general) and Windows file system semantics.

10.7 SECURITY IN LINUX

Linux, as a clone of MINIX and UNIX, has been a multiuser system almost from the beginning. This history means that security and control of information was built in very early on. In the following sections, we will look at some of the security aspects of Linux.

10.7.1 Fundamental Concepts

The user community for a Linux system consists of some number of registered users, each of whom has a unique **UID (User ID)**. A UID is an integer between 0 and 65,535. Files (but also processes and other resources) are marked with the UID of their owner. By default, the owner of a file is the person who created the file, although there is a way to change ownership.

Users can be organized into groups, which are also numbered with 16-bit integers called **GIDs (Group IDs)**. Assigning users to groups is done manually (by the system administrator) and consists of making entries in a system database telling which user is in which group. A user could be in one or more groups at the same time. For simplicity, we will not discuss this feature further.

The basic security mechanism in Linux is simple. Each process carries the UID and GID of its owner. When a file is created, it gets the UID and GID of the creating process. The file also gets a set of permissions determined by the creating process. These permissions specify what access the owner, the other members of the owner's group, and the rest of the users have to the file. For each of these three categories, potential accesses are read, write, and execute, designated by the letters *r*, *w*, and *x*, respectively. The ability to execute a file makes sense only if that file is an executable binary program, of course. An attempt to execute a file that has execute permission but which is not executable (i.e., does not start with a valid header) will fail with an error. Since there are three categories of users and 3 bits per category, 9 bits are sufficient to represent the access rights. Some examples of these 9-bit numbers and their meanings are given in Fig. 10-37.

Binary	Symbolic	Allowed file accesses
111000000	rwx------	Owner can read, write, and execute
111111000	rwxrwx---	Owner and group can read, write, and execute
110100000	rw-r-----	Owner can read and write; group can read
110100100	rw-r--r--	Owner can read and write; all others can read
111101101	rwxr-xr-x	Owner can do everything, rest can read and execute
000000000	---------	Nobody has any access
000000111	------rwx	Only outsiders have access (strange, but legal)

Figure 10-37. Some example file protection modes.

The first two entries in Fig. 10-37 are clear, allowing the owner and the owner's group full access, respectively. The next one allows the owner's group to read the file but not to change it, and prevents outsiders from any access. The fourth entry is common for a data file the owner wants to make public. Similarly, the fifth entry is the usual one for a publicly available program. The sixth entry denies all access to all users. This mode is sometimes used for dummy files used for mutual exclusion because an attempt to create such a file will fail if one already exists. Thus if multiple processes simultaneously attempt to create such a file as a lock, only one of them will succeed. The last example is strange indeed, since it gives the rest of the world more access than the owner. However, its existence follows from the protection rules. Fortunately, there is a way for the owner to subsequently change the protection mode, even without having any access to the file itself.

The user with UID 0 is special and is called the **superuser** (or **root**). The superuser has the power to read and write all files in the system, no matter who owns them and no matter how they are protected. Processes with UID 0 also have the ability to make a small number of protected system calls denied to ordinary users. Normally, only the system administrator knows the superuser's password, although many undergraduates consider it a great sport to try to look for security flaws in the system so they can log in as the superuser without knowing the password. Management tends to frown on such activity.

Directories are files and have the same protection modes that ordinary files do except that the *x* bits refer to search permission instead of execute permission. Thus a directory with mode *rwxr–xr–x* allows its owner to read, modify, and search the directory, but allows others only to read and search it, but not add or remove files from it.

Special files corresponding to the I/O devices have the same protection bits as regular files. This mechanism can be used to limit access to I/O devices. For example, the printer special file, */dev/lp*, could be owned by the root or by a special user, daemon, and have mode *rw–––––––* to keep everyone else from directly accessing the printer. After all, if everyone could just print at will, chaos would result.

Of course, having */dev/lp* owned by, say, daemon with protection mode *rw–––––––* means that nobody else can use the printer. While this would save many innocent trees from an early death, sometimes users do have a legitimate need to print something. In fact, there is a more general problem of allowing controlled access to all I/O devices and other system resources.

This problem was solved by adding a new protection bit, the **SETUID bit** to the 9 protection bits discussed above. When a program with the SETUID bit on is executed, the **effective UID** for that process becomes the UID of the executable file's owner instead of the UID of the user who invoked it. When a process attempts to open a file, it is the effective UID that is checked, not the underlying real UID. By making the program that accesses the printer be owned by daemon

but with the SETUID bit on, any user could execute it, and have the power of daemon (e.g., access to */dev/lp*) but only to run that program (which might queue print jobs for printing in an orderly fashion).

Many sensitive Linux programs are owned by the root but with the SETUID bit on. For example, the program that allows users to change their passwords, *passwd*, needs to write in the password file. Making the password file publicly writable would not be a good idea. Instead, there is a program that is owned by the root and which has the SETUID bit on. Although the program has complete access to the password file, it will only change the caller's password and not permit any other access to the password file.

In addition to the SETUID bit there is also a SETGID bit that works analogously, temporarily giving the user the effective GID of the program. In practice, this bit is rarely used, however.

10.7.2 Security System Calls in Linux

There are only a small number of system calls relating to security. The most important ones are listed in Fig. 10-38. The most heavily used security system call is chmod. It is used to change the protection mode. For example,

```
s = chmod("/usr/ast/newgame", 0755);
```

sets *newgame* to *rwxr–xr–x* so that everyone can run it (note that 0755 is an octal constant, which is convenient, since the protection bits come in groups of 3 bits). Only the owner of a file and the superuser can change its protection bits.

System call	Description
s = chmod(path, mode)	Change a file's protection mode
s = access(path, mode)	Check access using the real UID and GID
uid = getuid( )	Get the real UID
uid = geteuid( )	Get the effective UID
gid = getgid( )	Get the real GID
gid = getegid( )	Get the effective GID
s = chown(path, owner, group)	Change owner and group
s = setuid(uid)	Set the UID
s = setgid(gid)	Set the GID

Figure 10-38. Some system calls relating to security. The return code *s* is –1 if an error has occurred; *uid* and *gid* are the UID and GID, respectively. The parameters should be self explanatory.

The access call tests to see if a particular access would be allowed using the real UID and GID. This system call is needed to avoid security breaches in

programs that are SETUID and owned by the root. Such a program can do anything, and it is sometimes needed for the program to figure out if the user is allowed to perform a certain access. The program cannot just try it, because the access will always succeed. With the access call the program can find out if the access is allowed by the real UID and real GID.

The next four system calls return the real and effective UIDs and GIDs. The last three are only allowed for the superuser. They change a file's owner, and a process' UID and GID.

10.7.3 Implementation of Security in Linux

When a user logs in, the login program, *login* (which is SETUID root) asks for a login name and a password. It hashes the password and then looks in the password file, */etc/passwd*, to see if the hash matches the one there (networked systems work slightly differently). The reason for using hashes is to prevent the password from being stored in unencrypted form anywhere in the system. If the password is correct, the login program looks in */etc/passwd* to see the name of the user's preferred shell, possibly *bash*, but possibly some other shell such as *csh* or *ksh*. The login program then uses setuid and setgid to give itself the user's UID and GID (remember, it started out as SETUID root). Then it opens the keyboard for standard input (file descriptor 0), the screen for standard output (file descriptor 1), and the screen for standard error (file descriptor 2). Finally, it executes the preferred shell, thus terminating itself.

At this point the preferred shell is running with the correct UID and GID and standard input, output, and error all set to their default devices. All processes that it forks off (i.e., commands typed by the user) automatically inherit the shell's UID and GID, so they also will have the correct owner and group. All files they create also get these values.

When any process attempts to open a file, the system first checks the protection bits in the file's i-node against the caller's effective UID and effective GID to see if the access is permitted. If so, the file is opened and a file descriptor returned. If not, the file is not opened and −1 is returned. No checks are made on subsequent read or write calls. As a consequence, if the protection mode changes after a file is already open, the new mode will not affect processes that already have the file open.

The Linux security model and its implementation are essentially the same as in most other traditional UNIX systems.

10.8 SUMMARY

Linux began its life as an open-source, full production UNIX clone, and is now used on machines ranging from notebook computers to supercomputers. Three main interfaces to it exist: the shell, the C library, and the system calls themselves. In addition, a graphical user interface is often used to simplify user interaction with the system. The shell allows users to type commands for execution. These may be simple commands, pipelines, or more complex structures. Input and output may be redirected. The C library contains the system calls and also many enhanced calls, such as *printf* for writing formatted output to files. The actual system call interface is architecture dependent, and on x86 platforms consists of approximately 250 calls, each of which does what is needed and no more.

The key concepts in Linux include the process, the memory model, I/O, and the file system. Processes may fork off subprocesses, leading to a tree of processes. Process management in Linux is different compared to other UNIX systems in that Linux views each execution entity—a single-threaded process, or each thread within a multithreaded process or the kernel—as a distinguishable task. A process, or a single task in general, is then represented via two key components, the task structure and the additional information describing the user address space. The former is always in memory, but the latter data can be paged in and out of memory. Process creation is done by duplicating the process task structure, and then setting the memory image information to point to the parents' memory image. Actual copies of the memory image pages are created only if sharing is not allowed and a memory modification is required. This mechanism is called copy on write. Scheduling is done using a priority-based algorithm that favors interactive processes.

The memory model consists of three segments per process: text, data, and stack. Memory management is done by paging. An in-memory map keeps track of the state of each page, and the page daemon uses a modified dual-hand clock algorithm to keep enough free pages around.

I/O devices are accessed using special files, each of which has a major device number and a minor device number. Block device I/O uses a the main memory to cache disk blocks and reduce the number of disk accesses. Character I/O can be done in raw mode, or character streams can be modified via line disciplines. Networking devices are treated somewhat differently, by associating entire network protocol modules to process the network packets stream to and from the user process.

The file system is hierarchical with files and directories. All disks are mounted into a single directory tree starting at a unique root. Individual files can be linked into a directory from elsewhere in the file system. To use a file, it must be first opened, which yields a file descriptor for use in reading and writing the file. Internally, the file system uses three main tables: the file descriptor table, the open file description table, and the i-node table. The i-node table is the most important of these, containing all the administrative information about a file and the location of its blocks. Directories and devices are also represented as files, along with other special files.

Protection is based on controlling read, write, and execute access for the owner, group, and others. For directories, the execute bit means search permission.

PROBLEMS

1. A directory contains the following files:

aardvark	feret	koala	porpoise	unicorn
bonefish	grunion	llama	quacker	vicuna
capybara	hyena	marmot	rabbit	weasel
dingo	ibex	nuthatch	seahorse	yak
emu	jellyfish	ostrich	tuna	zebu

Which files will be listed by the command

```
ls [abc]*e*?
```

2. What does the following Linux shell pipeline do?

```
grep nd xyz | wc -l
```

3. Write a Linux pipeline that prints the eighth line of file z on standard output.

4. Why does Linux distinguish between standard output and standard error, when both default to the terminal?

5. A user at a terminal types the following commands:

```
a | b | c &
d | e | f &
```

After the shell has processed them, how many new processes are running?

6. When the Linux shell starts up a process, it puts copies of its environment variables, such as *HOME*, on the process' stack, so the process can find out what its home directory is. If this process should later fork, will the child automatically get these variables too?

7. About how long does it take a traditional UNIX system to fork off a child process under the following conditions: text size = 100 KB, data size = 20 KB, stack size = 10 KB, task structure = 1 KB, user structure = 5 KB. The kernel trap and return takes 1 msec, and the machine can copy one 32-bit word every 50 nsec. Text segments are shared, but data and stack segments are not.

8. As multi-megabyte programs became more common, the time spent executing the fork system call and copying the data and stack segments of the calling process grew proportionally. When fork is executed in Linux, the parent's address space is not copied, as traditional fork semantics would dictate. How does Linux prevent the child from doing something that would completely change the fork semantics?

9. Does it make sense to take away a process' memory when it enters zombie state? Why or why not?

10. Why do you think the designers of Linux made it impossible for a process to send a signal to another process that is not in its process group?

11. A system call is usually implemented using a software interrupt (trap) instruction. Could an ordinary procedure call be used as well on the Pentium hardware? If so, under what conditions and how? If not, why not?

12. In general, do you think daemons have higher priority or lower priority than interactive processes? Why?

13. When a new process is forked off, it must be assigned a unique integer as its PID. Is it sufficient to have a counter in the kernel that is incremented on each process creation, with the counter used as the new PID? Discuss your answer.

14. In every process' entry in the task structure, the PID of the parent is stored. Why?

15. What combination of the *sharing_flags* bits used by the Linux clone command corresponds to a conventional UNIX fork call? To creating a conventional UNIX thread?

16. The Linux scheduler went through a major overhaul between the 2.4 and 2.6 kernel. The current scheduler can make scheduling decisions in O(1) time. Explain why is this so?

17. When booting Linux (or most other operating systems for that matter), the bootstrap loader in sector 0 of the disk first loads a boot program which then loads the operating system. Why is this extra step necessary? Surely it would be simpler to have the bootstrap loader in sector 0 just load the operating system directly.

18. A certain editor has 100 KB of program text, 30 KB of initialized data, and 50 KB of BSS. The initial stack is 10 KB. Suppose that three copies of this editor are started simultaneously. How much physical memory is needed (a) if shared text is used, and (b) if it is not?

19. Why are open file descriptor tables necessary in Linux?

20. In Linux, the data and stack segments are paged and swapped to a scratch copy kept on a special paging disk or partition, but the text segment uses the executable binary file instead. Why?

21. Describe a way to use mmap and signals to construct an interprocess communication mechanism.

22. A file is mapped in using the following mmap system call:

```
mmap(65536, 32768, READ, FLAGS, fd, 0)
```

Pages are 8 KB. Which byte in the file is accessed by reading a byte at memory address 72,000?

23. After the system call of the previous problem has been executed, the call

```
munmap(65536, 8192)
```

is carried out. Does it succeed? If so, which bytes of the file remain mapped? If not, why does it fail?

24. Can a page fault ever lead to the faulting process being terminated? If so, give an example. If not, why not?

25. Is it possible that with the buddy system of memory management it ever occurs that two adjacent blocks of free memory of the same size co-exist without being merged into one block? If so, explain how. If not, show that it is impossible.

26. It is stated in the text that a paging partition will perform better than a paging file. Why is this so?

27. Give two examples of the advantages of relative path names over absolute ones.

28. The following locking calls are made by a collection of processes. For each call, tell what happens. If a process fails to get a lock, it blocks.

    (a) *A* wants a shared lock on bytes 0 through 10.
    (b) *B* wants an exclusive lock on bytes 20 through 30.
    (c) *C* wants a shared lock on bytes 8 through 40.
    (d) *A* wants a shared lock on bytes 25 through 35.
    (e) *B* wants an exclusive lock on byte 8.

29. Consider the locked file of Fig. 10-26(c). Suppose that a process tries to lock bytes 10 and 11 and blocks. Then, before *C* releases its lock, yet another process tries to lock bytes 10 and 11, and also blocks. What kinds of problems are introduced into the semantics by this situation? Propose and defend two solutions.

30. Suppose that an lseek system call seeks to a negative offset in a file. Given two possible ways of dealing with it.

31. If a Linux file has protection mode 755 (octal), what can the owner, the owner's group, and everyone else do to the file?

32. Some tape drives have numbered blocks and the ability to overwrite a particular block in place without disturbing the blocks in front of or behind it. Could such a device hold a mounted Linux file system?

33. In Fig. 10-24, both Fred and Lisa have access to the file *x* in their respective directories after linking. Is this access completely symmetrical in the sense that anything one of them can do with it the other one can too?

34. As we have seen, absolute path names are looked up starting at the root directory and relative path names are looked up starting at the working directory. Suggest an efficient way to implement both kinds of searches.

35. When the file */usr/ast/work/f* is opened, several disk accesses are needed to read i-node and directory blocks. Calculate the number of disk accesses required under the assumption that the i-node for the root directory is always in memory, and all directories are one block long.

36. A Linux i-node has 12 disk addresses for data blocks, as well as the addresses of single, double, and triple indirect blocks. If each of these holds 256 disk addresses, what is the size of the largest file that can be handled, assuming that a disk block is 1 KB?

37. When an i-node is read in from the disk during the process of opening a file, it is put into an i-node table in memory. This table has some fields that are not present on the disk. One of them is a counter that keeps track of the number of times the i-node has been opened. Why is this field needed?

38. On multi-CPU platforms, Linux maintains a *runqueue* for each CPU. Is this a good idea? Explain your answer?

39. *Pdflush* threads can be awakened periodically to write back to disk very old pages—older than 30 sec. Why is this necessary?

40. After a system crash and reboot, a recovery program is usually run. Suppose that this program discovers that the link count in a disk i-node is 2, but only one directory entry references the i-node. Can it fix the problem, and if so, how?

41. Make an educated guess as to which Linux system call is the fastest.

42. Is it possible to unlink a file that has never been linked? What happens?

43. Based on the information presented in this chapter, if a Linux ext2 file system were to be put on a 1.44 Mbyte floppy disk, what is the maximum amount of user file data that could be stored on the disk? Assume that disk blocks are 1 KB.

44. In view of all the trouble that students can cause if they get to be superuser, why does this concept exist in the first place?

45. A professor shares files with his students by placing them in a publicly accessible directory on the Computer Science department's Linux system. One day he realizes that a file placed there the previous day was left world-writable. He changes the permissions and verifies that the file is identical to his master copy. The next day he finds that the file has been changed. How could this have happened and how could it have been prevented?

46. Linux supports a system call fsuid. Unlike setuid, which grants the user all the rights of effective id associated with a program he is running, fsuid grants the user who is running the program special rights only with respect to access to files. Why is this feature useful?

47. Write a minimal shell that allows simple commands to be started. It should also allow them to be started in the background.

48. Using assembly language and BIOS calls, write a program that boots itself from a floppy disk on a Pentium-class computer. The program should use BIOS calls to read the keyboard and echo the characters typed, just to demonstrate that it is running.

49. Write a dumb terminal program to connect two Linux computers via the serial ports. Use the POSIX terminal management calls to configure the ports.

50. Write a client-server application which, on request, transfers a large file via sockets. Reimplement the same application using shared memory. Which version do you expect to perform better? Why? Conduct performance measurements with the code you have written and using different file sizes. What are your observations? What do you think happens inside the Linux kernel which results in this behavior?

51. Implement a basic user-level threads library to run on top of Linux. The library API should contain function calls like mythreads_init, mythreads_create, mythreads_join, mythreads_exit, mythreads_yield, mythreads_self, and perhaps a few others. Next, implement these synchronization variables to enable safe concurrent operations: mythreads_mutex_init, mythreads_mutex_lock, mythreads_mutex_unlock. Before start-

ing, clearly define the API and specify the semantics of each of the calls. Next implement the user-level library with a simple, round-robin preemptive scheduler. You will also need to write one or more multithreaded applications, which use your library, in order to test it. Finally, replace the simple scheduling mechanism with another one which behaves like the Linux 2.6 O(1) scheduler described in this chapter. Compare the performance your application(s) receive when using each of the schedulers.

11

CASE STUDY 2: WINDOWS VISTA

Windows is a modern operating system that runs on consumer and business desktop PCs and enterprise servers. The most recent desktop version is **Windows Vista**. The server version of Windows Vista is called **Windows Server 2008**. In this chapter we will examine various aspects of Windows Vista, starting with a brief history, then moving on to its architecture. After this we will look at processes, memory management, caching, I/O, the file system, and finally, security.

11.1 HISTORY OF WINDOWS VISTA

Microsoft's development of the Windows operating system for PC-based computers as well as servers can be divided into three eras: **MS-DOS**, **MS-DOS-based Windows**, and **NT-based Windows**. Technically, each of these systems is substantially different from the others. Each of these was dominant during different decades in the history of the personal computer. Fig. 11-1 shows the dates of the major Microsoft operating system releases for desktop computers (omitting the popular Microsoft Xenix version of UNIX, which Microsoft sold to the Santa Cruz Operation (SCO) in 1987). Below we will briefly sketch each of the eras shown in the table.

Year	MS-DOS	MS-DOS-based Windows	NT-based Windows	Notes
1981	MS-DOS 1.0			Initial release for IBM PC
1983	MS-DOS 2.0			Support for PC/XT
1984	MS-DOS 3.0			Support for PC/AT
1990		Windows 3.0		Ten million copies in 2 years
1991	MS-DOS 5.0			Added memory management
1992		Windows 3.1		Runs only on 286 and later
1993			Windows NT 3.1	
1995	MS-DOS 7.0	Windows 95		MS-DOS embedded in Win 95
1996			Windows NT 4.0	
1998		Windows 98		
2000	MS-DOS 8.0	Windows Me	Windows 2000	Win Me was inferior to Win 98
2001			Windows XP	Replaced Windows 98
2006			Windows Vista	

Figure 11-1. Major releases in the history of Microsoft operating systems for desktop PCs.

11.1.1 1980s: MS-DOS

In the early 1980s IBM, at the time the biggest and most powerful computer company in the world, was developing a **personal computer** based the Intel 8088 microprocessor. Since the mid-1970s, Microsoft had become the leading provider of the BASIC programming language for 8-bit microcomputers based on the 8080 and Z-80. When IBM approached Microsoft about licensing BASIC for the new IBM PC, Microsoft readily agreed and suggested that IBM contact Digital Research to license its CP/M operating system, since Microsoft was not then in the operating system business. IBM did that, but the president of Digital Research, Gary Kildall, was too busy to meet with IBM, so it came back to Microsoft. Within a short time, Microsoft bought a CP/M clone from a local company, Seattle Computer Products, ported it to the IBM PC, and licensed it to IBM. It was then renamed **MS-DOS 1.0** (**MicroSoft Disk Operating System**) and shipped with the first IBM PC in 1981.

MS-DOS was a 16-bit real-mode, single-user, command-line-oriented operating system consisting of 8 KB of memory resident code. Over the next decade, both the PC and MS-DOS continued to evolve, adding more features and capabilities. By 1986 when IBM built the PC/AT based on the Intel 286, MS-DOS had grown to be 36 KB, but continued to be a command-line-oriented, one application at a time, operating system.

11.1.2 1990s: MS-DOS-based Windows

Inspired by the graphical user interface of research systems at Stanford Research Institute and Xerox PARC, and their commercial progeny, the Apple Lisa and the Apple Macintosh, Microsoft decided to give MS-DOS a graphical user interface that it called **Windows**. The first two versions of Windows (1985 and 1987) were not very successful, due in part to the limitations of the PC hardware available at the time. In 1990 Microsoft released Windows 3.0 for the Intel 386, and sold over one million copies in six months.

Windows 3.0 was not a true operating system, but a graphical environment built on top of MS-DOS, which was still in control of the machine and the file system. All programs ran in the same address space and a bug in any one of them could bring the whole system to a frustrating halt.

In August 1995, **Windows 95** was released. It contained many of the features of a full-blown operating system, including virtual memory, process management, and multiprogramming, and introduced 32-bit programming interfaces. However, it still lacked security, and provided poor isolation between applications and the operating system. Thus the problems with instability continued, even with the subsequent releases of **Windows 98** and **Windows Me**, where MS-DOS was still there running 16-bit assembly code in the heart of the Windows operating system.

11.1.3 2000s: NT-based Windows

By end of the 1980s, Microsoft realized that continuing to evolve an operating system with MS-DOS at its center was not the best way to go. PC hardware was continuing to increase in speed and capability. and ultimately the PC market would collide with the desktop workstation and enterprise server computing markets, where UNIX was the dominant operating system. Microsoft was also concerned that the Intel microprocessor family might not continue to be competitive, as it was already being challenged by RISC architectures. To address these issues, Microsoft recruited a group of engineers from DEC led by Dave Cutler, one of the key designers of DEC's VMS operating system. Cutler was chartered to develop a brand-new 32-bit operating system that was intended to implement **OS/2**, the operating system API that Microsoft was jointly developing with IBM at the time. The original design documents by Cutler's team called the system *NT OS/2*.

Cutler's system was called **NT** for New Technology (and also because the original target processor was the new Intel 860, code named the N10). NT was designed to be portable across different processors and emphasized security and reliability, as well as compatibility with the MS-DOS-based versions of Windows. Cutler's background at DEC shows in various places, with there being more than a passing similarity between the design of NT and that of VMS and other operating systems designed by Cutler, shown in Fig. 11-2.

Year	DEC operating system	Characteristics
1973	RSX-11M	16-bit, multi-user, real-time, swapping
1978	VAX/VMS	32-bit, virtual memory
1987	VAXELAN	Real-time
1988	PRISM/Mica	Canceled in favor of MIPS/Ultrix

Figure 11-2. DEC Operating Systems developed by Dave Cutler.

When DEC's engineers (and later its lawyers) saw how similar NT was to VMS (and also to its never-released successor, MICA) a discussion ensued between DEC and Microsoft about Microsoft's use of DEC's intellectual property. The issue was eventually settled out of court. In addition, Microsoft agreed to support NT on the DEC Alpha for a certain period of time. However, none of this was enough to save DEC from its fixation on minicomputers and disdain for personal computers, typified by DEC founder Ken Olsen's 1977 remark: "There is no reason anyone would want a computer in their [*sic*] home." In 1998, what was left of DEC was sold to Compaq, which was later bought by Hewlett-Packard.

Programmers familiar only with UNIX find the architecture of NT to be quite different. This is not just because of the influence of VMS, but also because of the differences in the computer systems that were common at the time of design. UNIX was first designed in the 1970s for single-processor, 16-bit, tiny-memory, swapping systems where the process was the unit of concurrency and composition, and fork/exec were inexpensive operations (since swapping systems frequently copy processes to disk anyway). NT was designed in the early 1990s, when multiprocessor, 32-bit, multi-megabyte, virtual memory systems were common. In NT threads are the unit of concurrency, dynamic libraries the units of composition, and fork/exec are implemented by a single operation to create a new process *and* run another program without first making a copy.

The first version of NT-based Windows (Windows NT 3.1) was released in 1993. It was called 3.1 to correspond with the then-current consumer Windows 3.1. The joint project with IBM had foundered, so though the OS/2 interfaces were still supported, the primary interfaces were 32-bit extensions of the Windows APIs, called **Win32**. Between the time NT was started and first shipped, **Windows 3.0** had been released, and was extremely successful commercially. It too was able to run Win32 programs, but using the *Win32s* compatibility library.

Like the first version of MS-DOS-based Windows, NT-based Windows was not initially successful. NT required more memory, there were few 32-bit applications available, and incompatibilities with device drivers and applications caused many customers to stick with MS-DOS-based Windows which Microsoft was still improving, releasing Windows 95 in 1995. Windows 95 provided native 32-bit programming interfaces like NT, but better compatibility with existing 16-bit

software and applications. Not surprisingly, NT's early success was in the server market, competing with VMS and NetWare.

NT did meet its portability goals, with additional releases in 1994 and 1995 adding support for (little-endian) MIPS and PowerPC architectures. The first major upgrade to NT came with **Windows NT 4.0** in 1996. This system had the power, security, and reliability of NT, but also sported the same user interface as the by-then very popular Windows 95.

Fig. 11-3 shows the relationship of the Win32 API to Windows. Having a common API across both the MS-DOS-based and NT-based Windows was important to the success of NT.

This compatibility made it much easier for users to migrate from Windows 95 to NT, and the operating system became a strong player in the high-end desktop market as well as servers. However, customers were not as willing to adopt other processor architectures, and of the four architectures Windows NT 4.0 supported in 1996 (the DEC Alpha was added in that release), only the x86 (i.e., Pentium family) was still actively supported by the time of the next major release, **Windows 2000**.

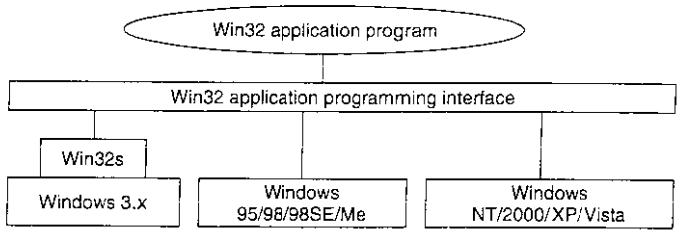

Figure 11-3. The Win32 API allows programs to run on almost all versions of Windows.

Windows 2000 represented a significant evolution for NT. The key technologies added were plug-and-play (for consumers who installed a new PCI card, eliminating the need to fiddle with jumpers), network directory services (for enterprise customers), improved power management (for notebook computers), and an improved GUI (for everyone).

The technical success of Windows 2000 led Microsoft to push toward the deprecation of Windows 98 by enhancing the application and device compatibility of the next NT release, **Windows XP**. Windows XP included a friendlier new look-and-feel to the graphical interface, bolstering Microsoft's strategy of hooking consumers and reaping the benefit as they pressured their employers to adopt systems with which they were already familiar. The strategy was overwhelmingly successful, with Windows XP being installed on hundreds of millions of PCs over its first few years, allowing Microsoft to achieve its goal of effectively ending the era of MS-DOS-based Windows.

Windows XP represented a new development reality for Microsoft, with separate releases for desktop clients from those for enterprise servers. The system was simply too complex to produce high-quality client and server releases at the same time. **Windows 2003** was the server release complementing the Windows XP client operating system. It provided support for the 64-bit Intel Itanium (IA64) and, at its first service pack, support for the AMD x64 architecture on both servers and desktops. Microsoft used the time between the client and server releases to add server-specific features, and conduct extended testing focused on the aspects of the system primarily used by businesses. Fig. 11-4 shows the relationship of client and server releases of Windows.

Year	Client version	Year	Server version
1996	Windows NT	1996	Windows NT Server
1999	Windows 2000	1999	Windows 2000 Server
2001	Windows XP	2003	Windows Server 2003
2006	Windows Vista	2007	Windows Server 2008

Figure 11-4. Split client and server releases of Windows.

Microsoft followed up Windows XP by embarking on an ambitious release to kindle renewed excitement among PC consumers. The result, **Windows Vista**, was completed in late 2006, more than five years after Windows XP shipped. Windows Vista boasted yet another redesign of the graphical interface, and new security features under the covers. Most of the changes were in customer-visible experiences and capabilities. The technologies under the covers of the system improved incrementally, with much clean-up of the code and many improvements in performance, scalability, and reliability. The server version of Vista (Windows Server 2008) was delivered about a year after the consumer version. It shares the same core system components, such as the kernel, drivers, and low-level libraries and programs with Vista.

The human story of the early development of NT is related in the book *Showstopper* (Zachary, 1994). The book tells a lot about the key people involved, and the difficulties of undertaking such an ambitious software development project.

11.1.4 Windows Vista

The release of Windows Vista culminated Microsoft's most extensive operating system project to date. The initial plans were so ambitious that a couple of years into its development Vista had to be restarted with a smaller scope. Plans to rely heavily on Microsoft's type-safe, garbage-collected .NET language C# were shelved, as were some significant features such as the WinFS unified storage system for searching and organizing data from many different sources. The size of

the full operating system is staggering. The original NT release of 3 million lines of C/C++ that had grown to 16 million in NT 4, 30 million in 2000, and 50 million in XP, is over 70 million lines in Vista.

Much of the size is due to Microsoft's emphasis on adding many new features to its products in every release. In the main *system32* directory, there are 1600 dynamic link libraries (DLLs) and 400 executables (EXEs), and that does not include the other directories containing the myriad of applets included with the operating system that allow users to surf the Web, play music and video, send e-mail, scan documents, organize photos, and even make movies. Because Microsoft wants customers to switch to new versions, it maintains compatibility by generally keeping all the features, APIs, *applets* (small applications), etc., from the previous version. Few things ever get deleted. The result is that Windows grows dramatically release to release. Technology has kept up, and Windows' distribution media have moved from floppy, to CD, and now with Windows Vista, DVD.

The bloat in features and applets at the top of Windows makes meaningful size comparisons with other operating systems problematic because the definition of what is or is not part of an operating system is difficult to decide. At the lower layers of operating systems, there is more correspondence because the functions performed are very similar. Even so we can see a big difference in the size of Windows. Fig. 11-5 compares the Windows and Linux kernels for three key functional areas: CPU scheduling, I/O infrastructure, and Virtual Memory. The first two components are half again as large in Windows, but the Virtual Memory component is an order of magnitude larger—due to the large number of features, the virtual memory model used, and implementation techniques that trade off code size to achieve higher performance.

Kernel area	Linux	Vista
CPU Scheduler	50,000	75,000
I/O infrastructure	45,000	60,000
Virtual Memory	25,000	175,000

Figure 11-5. Comparison of lines of code for selected kernel-mode modules in Linux and Windows (from Mark Russinovich, co-author of *Microsoft Windows Internals*).

11.2 PROGRAMMING WINDOWS VISTA

It is now time to start our technical study of Windows Vista. However, before getting into the details of the internal structure we will first take a look at the native NT API for system calls, and then the Win32 programming subsystem. Despite the availability of POSIX, virtually all the code written for Windows uses either Win32 directly, or .NET—which itself runs on top of Win32.

Fig. 11-6 shows the layers of the Windows Operating System. Beneath the applet and GUI layers of Windows are the programming interfaces that applications build on. As in most operating systems, these consist largely of code libraries (DLLs) which programs dynamically link to for access to operating system features. Windows also includes a number of programming interfaces which are implemented as services that run as separate processes. Applications communicate with user-mode services through remote-procedure-calls (RPC).

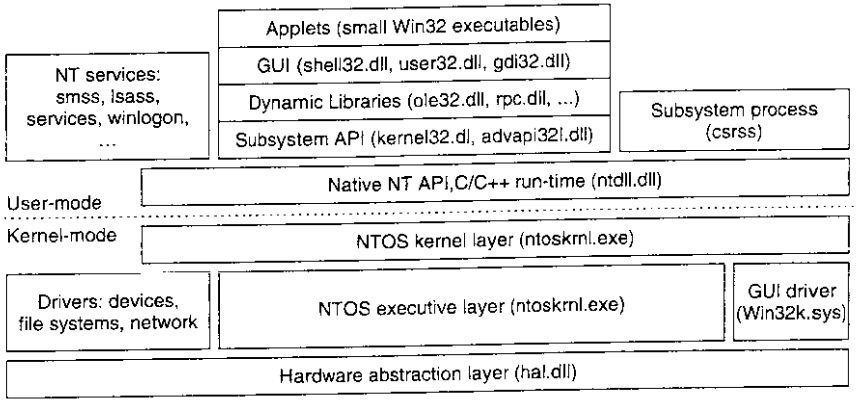

Figure 11-6. The programming layers in Windows.

The core of the NT operating system is the **NTOS** kernel-mode program *ntoskrnl.exe*), which provides the traditional system-call interfaces upon which the rest of the operating system is built. In Windows, only programmers at Microsoft write to the system call layer. The published user-mode interfaces all belong to operating system personalities that are implemented using **subsystems** that run on top of the NTOS layers.

Originally NT supported three personalities: OS/2, POSIX and Win32. OS/2 was discarded in Windows XP. POSIX was also removed, but customers can get an improved POSIX subsystem called *Interix* as part of Microsoft's *Services For UNIX* (SFU), so all the infrastructure to support POSIX remains in the system. Most Windows applications are written to use Win32, although Microsoft also supports other APIs.

Unlike Win32, .NET is not built as an official subsystem on the native NT kernel interfaces. Instead .NET is built on top of the Win32 programming model. This allows .NET to interoperate well with existing Win32 programs, which was never the goal with the POSIX and OS/2 subsystems. The WinFX API includes many of the features of Win32, and in fact many of the functions in the WinFX *Base Class Library* are simply wrappers around Win32 APIs. The advantages of WinFX have to do with the richness of the object types supported, the simplified

consistent interfaces, and use of the .NET Common Language Run-time (CLR), including garbage-collection.

As shown in Fig. 11-7, NT subsystems are built out of four components: a subsystem process, a set of libraries, hooks in *CreateProcess*, and support in the kernel. A subsystem process is really just a service. The only special property is that it is started by the *smss.exe* (session manager) program—the initial user-mode program started by NT—in response to a request from *CreateProcess* in Win32 or the corresponding API in a different subsystem.

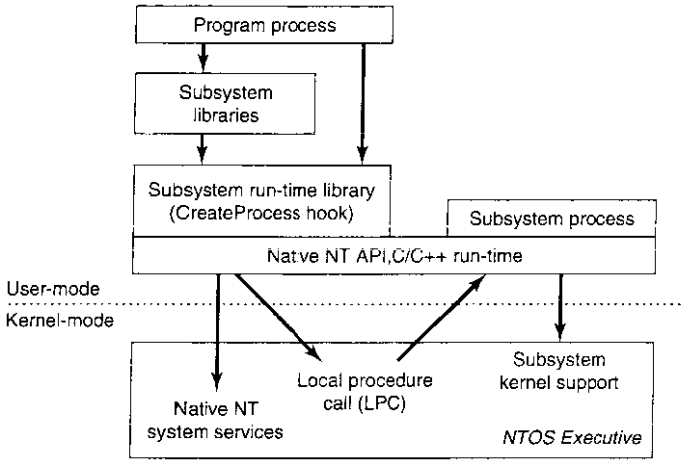

Figure 11-7. The components used to build NT subsystems.

The set of libraries implements both higher-level operating-system functions specific to the subsystem as well as containing the stub routines which communicate between processes using the subsystem (shown on the left) and the subsystem process itself (shown on the right). Calls to the subsystem process normally take place using the kernel-mode **LPC (Local Procedure Call)** facilities, which implement cross-process procedure calls.

The hook in Win32 CreateProcess detects which subsystem each program requires by looking at the binary image. It then asks *smss.exe* to start the subsystem process **csrss.exe** (if it is not already running). The subsystem process then takes over responsibility for loading the program. The implementation of other subsystems have a similar hook (e.g., in the exec system call in POSIX).

The NT kernel was designed to have a lot of general purpose facilities that can be used for writing operating-system-specific subsystems. But there is also special code that must be added to correctly implement each subsystem. As examples, the native NtCreateProcess system call implements process duplication in support of POSIX fork system call, and the kernel implements a particular kind

of string table for Win32 (called *atoms*) which allows read-only strings to be efficiently shared across processes.

The subsystem processes are native NT programs which use the native system calls provided by the NT kernel and core services, such as *smss.exe* and *lsass.exe* (local security administration). The native system calls include cross-process facilities to manage virtual addresses, threads, handles, and exceptions in the processes created to run programs written to use a particular subsystem.

11.2.1 The Native NT Application Programming Interface

Like all other operating systems, Windows Vista has a set of system calls it can perform. In Windows Vista these are implemented in the NTOS executive layer that runs in kernel mode. Microsoft has published very few of the details of these native system calls. They are used internally by lower-level programs that ship as part of the operating system (mainly services and the subsystems), as well as kernel-mode device drivers. The native NT system calls do not really change very much from release to release, but Microsoft chose not to make them public so that applications written for Windows would be based on Win32 and thus more likely to work with both the MS-DOS-based and NT-based Windows systems, since the Win32 API is common to both.

Most of the native NT system calls operate on kernel-mode objects of one kind or another, including files, processes, threads, pipes, semaphores, and so on. Fig. 11-8 gives a list of some of the common categories of kernel-mode objects supported by NT in Windows Vista. Later, when we discuss the object manager, we will provide further details on the specific object types.

Object category	Examples
Synchronization	Semaphores, mutexes, events, IPC ports, I/O completion queues
I/O	Files, devices, drivers, timers
Program	Jobs, processes, threads, sections, tokens
Win32 GUI	Desktops, application callbacks

Figure 11-8. Common categories of kernel-mode object types.

Sometimes use of the term *object* regarding the data structures manipulated by the operating system can be confusing because it is mistaken for *object-oriented*. Operating system objects do provide data hiding and abstraction, but they lack some of the most basic properties of object-oriented systems such as inheritance and polymorphism.

In the native NT API there are calls available to create new kernel-mode objects or access existing ones. Every call creating or opening an object returns a result called a **handle** to the caller. The handle can subsequently be used to perform operations on the object. Handles are specific to the process that created them. In

general handles cannot be passed directly to another process and used to refer to the same object. However, under certain circumstances, it is possible to duplicate a handle into the handle table of other processes in a protected way, allowing processes to share access to objects—even if the objects are not accessible in the namespace. The process duplicating each handle must itself have handles for both the source and target process.

Every object has a **security descriptor** associated with it, telling in detail who may and may not perform what kinds of operations on the object based on the access requested. When handles are duplicated between processes, new access restrictions can be added that are specific to the duplicated handle. Thus a process can duplicate a read-write handle and turn it into a read-only version in the target process.

Not all system-created data structures are objects and not all objects are kernel-mode objects. The only ones that are true kernel-mode objects are those that need to be named, protected, or shared in some way. Usually, these kernel-mode objects represent some kind of programming abstraction implemented in the kernel. Every kernel-mode object has a system-defined type, has well-defined operations on it, and occupies storage in kernel memory. Although user-mode programs can perform the operations (by making system calls), they cannot get at the data directly.

Fig. 11-9 shows a sampling of the native APIs, all of which use explicit handles to manipulate kernel-mode objects such as processes, threads, IPC ports, and **sections** (which are used to describe memory objects that can be mapped into address spaces). NtCreateProcess returns a handle to a newly created process object, representing an executing instance of the program represented by the SectionHandle. DebugPortHandle is used to communicate with a debugger when giving it control of the process after an exception (e.g., dividing-by-zero or accessing invalid memory). ExceptPortHandle is used to communicate with a subsystem process when errors occur and are not handled by an attached debugger.

```
NtCreateProcess(&ProcHandle, Access, SectionHandle, DebugPortHandle, ExceptPortHandle, ...)
NtCreateThread(&ThreadHandle, ProcHandle, Access, ThreadContext, CreateSuspended, ...)
NtAllocateVirtualMemory(ProcHandle, Addr, Size, Type, Protection, ...)
NtMapViewOfSection(SectHandle, ProcHandle, Addr, Size, Protection, ...)
NtReadVirtualMemory(ProcHandle, Addr, Size, ...)
NtWriteVirtualMemory(ProcHandle, Addr, Size, ...)
NtCreateFile(&FileHandle, FileNameDescriptor, Access, ...)
NtDuplicateObject(srcProcHandle, srcObjHandle, dstProcHandle, dstObjHandle, ...)
```

Figure 11-9. Examples of native NT API calls that use handles to manipulate objects across process boundaries.

NtCreateThread takes ProcHandle because it can create a thread in any process for which the calling process has a handle (with sufficient access rights). Similarly, NtAllocateVirtualMemory, NtMapViewOfSection, NtReadVirtualMemory and NtWriteVirtualMemory allow one process to operate not only on its own address space, but to allocate virtual addresses, map sections, and read or write virtual memory in other processes. NtCreateFile is the native API call for creating a new fuke, or opening an existing one. NtDuplicateObject is the API call for duplicating handles from one process to another.

Kernel-mode objects are of course not unique to Windows. UNIX systems also support a variety of kernel-mode objects, such as files, network sockets, pipes, devices, processes, and inter-process communication (IPC) facilities like shared-memory, message ports, semaphores, and I/O devices. In UNIX there are a variety of ways of naming and accessing objects, such as file descriptors, process IDs, and integer IDs for SystemV IPC objects, and i-nodes for devices. The implementation of each class of UNIX objects is specific to the class. Files and sockets use different facilities than the SystemV IPC mechanisms or processes or devices.

Kernel objects in Windows use a uniform facility based on handles and names in the NT namespace to reference kernel objects, along with a unified implementation in a centralized **object manager**. Handles are per-process but, as described above, can be duplicated into another process. The object manager allows objects to be given names when they are created, and then opened by name to get handles for the objects.

The object manager uses **Unicode** (wide characters) to represent names in the **NT namespace**. Unlike UNIX, NT does not generally distinguish between upper and lower case (it is *case-preserving* but *case-insensitive*). The NT namespace is a hierarchical tree-structured collection of directories, symbolic links and objects.

The object manager also provides unified facilities for synchronization, security, and object lifetime management. Whether the general facilities provided by the object manager are made available to users of any particular object is up to the executive components, as they provide the native APIs that manipulate each object type.

It is not only applications that use objects managed by the object manager. The operating system itself can also create and use objects—and does so heavily. Most of these objects are created to allow one component of the system to store some information for a substantial period of time or to pass some data structure to another component, and yet benefit from the naming and lifetime support of the object manager. For example, when a device is discovered, one or more **device objects** are created to represent the device and to logically describe how the device is connected to the rest of the system. To control the device a device driver is loaded, and a **driver object** is created holding its properties and providing pointers to the functions it implements for processing the I/O requests. Within the operating system the driver is then referred to by using its object. The driver can

also be accessed directly by name rather than indirectly through the devices it controls (e.g., to set parameters governing its operation from user mode).

Unlike UNIX, which places the root of its namespace in the file system, the root of the NT namespace is maintained in the kernel's virtual memory. This means that NT must recreate its top-level namespace every time the system boots. Using kernel virtual memory allows NT to store information in the namespace without first having to start the file system running. It also makes it much easier for NT to add new types of kernel-mode objects to the system because the formats of the file systems themselves do not have to be modified for each new object type.

A named object can be marked *permanent*, meaning that it continues to exist until explicitly deleted or the system reboots, even if no process currently has a handle for the object. Such objects can even extend the NT namespace by providing *parse* routines that allow the objects to function somewhat like mount points in UNIX. File systems and the registry use this facility to mount volumes and hives onto the NT namespace. Accessing the device object for a volume gives access to the raw volume, but the device object also represents an implicit mount of the volume into the NT namespace. The individual files on a volume can be accessed by concatenating the volume-relative filename onto the end of the name of the device object for that volume.

Permanent names are also used to represent synchronization objects and shared memory, so that they can be shared by processes without being continually recreated as processes stop and start. Device objects and often driver objects are given permanent names, giving them some of the persistence properties of the special i-nodes kept in the */dev* directory of UNIX.

We will describe many more of the features in the native NT API in the next section, where we discuss the Win32 APIs that provide wrappers around the NT system calls.

11.2.2 The Win32 Application Programming Interface

The Win32 function calls are collectively called the **Win32 API**. These interfaces are publicly disclosed and fully documented. They are implemented as library procedures that either wrap the native NT system calls used to get the work done or, in some cases, do the work right in user mode. Though the native NT APIs are not published, most of the functionality they provide is accessible through the Win32 API. The existing Win32 API calls rarely change with new releases of Windows, though many new functions are added to the API.

Fig. 11-10 shows various low-level Win32 API calls and the native NT API calls that they wrap. What is interesting about the figure is how uninteresting the mapping is. Most low-level Win32 functions have native NT equivalents, which is not surprising as Win32 was designed with NT in mind. In many cases the

Win32 layer must manipulate the Win32 parameters to map them onto NT. For example, canonicalizing pathnames and mapping onto the appropriate NT pathnames, including special MS-DOS device names (like *LPT:*). The Win32 APIs for creating processes and threads also must notify the Win32 subsystem process, *csrss.exe*, that there are new processes and threads for it to supervise, as we will describe in Sec. 11.4.

Win32 call	Native NT API call
CreateProcess	NtCreateProcess
CreateThread	NtCreateThread
SuspendThread	NtSuspendThread
CreateSemaphore	NtCreateSemaphore
ReadFile	NtReadFile
DeleteFile	NtSetInformationFile
CreateFileMapping	NtCreateSection
VirtualAlloc	NtAllocateVirtualMemory
MapViewOfFile	NtMapViewOfSection
DuplicateHandle	NtDuplicateObject
CloseHandle	NtClose

Figure 11-10. Examples of Win32 API calls and the native NT API calls that they wrap.

Some Win32 calls take pathnames, whereas the equivalent NT calls use handles. So the wrapper routines have to open the files, call NT, and then close the handle at the end. The wrappers also translate the Win32 APIs from ANSI to Unicode. The Win32 functions shown in Fig. 11-10 that use strings as parameters are actually two APIs, for example **CreateProcessW** and **CreateProcessA**. The strings passed to the latter API must be translated to Unicode before calling the underlying NT API, since NT works only with Unicode.

Since few changes are made to the existing Win32 interfaces in each release of Windows, in theory the binary programs that ran correctly on any previous release will continue to run correctly on a new release. In practice there are often many compatibility problems with new releases. Windows is so complex that a few seemingly inconsequential changes can cause application failures. And applications themselves are often to blame, since they frequently make explicit checks for specific OS versions or fall victim to their own latent bugs that are exposed when they run on a new release. Nevertheless, Microsoft makes an effort in every release to test a wide variety of applications to find incompatibilities and either correct them or provide application-specific workarounds.

Windows supports two special execution environments both called Windows-on-Windows (WOW). **WOW32** is used on 32-bit x86 systems to run 16-bit

Windows 3.x applications by mapping the system calls and parameters between the 16-bit and 32-bit worlds. Similarly **WOW64** allows 32-bit Windows applications to run on x64 systems.

The Windows API philosophy is very different from the UNIX philosophy. In the latter, the operating system functions are simple, with few parameters and few places where there are multiple ways to perform the same operation. Win32 provides very comprehensive interfaces with many parameters, often with three or four ways of doing the same thing, and mixing together low-level and high-level functions, like CreateFile and CopyFile.

This means Win32 provides a very rich set of interfaces, but it also introduces much complexity due to the poor layering of a system that intermixes both high-level and low-level functions in the same API. For our study of operating systems, only the low-level functions of the Win32 API that wrap the native NT API are relevant, so those are what we will focus on.

Win32 has calls for creating and managing processes and threads. There are also many calls that relate to inter-process communication, such as creating, destroying, and using mutexes, semaphores, events, communication ports, and other IPC objects.

Although much of the memory management system is invisible to programmers, one important feature is visible: namely the ability of a process to map a file onto a region of its virtual memory. This allows threads running in a process the ability to read and write parts of the file using pointers without having to explicitly perform read and write operations to transfer data between the disk and memory. With memory-mapped files the memory management system itself performs the I/Os as needed (demand paging).

Windows implements memory-mapped files using three completely different facilities. First it provides interfaces which allow processes to manage their own virtual address space, including reserving ranges of addresses for later use. Second, Win32 supports an abstraction called a *file mapping* which is used to represent addressable objects like files (a file mapping is called a *section* in the NT layer). Most often, file mappings are created to refer to files using a file handle, but they can also be created to refer to private pages allocated from the system pagefile.

The third facility maps *views* of file mappings into a process' address space. Win32 only allows a view to be created for the current process, but the underlying NT facility is more general, allowing views to be created for any process for which you have a handle with the appropriate permissions. Separating the creation of a file mapping from the operation of mapping the file into the address space is a different approach than used in the mmap function in UNIX.

In Windows the file mappings are kernel-mode objects represented by a handle. Like most handles, file mappings can be duplicated into other processes. Each of these processes can map the file mapping into its own address space as it sees fit. This is useful for sharing private memory between processes without having

to create files for sharing. At the NT layer, file mappings (sections) can also be made persistent in the NT namespace and accessed by name.

An important area for many programs is file I/O. In the basic Win32 view, a file is just a linear sequence of bytes. Win32 provides over 60 calls for creating and destroying files and directories, opening and closing files, reading and writing them, requesting and setting file attributes, locking ranges of bytes, and many more fundamental operations on both the organization of the file system and access to individual files.

There are also advanced facilities for managing data in files. In addition to the primary data stream, files stored on the NTFS file system can have additional data streams. Files (and even entire volumes) can be encrypted. Files can be compressed, and/or represented as a sparse stream of bytes where missing regions of data in the middle occupy no storage on disk. File system volumes can be organized out of multiple separate disk partitions using various levels of RAID storage. Modifications to files or directory sub-trees can be detected through a notification mechanism, or by reading the **journal** that NTFS maintains for each volume.

Each file system volume is implicitly mounted in the NT namespace, according to the name given to the volume, so a file *\foo\bar* might be named, for example, *\Device\HarddiskVolume\foo\bar*. Internal to each NTFS volume, mount points (called *reparse points* in Windows) and symbolic links are supported to help organize the individual volumes.

The low-level I/O model in Windows is fundamentally asynchronous. Once an I/O operation is begun, the system call can return and allow the thread which initiated the I/O to continue in parallel with the I/O operation. Windows supports cancellation, as well as a number of different mechanisms for threads to synchronize with I/O operations when they complete. Windows also allows programs to specify that I/O should be synchronous when a file is opened, and many library functions, such as the C library and many Win32 calls, specify synchronous I/O for compatibility or to simplify the programming model. In these cases the executive will explicitly synchronize with I/O completion before returning to user mode.

Another area for which Win32 provides calls is security. Every thread is associated with a kernel-mode object, called a **token**, which provides information about the identity and privileges associated with the thread. Every object can have an **ACL (Access Control List)** telling in great detail precisely which users may access it and which operations they may perform on it. This approach provides for fine-grained security in which specific users can be allowed or denied specific access to every object. The security model is extensible, allowing applications to add new security rules, such as limiting the hours access is permitted.

The Win32 namespace is different than the native NT namespace described in the previous section. Only parts of the NT namespace are visible to Win32 APIs (though the entire NT namespace can be accessed through a Win32 hack that uses

special prefix strings, like "\\."). In Win32, files are accessed relative to *drive letters*. The NT directory *\DosDevices* contains a set of symbolic links from drive letters to the actual device objects. For example *\DosDevices\C:* might be a link to *\Device\HarddiskVolume1*. This directory also contains links for other Win32 devices, such as *COM1:*, *LPT1:*, and *NUL:* (for the serial and printer ports, and the all-important null device). *\DosDevices* is really a symbolic link to *\??* which was chosen for efficiency. Another NT directory, *\BaseNamedObjects* is used to store miscellaneous named kernel-mode objects accessible through the Win32 API. These include synchronization objects like semaphores, shared memory, timers, and communication ports. MS-DOS and device names.

In addition to low-level system interfaces we have described, the Win32 API also supports many functions for GUI operations, including all the calls for managing the graphical interface of the system. There are calls for creating, destroying, managing and using windows, menus, tool bars, status bars, scroll bars, dialog boxes, icons, and many more items that appear on the screen. There are calls for drawing geometric figures, filling them in, managing the color palettes they use, dealing with fonts, and placing icons on the screen. Finally, there are calls for dealing with the keyboard, mouse and other human input devices as well as audio, printing, and other output devices.

The GUI operations work directly with the *win32k.sys* driver using special interfaces to access these functions in kernel mode from user-mode libraries. Since these calls do not involve the core system calls in the NTOS executive, we will not say more about them.

11.2.3 The Windows Registry

The root of the NT namespace is maintained in the kernel. Storage, such as file system volumes, is attached to the NT namespace. Since the NT namespace is constructed afresh every time the system boots, how does the system know about any specific details of the system configuration? The answer is that Windows attaches a special kind of file system (optimized for small files) to the NT namespace. This file system is called the **registry**. The registry is organized into separate volumes called **hives**. Each hive is kept in a separate file (in the directory *C:\Windows\system32\config* of the boot volume). When a Windows system boots, one particular hive named *SYSTEM* is loaded into memory by the same boot program that loads the kernel and other boot files, such as boot drivers, from the boot volume.

Windows keeps a great deal of crucial information in the SYSTEM hive, including information about what drivers to use with what devices, what software to run initially, and many parameters governing the operation of the system. This information is used even by the boot program itself to determine which drivers are boot drivers, being needed immediately upon boot. Such drivers include those

that understand the file system and disk drivers for the volume containing the operating system itself.

Other configuration hives are used after the system boots to describe information about the software installed on the system, particular users, and the classes of user-mode **COM (Component Object-Model)** objects that are installed on the system. Login information for local users is kept in the SAM (Security Access Manager) hive. Information for network users is maintained by the *lsass* service in the SECURITY hive, and coordinated with the network directory servers so that users can have a common account name and password across an entire network. A list of the hives used in Windows Vista is shown in Fig. 11-11.

Hive file	Mounted name	Use
SYSTEM	HKLM TEM	OS configuration information, used by kernel
HARDWARE	HKLM DWARE	In-memory hive recording hardware detected
BCD	HKLM BCD*	Boot Configuration Database
SAM	HKLM	Local user account information
SECURITY	HKLM URITY	lsass' account and other security information
DEFAULT	HKEY_USERS .DEFAULT	Default hive for new users
NTUSER.DAT	HKEY_USERS <user id>	User-specific hive, kept in home directory
SOFTWARE	HKLM TWARE	Application classes registered by COM
COMPONENTS	HKLM NENTS	Manifests and dependencies for sys. components

Figure 11-11. The registry hives in Windows Vista. HKLM is a short-hand for *HKEY_LOCAL_MACHINE*.

Prior to the introduction of the registry, configuration information in Windows was kept in hundreds of *.ini* (initialization) files spread across the disk. The registry gathers these files into a central store, which is available early in the process of booting the system. This is important for implementing Windows plug-and-play functionality. But the registry has become very disorganized as Windows has evolved. There are poorly defined conventions about how the configuration information should be arranged, and many applications take an ad hoc approach. Most users, applications, and all drivers run with full privileges, and frequently modify system parameters in the registry directly—sometimes interfering with each other and destabilizing the system.

The registry is a strange cross between a file system and a database, and yet really unlike either. Entire books have been written describing the registry (Born, 1998; Hipson, 2000; and Ivens, 1998), and many companies have sprung up to sell special software just to manage the complexity of the registry.

To explore the registry Windows has a GUI program called **regedit** that allows you to open and explore the directories (called *keys*) and data items (called *values*). Microsoft's new **PowerShell** scripting language can also be useful for walking through the keys and values of the registry as if they were directories and

files. A more interesting tool to use is *procmon*, which is available from Microsoft's tools' Website: *www.microsoft.com/technet/sysinternals*.

Procmon watches all the registry accesses that take place in the system and is very illuminating. Some programs will access the same key over and over tens of thousands of times.

As the name implies, *regedit* allows users to edit the registry—but be very careful if you ever do. It is very easy to render your system unable to boot, or damage the installation of applications so that you cannot fix them without a lot of wizardry. Microsoft promised to clean up the registry in future releases, but for now it is a huge mess—far more complicated than the configuration information maintained in UNIX.

Beginning with Windows Vista Microsoft has introduced a kernel-based transaction manager with support for coordinated transactions that span both file system and registry operations. Microsoft plans to use this facility in the future to avoid some of the metadata corruption problems that occur when software installation does not complete correctly and leaves around partial state in the system directories and registry hives.

The registry is accessible to the Win32 programmer. There are calls to create and delete keys, look up values within keys, and more. Some of the more useful ones are listed in Fig. 11-12.

Win32 API function	Description
RegCreateKeyEx	Create a new registry key
RegDeleteKey	Delete a registry key
RegOpenKeyEx	Open a key to get a handle to it
RegEnumKeyEx	Enumerate the subkeys subordinate to the key of the handle
RegQueryValueEx	Look up the data for a value within a key

Figure 11-12. Some of the Win32 API calls for using the registry

When the system is turned off, most of the registry information is stored on the disk in the hives. Because their integrity is so critical to correct system functioning, backups are made automatically and metadata writes are flushed to disk to prevent corruption in the event of a system crash. Loss of the registry requires reinstalling *all* software on the system.

11.3 SYSTEM STRUCTURE

In the previous sections we examined Windows Vista as seen by the programmer writing code for user mode. Now we are going to look under the hood to see how the system is organized internally, what the various components do,

and how they interact with each other and with user programs. This is the part of the system seen by the programmer implementing low-level user-mode code, like subsystems and native services, as well as the view of the system provided to device driver writers.

Although there are many books on how to use Windows, there are many fewer on how it works. One of the best places to look for additional information on this topic is *Microsoft Windows Internals*, 4th ed. (Russinovich and Solomon, 2004). This book describes Windows XP, but most of the description is still accurate, since internally, Windows XP and Windows Vista are quite similar.

Additionally Microsoft makes information about the Windows kernel available to faculty and students in universities through the Windows Academic Program. The program gives out source code for most of the Windows Server 2003 kernel, the original NT design documents from Cutler's team, and a large set of presentation materials derived from the Windows Internals book. The Windows Driver Kit also provides a lot of information about the internal workings of the kernel, since device drivers not only use I/O facilities, but also processes, threads, virtual memory, and IPC.

11.3.1 Operating System Structure

As described earlier, the Windows Vista operating system consists of many layers as depicted in Fig. 11-6. In the following sections we will dig into the lowest levels of the operating system: those that run in kernel mode. The central layer is the NTOS kernel itself, which is loaded from *ntoskrnl.exe* when Windows boots. NTOS has two layers, the executive containing most of the services, and a smaller layer which is (also) called the **kernel** and implements the underlying thread scheduling and synchronization abstractions (a kernel within the kernel?), as well as implementing trap handlers, interrupts, and other aspects of how the CPU is managed.

The division of NTOS into kernel and executive is a reflection of NT's VAX/VMS roots. The VMS operating system, which was also designed by Cutler, had four hardware-enforced layers: user, supervisor, executive, and kernel corresponding to the four protection modes provided by the VAX processor architecture. The Intel CPUs also supports four rings of protection, but some of the early target processors for NT did not, so the kernel and executive layers represent a software-enforced abstraction, and the functions that VMS provides in supervisor mode, such as printer spooling, are provided by NT as user-mode services.

The kernel-mode layers of NT are shown in Fig. 11-13. The kernel layer of NTOS is shown above the executive layer because it implements the trap and interrupt mechanisms used to transition from user mode to kernel mode. The uppermost layer in Fig. 11-13 is the system library *ntdll.dll*), which actually runs in user mode. The system library includes a number of support functions for the compiler run-time and low-level libraries, similar to what is in *libc* in UNIX. *ntdll.dll* also

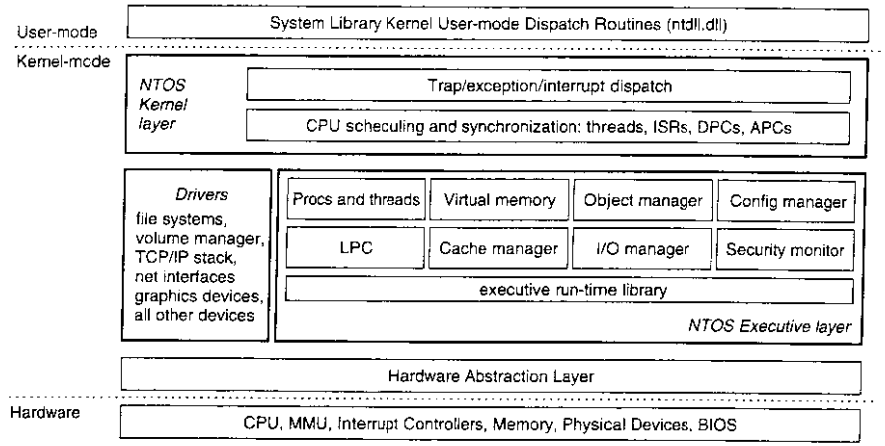

Figure 11-13. Windows kernel-mode organization.

contains special code entry points used by the kernel to initialize threads and dispatch exceptions and user-mode **APCs** (**Asynchronous Procedure Calls**). Because the system library is so integral to the operation of the kernel every user-mode process created by NTOS has *ntdll* mapped at the same fixed address. When NTOS is initializing the system it creates a section object to use when mapping *ntdll*, and also records addresses of the *ntdll* entry points used by the kernel.

Below the NTOS kernel and executive layers there is software called the **HAL** (**Hardware Abstraction Layer**) which abstracts low-level hardware details like access to device registers and DMA operations, and how the BIOS firmware represents configuration information and deals with differences in the CPU support chips, such as various interrupt controllers. The BIOS is available from a number of companies, and integrated into persistent (EEPROM) memory that resides on the computer parentboard.

The other major components of kernel mode are the device drivers. Windows uses device drivers for any kernel-mode facilities which are not part of NTOS or the HAL. This includes file systems and network protocol stacks, and kernel extensions like antivirus and **DRM** (**Digital Rights Management**) software, as well as drivers for managing physical devices, interfacing to hardware buses, and so on.

The I/O and virtual memory components cooperate to load (and unload) device drivers into kernel memory and link them to the NTOS and HAL layers. The I/O manager provides interfaces which allow devices to be discovered, organized, and operated—including arranging to load the appropriate device driver. Much of the configuration information for managing devices and drivers is maintained in the SYSTEM hive of the registry. The plug-and-play sub-component of the I/O

manager maintains information about the hardware detected within the HARDWARE hive, which is a volatile hive maintained in memory rather than on disk as it is completely recreated every time the system boots.

We will now examine the various components of the operating system in a bit more detail.

The Hardware Abstraction Layer

One of the goals of Windows Vista, like the NT-based releases of Windows before it, was to make the operating system portable across hardware platforms. Ideally, to bring up an operating system on a new type of computer system it should be possible to just recompile the operating system with a compiler for the new machine and have it run the first time. Unfortunately, it is not so simple. While many of the components in some layers of the operating system can be largely portable (because they mostly deal with internal data structures and abstractions that support the programming model), other layers must deal with device registers, interrupts, DMA, and other hardware features that differ significantly from machine to machine.

Most of the source code for the NTOS kernel is written in C rather than assembly language (only 2% is assembly on x86, and less than 1% on x64). However, all this C code cannot just be scooped up from an x86 system, plopped down on, say, a SPARC system, recompiled, and rebooted due to the many hardware differences between processor architectures that have nothing to do with the different instruction sets and which cannot be hidden by the compiler. Languages like C make it difficult to abstract away some hardware data structures and parameters, such as the format of page-table entries and the physical memory page sizes and word length, without severe performance penalties. All of these, as well as a slew of hardware-specific optimizations, would have to be manually ported even though they are not written in assembly code.

Hardware details about how memory is organized on large servers, or what hardware synchronization primitives are available, can also have a big impact on higher levels of the system. For example, NT's virtual memory manager and the kernel layer are aware of hardware details related to cache and memory locality. Throughout the system NT uses *compare&swap* synchronization primitives, and it would be difficult to port to a system that does not have them. Finally, there are many dependencies in the system on the ordering of bytes within words. On all the systems NT has ever been ported to, the hardware was set to little-endian mode.

Besides these larger issues of portability, there are also a large number of minor ones even between different parentboards from different manufacturers. Differences in CPU versions affect how synchronization primitives like spin-locks are implemented. There are several families of support chips that create differences in how hardware interrupts are prioritized, how I/O device registers are

accessed, management of DMA transfers, control of the timers and real-time clock, multiprocessor synchronization, working with BIOS facilities such as ACPI (Advanced Configuration and Power Interface), and so on. Microsoft made a serious attempt to hide these types of machine dependencies in a thin layer at the bottom called the HAL, as mentioned earlier. The job of the HAL is to present the rest of the operating system with abstract hardware that hides the specific details of processor version, support chipset, and other configuration variations. These HAL abstractions are presented in the form of machine-independent services (procedure calls and macros) that NTOS and the drivers can use.

By using the HAL services and not addressing the hardware directly, drivers and the kernel require fewer changes when being ported to new processors—and in most all cases can run unmodified on systems with the same processor architecture, despite differences in versions and support chips.

The HAL does not provide abstractions or services for specific I/O devices such as keyboards, mice, disks or for the memory management unit. These facilities are spread throughout the kernel-mode components, and without the HAL the amount of code that would have to be modified when porting would be substantial, even when the actual hardware differences were small. Porting the HAL itself is straightforward because all the machine-dependent code is concentrated in one place and the goals of the port are well defined: implement all of the HAL services. For many releases Microsoft supported a *HAL Development Kit* which allowed system manufacturers to build their own HAL which would allow other kernel components to work on new systems without modification, provided that the hardware changes were not too great.

As an example of what the hardware abstraction layer does, consider the issue of memory-mapped I/O versus I/O ports. Some machines have one and some have the other. How should a driver be programmed: to use memory-mapped I/O or not? Rather than forcing a choice, which would make the driver not portable to a machine that did it the other way, the hardware abstraction layer offers three procedures for driver writers to use for reading the device registers and another three for writing them:

```
uc = READ_PORT_UCHAR(port);        WRITE_PORT_UCHAR(port, uc);
us = READ_PORT_USHORT(port);       WRITE_PORT_USHORT(port, us);
ul = READ_PORT_ULONG(port);        WRITE_PORT_LONG(port, ul);
```

These procedures read and write unsigned 8-, 16-, and 32-bit integers, respectively, to the specified port. It is up to the hardware abstraction layer to decide whether memory-mapped I/O is needed here. In this way, a driver can be moved without modification between machines that differ in the way the device registers are implemented.

Drivers frequently need to access specific I/O devices for various purposes. At the hardware level, a device has one or more addresses on a certain bus. Since modern computers often have multiple buses (ISA, PCI, PCI-X, USB, 1394, etc.),

it can happen that more than one device may have the same address on different buses, so some way is needed to distinguish them. The HAL provides a service for identifying devices by mapping bus-relative device addresses onto system-wide logical addresses. In this way, drivers do not have to keep track of which device is connected to which bus. This mechanism also shields higher layers from properties of alternative bus structures and addressing conventions.

Interrupts have a similar problem—they are also bus dependent. Here, too, the HAL provides services to name interrupts in a system-wide way and also provides services to allow drivers to attach interrupt service routines to interrupts in a portable way, without having to know anything about which interrupt vector is for which bus. Interrupt request level management is also handled in the HAL.

Another HAL service is setting up and managing DMA transfers in a device-independent way. Both the system-wide DMA engine and DMA engines on specific I/O cards can be handled. Devices are referred to by their logical addresses. The HAL implements software scatter/gather (writing or reading from noncontiguous blocks of physical memory).

The HAL also manages clocks and timers in a portable way. Time is kept track of in units of 100 nanoseconds starting at 1 January 1601, which is the first date in the previous quadricentury, which simplifies leap year computations. (Quick Quiz: Was 1800 a leap year? Quick Answer: No.) The time services decouple the drivers from the actual frequencies at which the clocks run.

Kernel components sometimes need to synchronize at a very low level, especially to prevent race conditions in multiprocessor systems. The HAL provides primitives to manage this synchronization, such as spin locks, in which one CPU simply waits for a resource held by another CPU to be released, particularly in situations where the resource is typically only held for a few machine instructions.

Finally, after the system has been booted, the HAL talks to the BIOS and inspects the system configuration to find out which buses and I/O devices the system contains and how they have been configured. This information is then put into the registry. A summary of some of the things the HAL does is given in Fig. 11-14.

The Kernel Layer

Above the hardware abstraction layer is NTOS, consisting of two layers: the **kernel** and the **executive**. "Kernel" is a confusing term in Windows. It can refer to all the code that runs in the processor's kernel mode. It can also refer to the *ntoskrnl.exe* file which contains NTOS, the core of the Windows operating system. Or it can refer to the kernel layer within NTOS, which is how we use it in this section. It is even used to name the user-mode Win32 library that provides the wrappers for the native system calls: *kernel32.dll*.

In the Windows operating system the kernel layer, ilustrated above the executive layer in Fig. 11-13, provides a set of abstractions for managing the CPU. The

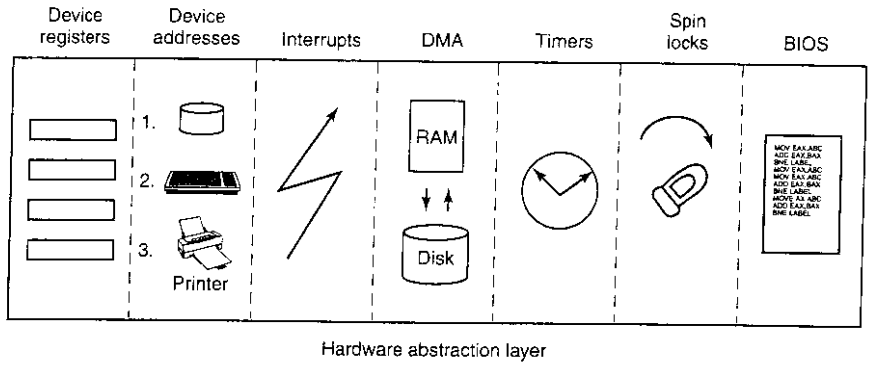

Figure 11-14. Some of the hardware functions the HAL manages.

most central abstraction is threads, but the kernel also implements exception handling, traps, and several kinds of interrupts. Creating and destroying the data structures which support threading is implemented in the executive layer. The kernel layer is responsible for scheduling and synchronization of threads. Having support for threads in a separate layer allows the executive layer to be implemented using the same preemptive multithreading model used to write concurrent code in user mode, though the synchronization primitives in the executive are much more specialized.

The kernel's thread scheduler is responsible for determining which thread is executing on each CPU in the system. Each thread executes until a timer interrupt signals that it is time to switch to another thread (quantum expired), or until the thread needs to wait for something to happen, such as an I/O to complete or for a lock to be released, or a higher-priority thread becomes runnable and needs the CPU. When switching from one thread to another, the scheduler runs on the CPU and ensures that the registers and other hardware state have been saved. The scheduler then selects another thread to run on the CPU and restores the state that was previously saved from the last time that thread ran.

If the next thread to be run is in a different address space (i.e., process) than the thread being switched from, the scheduler must also change address spaces. The details of the scheduling algorithm itself will be discussed later in this chapter when we come to processes and threads.

In addition to providing a higher-level abstraction of the hardware and handling thread switches, the kernel layer also has another key function: providing low-level support for two classes of synchronization mechanisms: control objects and dispatcher objects. **Control objects** are the data structures that the kernel layer provides as abstractions to the executive layer for managing the CPU. They are allocated by the executive but they are manipulated with routines provided by

the kernel layer. **Dispatcher objects** are the class of ordinary executive objects that use a common data structure for synchronization.

Deferred Procedure Calls

Control objects include primitive objects for threads, interrupts, timers, synchronization, profiling, and two special objects for implementing DPCs and APCs. **DPC (Deferred Procedure Call)** objects are used to reduce the time taken to execute **ISRs (Interrupt Service Routines)** in response to an interrupt from a particular device.

The system hardware assigns a hardware priority level to interrupts. The CPU also associates a priority level with the work it is performing. The CPU only responds to interrupts at a higher priority level than it is currently using. Normal priority levels, including the priority level of all user-mode work, is 0. Device interrupts occur at priority 3 or higher, and the ISR for a device interrupt normally executes at the same priority level as the interrupt in order to keep other less important interrupts from occurring while it is processing a more important one.

If an ISR executes too long, the servicing of lower-priority interrupts will be delayed, perhaps causing data to be lost or slowing the I/O throughput of the system. Multiple ISRs can be in progress at any one time, with each successive ISR being due to interrupts at higher and higher priority levels.

To reduce the time spent processing ISRs, only the critical operations are performed, such as capturing the result of an I/O operation and reinitializing the device. Further processing of the interrupt is deferred until the CPU priority level is lowered and no longer blocking the servicing of other interrupts. The DPC object is used to represent the further work to be done and the ISR calls the kernel layer to queue the DPC to the list of DPCs for a particular processor. If the DPC is the first on the list, the kernel registers a special request with the hardware to interrupt the CPU at priority 2 (which NT calls DISPATCH level). When the last of any executing ISRs complete, the interrupt level of the processor will drop back below 2, and that will unblock the interrupt for DPC processing. The ISR for the DPC interrupt will process each of the DPC objects that the kernel had queued.

The technique of using software interrupts to defer interrupt processing is a well-established method of reducing ISR latency. UNIX and other systems started using deferred processing in the 1970s to deal with the slow hardware and limited buffering of serial connections to terminals. The ISR would deal with fetching characters from the hardware and queuing them. After all higher-level interrupt processing was completed, a software interrupt would run a low-priority ISR to do character processing, such as implementing backspace by sending control characters to the terminal to erase the last character displayed and move the cursor backward.

A similar example in Windows today is the keyboard device. After a key is struck, the keyboard ISR reads the key code from a register and then reenables the keyboard interrupt, but does not do further processing of the key immediately. Instead it uses a DPC to queue the processing of the key code until all outstanding device interrupts have been processed.

Because DPCs run at level 2 they do not keep device ISRs from executing, but they do prevent any threads from running until all the queued DPCs complete and the CPU priority level is lowered below 2. Device drivers and the system itself must take care not to run either ISRs or DPCs for too long. Because threads are not allowed to execute, ISRs and DPCs can make the system appear sluggish, and produce glitches when playing music by stalling the threads writing the music buffer to the sound device. Another common use of DPCs is running routines in response to a timer interrupt. To avoid blocking threads, timer events which need to run for an extended time should queue requests to the pool of worker threads the kernel maintains for background activities. These threads have scheduling priority 12, 13, or 15. As we will see in the section on thread scheduling, these priorities mean that work items will execute ahead of most threads, but not interfere with *real-time* threads.

Asynchronous Procedure Calls

The other special kernel control object is the APC (asynchronous procedure call) object. APCs are like DPCs in that they defer processing of a system routine, but unlike DPCs, which operate in the context of particular CPUs, APCs execute in the context of a specific thread. When processing a key press, it does not matter which context the DPC runs in because a DPC is simply another part of interrupt processing, and interrupts only need to manage the physical device and perform thread-independent operations such as recording the data in a buffer in kernel space.

The DPC routine runs in the context of whatever thread happened to be running when the original interrupt occurred. It calls into the I/O system to report that the I/O operation has been completed, and the I/O system queues an APC to run in the context of the thread making the original I/O request, where it can access the user-mode address space of the thread that will process the input.

At the next convenient time the kernel layer delivers the APC to the thread and schedules the thread to run. An APC is designed to look like an unexpected procedure call, somewhat similar to signal handlers in UNIX. The kernel-mode APC for completing I/O executes in the context of the thread that initiated the I/O, but in kernel mode. This gives the APC access to both the kernel-mode buffer as well as all of the user-mode address space belonging to the process containing the thread. *When* an APC is delivered depends on what the thread is already doing, and even what type of system. In a multiprocessor system the thread receiving the APC may begin executing even before the DPC finishes running.

User-mode APCs can also be used to deliver notification of I/O completion in user mode to the thread that initiated the I/O. User-mode APCs invoke a user-

mode procedure designated by the application, but only when the target thread has blocked in the kernel and is marked as willing to accept APCs. The kernel interrupts the thread from waiting and returns to user mode, but with the user-mode stack and registers modified to run the APC dispatch routine in the *ntdll.dll* system library. The APC dispatch routine invokes the user-mode routine that the application has associated with the I/O operation. Besides specifying user-mode APCs as a means of executing code when I/Os complete, the Win32 API QueueUserAPC allows APCs to be used for arbitrary purposes.

The executive layer also uses APCs for operations other than I/O completion. Because the APC mechanism is carefully designed to deliver APCs only when it is safe to do so, it can be used to safely terminate threads. If it is not a good time to terminate the thread, the thread will have declared that it was entering a critical region and defer deliveries of APCs until it leaves. Kernel threads mark themselves as entering critical regions to defer APCs when before acquiring locks or other resources, so that they cannot be terminated while still holding the resource.

Dispatcher Objects

Another kind of synchronization object is the **dispatcher object**. This is any of the ordinary kernel-mode objects (the kind that users can refer to with handles) that contain a data structure called a **dispatcher_header**, shown in Fig. 11-15.

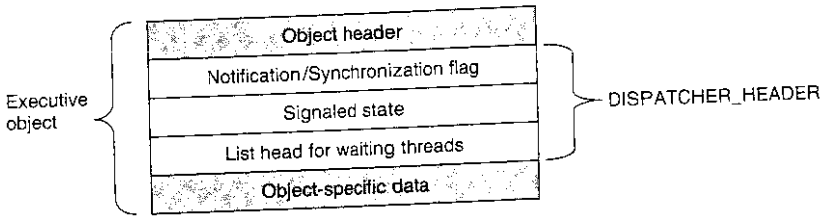

Figure 11-15. *dispatcher_header* data structure embedded in many executive objects (*dispatcher objects*).

These include semaphores, mutexes, events, waitable timers, and other objects that threads can wait on to synchronize execution with other threads. They also include objects representing open files, processes, threads, and IPC ports. The dispatcher data structure contains a flag representing the signaled state of the object, and a queue of threads waiting for the object to be signaled.

Synchronization primitives, like semaphores, are natural dispatcher objects. Also timers, files, ports, threads, and processes use the dispatcher object mechanisms for notifications. When a timer fires, I/O completes on a file, data are available on a port, or a thread or process terminates, the associated dispatcher object is signaled, waking all threads waiting for that event.

Since Windows uses a single unified mechanism for synchronization with kernel-mode objects, specialized APIs, such as wait3 for waiting for child processes in UNIX, are not needed to wait for events. Often threads want to wait for multiple events at once. In UNIX a process can wait for data to be available on any of 64 network sockets using the select system call. In Windows there is a similar API **WaitForMultipleObjects**, but it allows for a thread to wait on any type of dispatcher object for which it has a handle. Up to 64 handles can be specified to WaitForMultipleObjects, as well as an optional timeout value. The thread becomes ready to run whenever any of the events associated with the handles is signaled, or the timeout occurs.

There are actually two different procedures the kernel uses for making the threads waiting on a dispatcher object runnable. Signaling a **notification object** will make every waiting thread runnable. **Synchronization objects** only make the first waiting thread runnable and are used for dispatcher objects that implement locking primitives, like mutexes. When a thread waiting for a lock begins running again, the first thing it does is retry acquiring the lock again. If only one thread can hold the lock at a time, all the other threads made runnable might immediately block, incurring lots of unnecessary context switching. The difference between dispatcher objects using synchronization versus notification is a flag in the dispatcher_header structure.

As a little aside, mutexes in Windows are called "mutants" in the code because they were required to implement the OS/2 semantics of not automatically unlocking themselves when a thread holding one exited, something Cutler considered bizarre.

The Executive Layer

As shown in Fig. 11-13, below the kernel layer of NTOS there is the **executive**. The executive layer is written in C, is mostly architecture independent (the memory manager being a notable exception), and has been ported to new processors with only modest effort (MIPS, x86, PowerPC, Alpha, IA64, and x64). The executive contains a number of different components, all of which run using the control abstractions provided by the kernel layer.

Each component is divided into internal and external data structures and interfaces. The internal aspects of each component are hidden and only used within the component itself, while the external are available to all the other components within the executive. A subset of the external interfaces are exported from the *ntoskrnl.exe* executable and device drivers can link to them as if the executive were a library. Microsoft calls many of the executive components "managers", because each is charge of managing some aspect of the operating services, such as I/O, memory, processes, objects, etc.

As with most operating systems, much of the functionality in the Windows executive is like library code, except that it runs in kernel-mode so that its data

structures can be shared and protected from access by user-mode code, and so it can access privileged hardware state, such as the MMU control registers. But otherwise the executive is simply executing OS functions on behalf of its caller, and thus runs in the thread of its caller.

When any of the executive functions block waiting to synchronize with other threads, the user-mode thread is blocked too. This makes sense when working on behalf of a particular user-mode thread, but can be unfair when doing work related to common housekeeping tasks. To avoid hijacking the current thread when the executive determines that some housekeeping is needed, a number of kernel-mode threads are created when the system boots and dedicated to specific tasks, such as making sure that modified pages get written to disk.

For predictable, low-frequency tasks, there is a thread that runs once a second and has a laundry list of items to handle. For less predictable work there is the pool of high-priority worker threads mentioned earlier which can be used to run bounded tasks by queuing a request and signaling the synchronization event that the worker threads are waiting on.

The **object manager** manages most of the interesting kernel-mode objects used in the executive layer. These include processes, threads, files, semaphores, I/O devices and drivers, timers, and many others. As described previously, kernel-mode objects are really just data structures allocated and used by the kernel. In Windows, kernel data structures have enough in common that it is very useful to manage many of them in a unified facility.

The facilities provided by the object manager include managing the allocation and freeing of memory for objects, quota accounting, supporting access to objects using handles, maintaining reference counts for kernel-mode pointer references as well as handle references, giving objects names in the NT namespace, and providing an extensible mechanism for managing the lifecycle for each object. Kernel data structures which need some of these facilities are managed by the object manager. Other data structures, such as the control objects used by the kernel layer, or objects that are just extensions of kernel-mode objects, are not managed by them.

Object manager objects each have a type which is used to specify how the lifecycle of objects of that type is to be managed. These are not types in the object-oriented sense, but are simply a collection of parameters specified when the object type is created. To create a new type, an executive component simply calls an object manager API to create a new type. Objects are so central to the functioning of Windows that the object manager will be discussed in more detail in the next section.

The **I/O manager** provides the framework for implementing I/O device drivers and provides a number of executive services specific to configuring, accessing, and performing operations on devices. In Windows, device drivers not only manage physical devices but they also provide extensibility to the operating system. Many functions that are compiled into the kernel on other systems are

dynamically loaded and linked by the kernel on Windows, including network protocol stacks and file systems.

Recent versions of Windows have a lot more support for running device drivers in user mode, and this is the preferred model for new device drivers. There are hundreds of thousands of different device drivers for Windows Vista working with more than a million distinct devices. This represents a lot of code to get correct. It is much better if bugs cause a device to become inaccessible by crashing in a user-mode process rather than causing the system to bugcheck. Bugs in kernel-mode device drivers are the major source of the dreaded **BSOD** (**Blue Screen Of Death**) where Windows detects a fatal error within kernel-mode and shuts down or reboots the system. BSOD's are comparable to kernel panics on UNIX systems.

In essence, Microsoft has now officially recognized what researchers in the area of microkernels such as MINIX 3 and L4 have known for years: the more code there is in the kernel, the more bugs there are in the kernel. Since device drivers make up something like 70% of the code in the kernel, the more drivers that can be moved into user-mode processes, where a bug will only trigger the failure of a single driver (rather than bringing down the entire system) the better. The trend of moving code from the kernel to user-mode processes is expected to accelerate in the coming years.

The I/O manager also includes the plug-and-play and power management facilities. **Plug-and-play** comes into action when new devices are detected on the system. The plug-and-play subcomponent is first notified. It works with a service, the user-mode plug-and-play manager, to find the appropriate device driver and load it into the system. Finding the right device driver is not always easy, and sometimes depends on sophisticated matching of the specific hardware device version to a particular version of the drivers. Sometimes a single device supports a standard interface which is supported by multiple different drivers, written by different companies.

Power management reduces power consumption when possible, extending battery life on notebooks, and saving energy on desktops and servers. Getting power management correct can be challenging, as there are many subtle dependencies between devices and the buses that connect them to the CPU and memory. Power consumption is not just affected by what devices are powered-on, but also by the clock rate of the CPU, which is also controlled by the power manager.

We will study I/O further in Sec. 11.7 and the most important NT file system, NTFS, in Sec. 11.8.

The **process manager** manages the creation and termination of processes and threads, including establishing the policies and parameters which govern them. But the operational aspects of threads are determined by the kernel layer, which controls scheduling and synchronization of threads, as well as their interaction with the control objects, like APCs. Processes contain threads, an address space, and a handle table containing the handles the process can use to refer to kernel-

mode objects. Processes also include information needed by the scheduler for switching between address spaces and managing process-specific hardware information (such as segment descriptors). We will study process and thread management in Sec. 11.4.

The executive **memory manager** implements the demand-paged virtual memory architecture. It manages the mapping of virtual pages onto physical page frames, the management of the available physical frames, and management of the pagefile on disk used to back private instances of virtual pages that are no longer loaded in memory. The memory manager also provides special facilities for large server applications such as databases and programming language run-time components such as garbage collectors. We will study memory management later in this chapter, in Sec. 11.5.

The **cache manager** optimizes the performance of I/O to the file system by maintaining a cache of file system pages in the kernel virtual address space. The cache manager uses virtually addressed caching, that is, organizing cached pages in terms of their location in their files. This differs from physical block caching, as in UNIX, where the system maintains a cache of the physically addressed blocks of the raw disk volume.

Cache management is implemented using memory mapping of the files. The actual caching is performed by the memory manager. The cache manager need only be concerned with deciding what parts of what files to cache, ensuring that cached data is flushed to disk in a timely fashion, and managing the kernel virtual addresses used to map the cached file pages. If a page needed for I/O to a file is not available in the cache, the page will be faulted in using the memory manager. We will study the cache manager in Sec. 11.6.

The **security reference monitor** enforces Windows' elaborate security mechanisms, which support the international standards for computer security called **Common Criteria**, an evolution of United States Department of Defense Orange Book security requirements. These standards specify a large number of rules that a conforming system must meet, such as authenticated login, auditing, zeroing of allocated memory, and many more. One of the rules requires that all access checks be implemented by a single module within the system. In Windows this module is the security reference monitor in the kernel. We will study the security system in more detail in Sec. 11.9.

The executive contains a number of other components that we will briefly describe. The **configuration manager** is the executive component which implements the registry, as described earlier. The registry contains configuration data for the system in file system files called *hives*. The most critical hive is the *SYSTEM* hive which is loaded into memory at boot time. Only after the executive layer has successfully initialized its key components, including the I/O drivers that talk to the system disk, is the in-memory copy of the hive reassociated with the copy in the file system. Thus if something bad happens while trying to boot the system, the on-disk copy is much less likely to be corrupted.

The LPC component provides for a highly efficient inter-process communication used between processes running on the same system. It is one of the data transports used by the standards-based remote-procedure call (RPC) facility to implement the client/server style of computing. RPC also uses named pipes and TCP/IP as transports.

LPC was substantially enhanced in Windows Vista (it is now called **ALPC**, for **Advanced LPC**) to provide support for new features in RPC, including RPC from kernel-mode components, like drivers. LPC was a critical component in the original design of NT because it is used by the subsystem layer to implement communication between library stub routines that run in each process and the subsystem process which implements the facilities common to a particular operating system personality, such as Win32 or POSIX.

In Windows NT 4.0 much of the code related to the Win32 graphical interface was moved into the kernel because the then-current hardware could not provide the required performance. This code previously resided in the *csrss.exe* subsystem process which implemented the Win32 interfaces. The kernel-based GUI code resides in a special kernel-driver, *win32k.sys*. This change was expected to improve Win32 performance because the extra user-mode/kernel-mode transitions and the cost of switching address spaces to implement communication via LPC was eliminated. But it has not been as successful as expected because the requirements on code running in the kernel are very strict, and the additional overhead of running in kernel-mode offsets some of the gains from reducing switching costs.

The Device Drivers

The final part of Fig. 11-13 consists of the **device drivers**. Device drivers in Windows are dynamic link libraries which are loaded by the NTOS executive. Though they are primarily used to implement the drivers for specific hardware, such as physical devices and I/O buses, the device driver mechanism is also used as the general extensibility mechanism for kernel mode. As described above, much of the Win32 subsystem is loaded as a driver.

The I/O manager organizes a data flow path for each instance of a device, as shown in Fig. 11-16. This path is called a **device stack** and consists of private instances of kernel **device objects** allocated for the path. Each device object in the device stack is linked to a particular driver object, which contains the table of routines to use for the I/O request packets that flow through the device stack. In some cases the devices in the stack represent drivers whose sole purpose is to **filter** I/O operations aimed at a particular device, bus, or network driver. Filtering is used for a number of reasons. Sometimes preprocessing or post-processing I/O operations results in a cleaner architecture, while other times it is just pragmatic because the sources or rights to modify a driver are not available and filtering is

used to work around it. Filters can also implement completely new functionality, such as turning disks into partitions or multiple disks into RAID volumes.

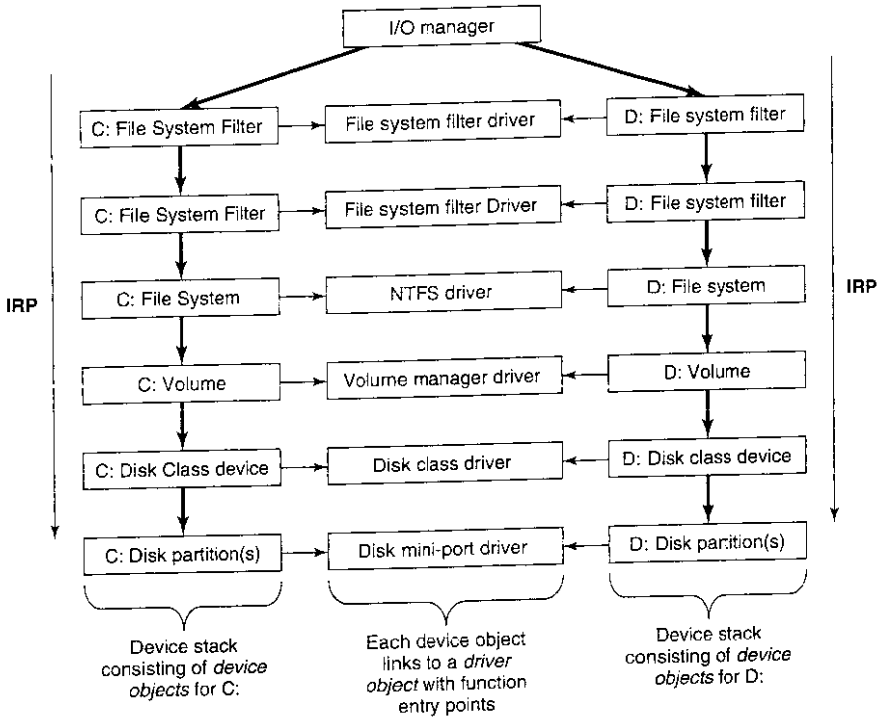

Figure 11-16. Simplified depiction of device stacks for two NTFS file volumes. The I/O request packet is passed from down the stack. The appropriate routines from the associated drivers are called at each level in the stack. The device stacks themselves consist of device objects allocated specifically to each stack.

The file systems are loaded as drivers. Each instance of a volume for a file system has a device object created as part of the device stack for that volume. This device object will be linked to the driver object for the file system appropriate to the volume's formatting. Special filter drivers, called **file system filter drivers**, can insert device objects before the file system device object to apply functionality to the I/O requests being sent to each volume, such as inspecting data read or written for viruses.

The network protocols, such as Windows Vista's integrated IPv4/IPv6 TCP/IP implementation, are also loaded as drivers using the I/O model. For compatibility with the older MS-DOS-based Windows, the TCP/IP driver implements a special protocol for talking to network interfaces on top of the Windows I/O model.

There are other drivers that also implement such arrangements, which Windows calls **mini-ports**. The shared functionality is in a **class driver**. For example, common functionality for SCSI or IDE disks or USB devices is supplied by a class driver, which mini-port drivers for each particular type of such devices link to as a library.

We will not discuss any particular device driver in this chapter, but will provide more detail about how the I/O manager interacts with device drivers in Sec. 11.7.

11.3.2 Booting Windows Vista

Getting an operating system to run requires several steps. When a computer is turned on, the CPU is initialized by the hardware, and then set to start executing a program in memory. But the only available code is in some form of nonvolatile CMOS memory that is initialized by the computer manufacturer (and sometimes updated by the user, in a process called **flashing**). On most PC's this initial program is the BIOS (Basic Input/Output System) which knows how to talk to the standard types of devices found on a PC. The BIOS brings up Windows Vista by first loading small bootstrap programs found at the beginning of the disk drive partitions.

The bootstrap programs know how to read enough information off a file system volume to find the standalone Windows *BootMgr* program in the root directory. *BootMgr* determines if the system had previously been hibernated or was in stand-by mode (special power-saving modes that allow the system to turn back on without booting). If so, *BootMgr* loads and executes *WinResume.exe*. Otherwise it loads and executes *WinLoad.exe* to perform a fresh boot. *WinLoad* loads the boot components of the system into memory: the kernel/executive (normally *ntoskrnl.exe*), the HAL (*hal.dll*), the file containing the SYSTEM hive, the *Win32k.sys* driver containing the kernel-mode parts of the Win32 subsystem, as well as images of any other drivers that are listed in the SYSTEM hive as **boot drivers**—meaning they are needed when the system first boots.

Once the Windows boot components are loaded into memory, control is given to low-level code in NTOS which proceeds to initialize the HAL, kernel and executive layers, link in the driver images, and access/update configuration data in the SYSTEM hive. After all the kernel-mode components are initialized, the first user-mode process is created using for running the *smss.exe* program (which is like */etc/init* in UNIX systems).

The Windows boot programs have logic to deal with common problems users encounter when booting the system fails. Sometimes installation of a bad device driver, or running a program like *regedit* (which can corrupt the SYSTEM hive), will prevent the system from booting normally. There is support for ignoring recent changes and booting to the *last known good* configuration of the system. Other boot options include **safe-boot** which turns off many optional drivers and

the **recovery console**, which fires up a *cmd.exe* command-line window, providing an experience similar to single-user mode in UNIX.

Another common problem for users has been that occasionally some Windows systems appear to be very flaky, with frequent (seemingly random) crashes of both the system and applications. Data taken from Microsoft's On-line Crash Analysis program provided evidence that many of these crashes were due to bad physical memory, so the boot process in Windows Vista provides the option of running an extensive memory diagnostic. Perhaps future PC hardware will commonly support ECC (or maybe parity) for memory, but most of the desktop and notebook systems today are vulnerable to even single-bit errors in the billions of bits of memory they contain.

11.3.3 Implementation of the Object Manager

The object manager is probably the single most important component in the Windows executive, which is why we have already introduced many of its concepts. As described earlier, it provides a uniform and consistent interface for managing system resources and data structures, such as open files, processes, threads, memory sections, timers, devices, drivers, and semaphores. Even more specialized objects representing things like kernel transactions, profiles, security tokens, and Win32 desktops are managed by the object manager. Device objects link together the descriptions of the I/O system, including providing the link between the NT namespace and file system volumes. The configuration manager uses an object of type **Key** to link in the registry hives. The object manager itself has objects it uses to manage the NT namespace and implement objects using a common facility. These are directory, symbolic link, and object-type objects.

The uniformity provided by the object manager has various facets. All these objects use the same mechanism for how they are created, destroyed, and accounted for in the quota system. They can all be accessed from user-mode processes using handles. There is a unified convention for managing pointer references to objects from within the kernel. Objects can be given names in the NT namespace (which is managed by the object manager). Dispatcher objects (objects that begin with the common data structure for signaling events) can use common synchronization and notification interfaces, like WaitForMultipleObjects. There is the common security system with ACLs enforced on objects opened by name, and access checks on each use of a handle. There are even facilities to help kernel-mode developers debug problems by tracing the use of objects.

A key to understanding objects is to realize that an (executive) object is just a data structure in the virtual memory accessible to kernel mode. These data structures are commonly used to represent more abstract concepts. As examples, executive file objects are created for each instance of a file system file that has been opened. Process objects are created to represent each process.

A consequence of the fact that objects are just kernel data structures is that when the system is rebooted (or crashes) all objects are lost. When the system boots, there are no objects present at all, not even the object type descriptors. All object types, and the objects themselves, have to be created dynamically by other components of the executive layer by calling the interfaces provided by the object manager. When objects are created and a name is specified, they can later be referenced through the NT namespace. So building up the objects as the system boots also builds the NT namespace.

Objects have a structure, as shown in Fig. 11-17. Each object contains a header with certain information common to all objects of all types. The fields in this header include the object's name, the object directory in which it lives in the NT namespace, and a pointer to a security descriptor representing the ACL for the object.

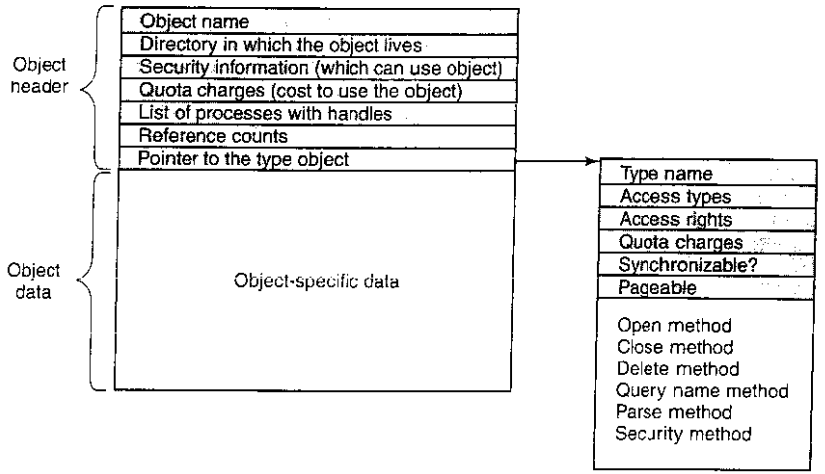

Figure 11-17. The structure of an executive object managed by the object manager

The memory allocated for objects comes from one of two heaps (or pools) of memory maintained by the executive layer. There are (malloc-like) utility functions in the executive that allow kernel-mode components to allocate either pageable kernel memory or nonpageable kernel memory. Nonpageable memory is required for any data structure or kernel-mode object that might need to be accessed from a CPU priority level of 2 or more. This includes ISRs and DPCs (but not APCs), and the thread scheduler itself. The pagefault handle also requires its data structures to be allocated from nonpageable kernel memory to avoid recursion.

Most allocations from the kernel heap manager are achieved using per-processor lookaside lists which contain LIFO lists of allocations the same size. These

LIFOs are optimized for lock-free operation, improving the performance and scalability of the system.

Each object header contains a quota charge field, which is the charge levied against a process for opening the object. Quotas are used to keep a user from using too many system resources. There are separate limits for nonpageable kernel memory (which requires allocation of both physical memory and kernel virtual addresses) and pageable kernel memory (which uses up kernel virtual addresses). When the cumulative charges for either memory type hit the quota limit, allocations for that process fail due to insufficient resources. Quotas also are used by the memory manager to control working-set size, and the thread manager to limit the rate of CPU usage.

Both physical memory and kernel virtual addresses are valuable resources. When an object is no longer needed, it should be removed and its memory and addresses reclaimed. But if an object is reclaimed while it is still in use, then the memory may be allocated to another object, and then the data structures are likely to become corrupted. It is easy for this to happen in the Windows executive layer because it is highly multithreaded, and implements many asynchronous operations (functions that return to their caller before completing work on the data structures passed to them).

To avoid freeing objects prematurely due to race conditions, the object manager implements a reference counting mechanism, and the concept of a **referenced pointer**. A referenced pointer is needed to access an object whenever that object is in danger of being deleted. Depending on the conventions regarding each particular object type, there are only certain times when an object might be deleted by another thread. At other times the use of locks, dependencies between data structures, and even the fact that no other thread has a pointer to an object are sufficient to keep the object from being prematurely deleted.

Handles

User-mode references to kernel-mode objects cannot use pointers because they are too difficult to validate. Instead kernel-mode objects must be named in some other way so the user code can refer to them. Windows uses **handles** to refer to kernel-mode objects. Handles are opaque values which are converted by the object manager into references to the specific kernel-mode data structure representing an object. Figure 11-18 shows the handle table data structure used to translate handles into object pointers. The handle table is expandable by adding extra layers of indirection. Each process has its own table, including the system process which contains all the kernel threads not associated with a user-mode process.

Figure 11-19 shows a handle table with two extra levels of indirection, the maximum supported. It is sometimes convenient for code executing in kernel-mode to be able to use handles rather than referenced pointers. These are called

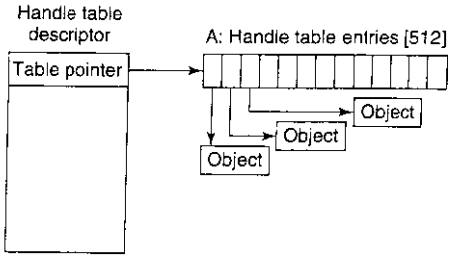

Figure 11-18. Handle table data structures for a minimal table using a single page for up to 512 handles.

kernel handles and are specially encoded so that they can be distinguished from user-mode handles. Kernel handles are kept in the system processes' handle table, and cannot be accessed from user mode. Just as most of the kernel virtual address space is shared across all processes, the system handle table is shared by all kernel components, no matter what the current user-mode process is.

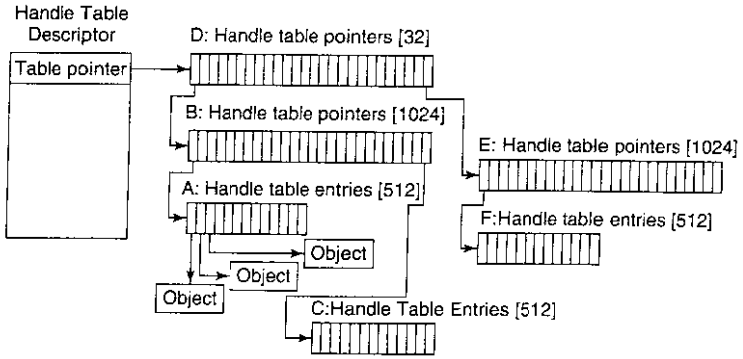

Figure 11-19. Handle table data structures for a maximal table of up to 16 million handles.

Users can create new objects or open existing objects by making Win32 calls such as CreateSemaphore or OpenSemaphore. These are calls to library procedures that ultimately result in the appropriate system calls being made. The result of any successful call that creates or opens an object is a 64-bit handle table entry that is stored in the process' private handle table in kernel memory. The 32-bit index of the handle's logical position in the table is returned to the user to use on subsequent calls. The 64-bit handle table entry in the kernel contains two 32-bit words. One word contains a 29-bit pointer to the object's header. The low-order 3 bits are used as flags (e.g., whether the handle is inherited by processes it

creates). These 3 bits are masked off before the pointer is followed. The other word contains a 32-bit rights mask. It is needed because permissions checking is done only at the time the object is created or opened. If a process has only read permission to an object, all the other rights bits in the mask will be 0s, giving the operating system the ability to reject any operation on the object other than reads.

The Object Name Space

Processes can share objects by having one process duplicate a handle to the object into the others. But this requires that the duplicating process have handles to the other processes, and is thus impractical in many situations, such as when the processes sharing an object are unrelated, or are protected from each other. In other cases it is important that objects persist even when they are not being used by any process, such as device objects representing physical devices, or mounted volumes, or the objects used to implement the object manager and the NT namespace itself. To address general sharing and persistence requirements, the object manager allows arbitrary objects to be given names in the NT namespace when they are created. However, it is up to the executive component that manipulates objects of a particular type to provide interfaces that support use of the object manager's naming facilities.

The NT namespace is hierarchical, with the object manager implementing directories and symbolic links. The namespace is also extensible, allowing any object type to specify extensions of the namespace by supplying a routine named **Parse** routine. The *Parse* routine is one of the procedures that can be supplied for each object type when it is created, as shown in Fig. 11-20.

Procedure	When called	Notes
Open	For every new handle	Rarely used
Parse	For object types that extend the namespace	Used for files and registry keys
Close	At last handle close	Clean up visible side effects
Delete	At last pointer dereference	Object is about to be deleted
Security	Get or set object's security descriptor	Protection
QueryName	Get object's name	Rarely used outside kernel

Figure 11-20. The object procedures supplied when specifying a new object type.

The *Open* procedure is rarely used because the default object manager behavior is usually what is needed and so the procedure is specified as NULL for almost all object types.

The *Close* and *Delete* procedures represent different phases of being done with an object. When the last handle for an object is closed, there may be actions

necessary to clean up the state, which are performed by the *Close* procedure. When the final pointer reference is removed from the object, the *Delete* procedure is called so that the object can be prepared to be deleted and have its memory reused. With file objects, both of these procedures are implemented as callbacks into the I/O manager, which is the component that declared the file object type. The object manager operations result in corresponding I/O operations that are sent down the device stack associated with the file object, and the file system does most of the work.

The *Parse* procedure is used to open or create objects, like files and registry keys, that extend the NT namespace. When the object manager is attempting to open an object by name and encounters a leaf node in the part of the namespace it manages, it checks to see if the type for the leaf node object has specified a *Parse* procedure. If so, it invokes the procedure, passing it any unused part of the pathname. Again using file objects as an example, the leaf node is a device object representing a particular file system volume. The *Parse* procedure is implemented by the I/O manager, and results in an I/O operation to the file system to fill in a file object to refer to an open instance of the file that the pathname refers to on the volume. We will explore this particular example step-by-step below.

The *QueryName* procedure is used to look up the name associated with an object. The *Security* procedure is used to get, set, or delete the security descriptors on an object. For most objects types this procedure is supplied as a standard entry point in the executive's Security Reference Monitor component.

Note that the procedures in Fig. 11-20 do not perform the most interesting operations for each type of object. Rather, these procedures supply the callback functions the object manager needs to correctly implement functions such as providing access to objects and cleaning up objects when done with them. Apart from these callbacks, the object manager also provides a set of generic object routines for operations like creating objects and object types, duplicating handles, getting a referenced pointer from a handle or name, and adding and subtracting reference counts to the object header.

The interesting operations on objects are the native NT API system calls, like those shown in Fig. 11-9, such as NtCreateProcess, NtCreateFile, or NtClose (the generic function that closes all types of handles).

Although the object name space is crucial to the entire operation of the system, few people know that it even exists because it is not visible to users without special viewing tools. One such viewing tool is *winobj*, available for free at *www.microsoft.com/technet/sysinternals*. When run, this tool depicts an object name space that typically contains the object directories listed in Fig. 11-21 as well as a few others.

The strangely named directory *\??* contains the names of all the MS-DOS-style device names, such as *A:* for the floppy disk and *C:* for the first hard disk. These names are actually symbolic links to the directory *\Device* where the device objects live. The name *\??* was chosen to make it alphabetically first so as to

Directory	Contents
??	Starting place for looking up MS-DOS devices like C:
DosDevices	Official name of ??, but really just a symbolic link to ??
Device	All discovered I/O devices
Driver	Objects corresponding to each loaded device driver
ObjectTypes	The type objects such as those listed in Fig. 11-22
Windows	Objects for sending messages to all the Win32 GUI windows
BaseNamedObjects	User-created Win32 objects such as semaphores, mutexes, etc.
Arcname	Partition names discovered by the boot loader
NLS	National Language Support objects
FileSystem	File system driver objects and file system recognizer objects
Security	Objects belonging to the security system
KnownDLLs	Key shared libraries that are opened early and held open

Figure 11-21. Some typical directories in the object name space.

speed up lookup of all path names beginning with a drive letter. The contents of the other object directories should be self explanatory.

As described above, the object manager keeps a separate handle count in every object. This count is never larger than the referenced pointer count because each valid handle has a referenced pointer to the object in its handle table entry. The reason for the separate handle count is that many types of objects may need to have their state cleaned up when the last user-mode reference disappears, even though they are not yet ready to have their memory deleted.

One example is file objects, which represent an instance of an opened file. In Windows files can be opened for exclusive access. When the last handle for a file object is closed it is important to delete the exclusive access at that point rather than wait for any incidental kernel references to eventually go away (e.g., after the last flush of data from memory). Otherwise closing and reopening a file from user-mode may not work as expected because the file still appears to be in use.

Though the object manager has comprehensive mechanisms for managing object lifetimes within the kernel, neither the NT APIs nor the Win32 APIs provide a reference mechanism for dealing with the use of handles across multiple concurrent threads in user mode. Thus many multithreaded applications have race conditions and bugs where they will close a handle in one thread before they are finished with it in another. Or close a handle multiple times. Or close a handle that another thread is still using and reopen it to refer to a different object.

Perhaps the Windows APIs should have been designed to require a close API per object type rather than the single generic NtClose operation. That would have at least reduced the frequency of bugs due to user-mode threads closing the wrong handles. Another solution might be to embed a sequence field in each handle in addition to the index into the handle table.

To help application writers find problems like these in their programs, Windows has an **application verifier** that software developers can download from Microsoft. Similar to the verifier for drivers we will describe in Sec. 11.7, the application verifier does extensive rules checking to help programmers find bugs that might not be found by ordinary testing. It can also turn on a FIFO ordering for the handle free list, so that handles are not reused immediately (i.e., turns off the better-performing LIFO ordering normally used for handle tables). Keeping handles from being reused quickly transforms situations where an operation uses the wrong handle into use of a closed handle, which is easy to detect.

The device object is one of the most important and versatile kernel-mode objects in the executive. The type is specified by the I/O manager, which along with the device drivers, are the primary user of device objects. Device objects are closely related to drivers, and each device object usually has a link to a specific driver object, which describes how to access the I/O processing routines for the driver corresponding to the device.

Device objects represent hardware devices, interfaces, and buses, as well as logical disk partitions, disk volumes, and even file systems and kernel extensions like antivirus filters. Many device drivers are given names, so they can be accessed without having to open handles to instances of the devices, as in UNIX. We will use device objects to illustrate how the *Parse* procedure is used, as illustrated in Fig. 11-22:

1. When an executive component, such as the I/O manager implementing the native system call NtCreateFile, calls ObOpenObjectByName in the object manager, it passes a Unicode pathname for the NT namespace, say *\??\C:\foo\bar*.
2. The object manager searches through directories and symbolic links and ultimately finds that *\??\C:* refers to a device object (a type defined by the I/O manager). The device object is a leaf node in the part of the NT name space that the object manager manages.
3. The object manager then calls the *Parse* procedure for this object type, which happens to be IopParseDevice implemented by the I/O manager. It not only passes a pointer to the device object it found (for *C:*), but also the remaining string *\foo\bar*.
4. The I/O manager will create an **IRP** (**I/O Request Packet**), allocate a file object, and send the request to the stack of I/O devices determined by the device object found by the object manager.
5. The IRP is passed down the I/O stack until it reaches a device object representing the file system instance for *C:*. At each stage, control is passed to an entry point into the driver object associated with the

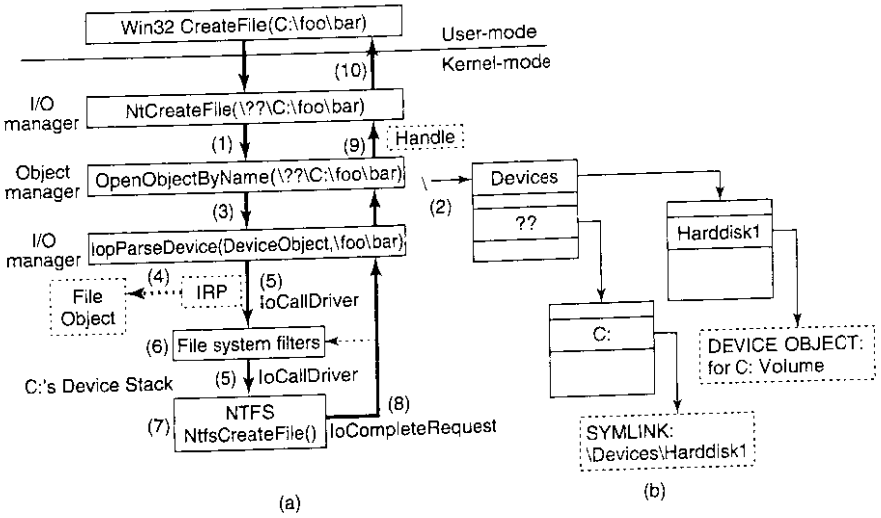

Figure 11-22. I/O and object manager steps for creating/opening a file and getting back a file handle.

device object at that level. The entry point used in this case is for CREATE operations, since the request is to create or open a file named *\foo\bar* on the volume.

6. The device objects encountered as the IRP heads toward the file system represent file system filter drivers, which may modify the I/O operation before it reaches the file system device object. Typically these intermediate devices represent system extensions like antivirus filters.
7. The file system device object has a link to the file system driver object, say NTFS. So, the driver object contains the address of the CREATE operation within NTFS.
8. NTFS will fill in the file object and return it to the I/O manager, which returns back up through all the devices on the stack until IopParseDevice returns to the object manager (see Sec. 11.8).
9. The object manager is finished with its namespace lookup. It received back an initialized object from the *Parse* routine (which happens to be a file object—not the original device object it found). So the object manager creates a handle for the file object in the handle table of the current process, and returns the handle to its caller.

10. The final step is to return back to the user-mode caller, which in this example is the Win32 API CreateFile which will return the handle to the application.

Executive components can create new types dynamically, by calling the ObCreateObjectType interface to the object manager. There is no definitive list of object types and they change from release to release. Some of the more common ones in Windows Vista are listed in Fig. 11-23. Let us briefly go over the object types in the figure.

Type	Description
Process	User process
Thread	Thread within a process
Semaphore	Counting semaphore used for inter-process synchronization
Mutex	Binary semaphore used to enter a critical region
Event	Synchronization object with persistent state (signaled/not)
ALPC Port	Mechanism for inter-process message passing
Timer	Object allowing a thread to sleep for a fixed time interval
Queue	Object used for completion notification on asynchronous I/O
Open file	Object associated with an open file
Access token	Security descriptor for some object
Profile	Data structure used for profiling CPU usage
Section	Object used for representing mappable files
Key	Registry key, used to attach registry to object manager namespace
Object directory	Directory for grouping objects within the object manager
Symbolic link	Refers to another object manager object by pathname
Device	I/O device object for a physical device, bus, driver, or volume instance
Device driver	Each loaded device driver has its own object

Figure 11-23. Some common executive object types managed by object manager.

Process and thread are obvious. There is one object for every process and every thread, which holds the main properties needed to manage the process or thread. The next three objects, semaphore, mutex, and event, all deal with interprocess synchronization. Semaphores and mutexes work as expected, but with various extra bells and whistles (e.g., maximum values and timeouts). Events can be in one of two states: signaled or nonsignaled. If a thread waits on an event that is in signaled state, the thread is released immediately. If the event is in nonsignaled state, it blocks until some other thread signals the event, which releases either all blocked threads (notification events) or just the first blocked thread (synchronization events). An event can also be set up so that after a signal has been

successfully waited for, it will automatically revert to the nonsignaled state, rather than staying in the signaled state.

Port, timer, and queue objects also relate to communication and synchronization. Ports are channels between processes for exchanging LPC messages. Timers provide a way to block for a specific time interval. Queues are used to notify threads that a previously started asynchronous I/O operation has completed or that a port has a message waiting. (They are designed to manage the level of concurrency in an application, and are used in high-performance multiprocessor applications, like SQL).

Open file objects are created when a file is opened. Files that are not opened do not have objects managed by the object manager. Access tokens are security objects. They identify a user and tell what special privileges the user has, if any. Profiles are structures used for storing periodic samples of the program counter of a running thread to see where the program is spending its time.

Sections are used to represent memory objects that applications can ask the memory manager to map into their address space. They record the section of the file (or pagefile) that represents the pages of the memory object when they are on disk. Keys represent the mount point for the registry namespace on the object manager namespace. There is usually only one key object, named *\REGISTRY*, which connects the names of the registry keys and values to the NT namespace.

Object directories and symbolic links are entirely local to the part of the NT namespace managed by the object manager. They are similar to their file system counterparts: Directories allow related objects to be collected together. Symbolic links allow a name in one part of the object namespace to refer to an object in a different part of the object namespace.

Each device known to the operating system has one or more device objects that contain information about it and are used to refer to the device by the system. Finally, each device driver that has been loaded has a driver object in the object space. The driver objects are shared by all the device objects that represent instances of the devices controlled by those drivers.

Other objects, not shown, have more specialized purposes, such as interacting with kernel transactions, or the Win32 threadpool's worker thread factory.

11.3.4 Subsystems, DLLs, and User-Mode Services

Going back to Fig. 11-6, we see that the Windows Vista operating system consists of components in kernel-mode and components, in user mode. We have now completed our overview of the kernel-mode components; so it is time to look at the user-mode components, of which there are three kinds that are particularly important to Windows: environment subsystems, DLLs, and service processes.

We have already described the Windows subsystem model; we will not go into more detail now other than to mention that in the original design of NT, subsystems were seen as a way of supporting multiple operating system personalities

with the same underlying software running in kernel mode. Perhaps this was an attempt to avoid having operating systems compete for the same platform, as VMS and Berkeley UNIX did on DEC's VAX. Or maybe it was just that nobody at Microsoft knew whether OS/2 would be a success as a programming interface, so they were hedging their bets. In any case, OS/2 became irrelevant, and a latecomer, the Win32 API designed to be shared with Windows 95, became dominant.

A second key aspect of the user-mode design of Windows is the dynamic link library (DLL) which is code that is linked to executable programs at run-time rather than compile-time. Shared libraries are not a new concept, and most modern operating systems use them. In Windows almost all libraries are DLLs, from the system library *ntdll.dll* that is loaded into every process to the high-level libraries of common functions that are intended to allow rampant code-reuse by application developers.

DLLs improve the efficiency of the system by allowing common code to be shared among processes, reduce program load times from disk by keeping commonly used code around in memory, and increase the serviceability of the system by allowing operating system library code to be updated without having to recompile or relink all the application programs that use it.

On the other hand, shared libraries introduce the problem of versioning and increase the complexity of the system because changes introduced into a shared library to help one particular program have the potential of exposing latent bugs in other applications, or just breaking them due to changes in the implementation—a problem that in the Windows world is referred to as **DLL hell**.

The implementation of DLLs is simple in concept. Instead of the compiler emitting code that calls directly to subroutines in the same executable image, a level of indirection is introduced: the **IAT** (**Import Address Table**). When an executable is loaded it is searched for the list of DLLs that must also be loaded (this will be a graph in general, as the listed DLLs will themselves will generally list other DLLs needed in order to run). The required DLLs are loaded and the IAT is filled in for them all.

The reality is more complicated. Another problem is that the graphs that represent the relationships between DLLs can contain cycles, or have nondeterministic behaviors, so computing the list of DLLs to load can result in a sequence that does not work. Also, in Windows the DLL libraries are given a chance to run code whenever they are loaded into a process, or when a new thread is created. Generally, this is so they can perform initialization, or allocate per-thread storage, but many DLLs perform a lot of computation in these *attach* routines. If any of the functions called in an *attach* routine needs to examine the list of loaded DLLs, a deadlock can occur hanging the process.

DLLs are used for more than just sharing common code. They enable a *hosting* model for extending applications. Internet Explorer can download and link to DLLs called **ActiveX controls**. At the other end of the Internet, Web servers also

load dynamic code to produce a better Web experience for the pages they display. Applications like Microsoft Office link and run DLLs to allow Office to be used as a platform for building other applications. The COM (component object model) style of programming allows programs to dynamically find and load code written to provide a particular published interface, which leads to in-process hosting of DLLs by almost all the applications that use COM.

All this dynamic loading of code has resulted in even greater complexity for the operating system, as library version management is not just a matter of matching executables to the right versions of the DLLs, but sometimes loading multiple versions of the same DLL into a process—which Microsoft calls **side-by-side**. A single program can host two different dynamic code libraries, each of which may want to load the same Windows library—yet have different version requirements for that library.

A better solution would be hosting code in separate processes. But out-of-process hosting of code results has lower performance, and makes for a more complicated programming model in many cases. Microsoft has yet to develop a good solution for all of this complexity in user mode. It makes one yearn for the relative simplicity of kernel mode.

One of the reasons that kernel mode has less complexity than user mode is that it supports relatively few extensibility opportunities outside of the device driver model. In Windows, system functionality is extended by writing user-mode services. This worked well enough for subsystems, and works even better when only a few new services are being provided rather than a complete operating system personality. There are relatively few functional differences between services implemented in the kernel and services implemented in user-mode processes. Both the kernel and process provide private address spaces where data structures can be protected and service requests can be scrutinized.

However, there can be significant performance differences between services in the kernel versus services in user-mode processes. Entering the kernel from user-mode is slow on modern hardware, but not as slow as having to do it twice because you are switching back and forth to another process. Also cross-process communication has lower bandwidth.

Kernel-mode code can (very carefully) access data at the user-mode addresses passed as parameters to its system calls. With user-mode services, that data must either be copied to the service process, or some games played by mapping memory back and forth (the ALPC facilities in Windows Vista handle this under the covers).

In the future it is possible that the hardware costs of crossing between address spaces and protection modes will be reduced, or perhaps even become irrelevant. The Singularity project in Microsoft Research (Fandrich, et al., 2006) uses runtime techniques, like those used with C# and Java, to make protection a completely software issue. No hardware switching between address spaces or protection modes is required.

Windows Vista makes significant use of user-mode service processes to extend the functionality of the system. Some of these services are strongly tied to the operation of kernel-mode components, such as *lsass.exe* which is the local security authentication service which manages the token objects that represent user-identity, as well as managing encryption keys used by the file system. The user-mode plug-and-play manager is responsible for determining the correct driver to use when a new hardware device is encountered, installing it, and telling the kernel to load it. Many facilities provided by third parties, such as antivirus and digital rights management, are implemented as a combination of kernel-mode drivers and user-mode services.

In Windows Vista *taskmgr.exe* has a tab which identifies the services running on the system. (Earlier versions of Windows will show a list of services with the *net start* command). Multiple services can be seen to be running in the same process (**svchost.exe**). Windows does this for many of its own boot-time services to reduce the time needed to start up the system. Services can be combined into the same process as long as they can safely operate with the same security credentials.

Within each of the shared service processes, individual services are loaded as DLLs. They normally share a pool of threads using the Win32 threadpool facility, so that only the minimal number of threads needs to be running across all the resident services.

Services are common sources of security vulnerabilities in the system because they are often accessible remotely (depending on the TCP/IP firewall and IP Security settings), and not all programmers who write services are as careful as they should be to validate the parameters and buffers that are passed in via RPC.

The number of services running constantly in Windows is staggering. Yet few of those services ever receive a single request, though if they do it is likely to be from an attacker attempting to exploit a vulnerability. As a result more and more services in Windows are turned off by default, particularly on versions of Windows Server.

11.4 PROCESSES AND THREADS IN WINDOWS VISTA

Windows has a number of concepts for managing the CPU and grouping resources together. In the following sections we will examine these, discussing some of the relevant Win32 API calls, and show how they are implemented.

11.4.1 Fundamental Concepts

In Windows Vista processes are containers for programs. They hold the virtual address space, the handles that refer to kernel-mode objects, and threads. In their role as a container for threads they hold common resources used for thread

execution, such as the pointer to the quota structure, the shared token object, and default parameters used to initialize threads—including the priority and scheduling class. Each process has user-mode system data, called the **PEB (Process Environment Block)**. The PEB includes the list of loaded modules (i.e., the EXE and DLLs), the memory containing environment strings, the current working directory, and data for managing the process' heaps—as well as lots of special-case Win32 cruft that has been added over time.

Threads are the kernel's abstraction for scheduling the CPU in Windows. Priorities are assigned to each thread based on the priority value in the containing process. Threads can also be **affinitized** to only run on certain processors. This helps concurrent programs running on multiprocessors to explicitly spread out work. Each thread has two separate call stacks, one for execution in user mode and one for kernel mode. There is also a **TEB (Thread Environment Block)** that keeps user-mode data specific to the thread, including per-thread storage (**Thread Local Storage**) and fields for Win32, language and cultural localization, and other specialized fields that have been added by various facilities.

Besides the PEBs and TEBs, there is another data structure that kernel mode shares with each process, namely, **user shared data**. This is a page that is writable by the kernel, but read-only in every user-mode process. It contains a number of values maintained by the kernel, such as various forms of time, version information, amount of physical memory, and a large number of shared flags used by various user-mode components, such as COM, terminal services, and the debuggers. The use of this read-only shared page is purely a performance optimization, as the values could also be obtained by a system call into kernel mode. But system calls are much more expensive than a single memory access, so for some system-maintained fields, such as the time, this makes a lot of sense. The other fields, such as the current time zone, change infrequently, but code that relies on these fields must query them often just to see if they have changed.

Processes

Processes are created from section objects, each of which describes a memory object backed by a file on disk. When a process is created, the creating process receives a handle for the process that allows it to modify the new process by mapping sections, allocating virtual memory, writing parameters and environmental data, duplicating file descriptors into its handle table, and creating threads. This is very different than how processes are created in UNIX and reflects the difference in the target systems for the original designs of UNIX versus Windows.

As described in Sec. 11.1, UNIX was designed for 16-bit single processor systems that used swapping to share memory among processes. In such systems, having the process as the unit of concurrency and using an operation like fork to create processes was a brilliant idea. To run a new process with small memory and no virtual memory hardware, processes in memory have to be swapped out to

disk to create space. UNIX originally implemented fork simply by swapping out the parent process and handing its physical memory to the child. The operation was almost free.

In contrast, the hardware environment at the time Cutler's team wrote NT was 32-bit multiprocessor systems with virtual memory hardware to share 1–16 MB of physical memory. Multiprocessors provide the opportunity to run parts of programs concurrently, so NT used processes as containers for sharing memory and object resources, and used threads as the unit of concurrency for scheduling.

Of course, the systems of the next few years will look nothing like either of these target environments, having 64-bit address spaces with dozens (or hundreds) of CPU cores per chip socket, and multiple GB of physical memory—as well as **flash devices** and other nonvolatile stores added to the memory hierarchy, broader support for virtualization. ubiquitous networking, and support for synchronization innovations like **transactional memory**. Windows and UNIX will continue to be adapted to new hardware realities, but what will be really interesting is to see what new operating systems are designed specifically for systems based on these advances.

Jobs and Fibers

Windows can group processes together into jobs, but the job abstraction is not very general. It was specifically designed for grouping processes in order to apply constraints to the threads they contain, such as limiting resource use via a shared quota or enforcing a **restricted token** that prevents threads from accessing many system objects. The most significant property of jobs for resource management is that once a process is in a job, all processes threads in those processes create will also be in the job. There is no escape. As suggested by the name, jobs were designed for situations that are more like batch processing than ordinary interactive computing.

A Process can be in (at most) one job. This makes sense, as what it would mean for a process to be subject to multiple shared quotas or restricted tokens is hard to define. But this means that if multiple services in the system attempt to use jobs to manage processes, there will be conflicts if they attempt to manage the same processes. For example, an administrative tool which sought to constrain resource use by putting processes into jobs would be foiled if the process first inserted itself into its own job, or if a security tool had already put the process into a job with a restricted token to limit its access to system objects. As a result the use of jobs within Windows is rare.

Fig. 11-24 shows the relationship between jobs, processes, threads, and fibers. Jobs contain processes. Processes contain threads. But threads do not contain fibers. The relationship of threads to fibers is normally many-to-many.

Fibers are created by allocating a stack and a user-mode fiber data structure for storing registers and data associated with the fiber. Threads are converted to

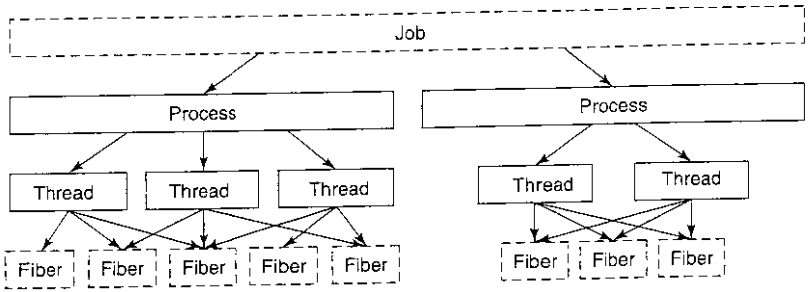

Figure 11-24. The relationship between jobs, processes, threads and fibers. Jobs and fibers are optional; not all processes are in jobs or contain fibers.

fibers, but fibers can also be created independently of threads. Such fibers will not run until a fiber already running on a thread explicitly calls SwitchToFiber to run the fiber. Threads could attempt to switch to the a fiber that is already running, so the programmer must provide synchronization to prevent this.

The primary advantage of fibers is that the overhead of switching between fibers is much, much lower than switching between threads. A thread switch requires entering and exiting the kernel. A fiber switch saves and restores a few registers without changing modes at all.

Although fibers are cooperatively scheduled, if there are multiple threads scheduling the fibers, a lot of careful synchronization is required to make sure fibers do not interfere with each other. To simplify the interaction between threads and fibers, it is often useful to create only as many threads as there are processors to run them, and affinitize the threads to each run only on a distinct set of available processors, or even just one processor.

Each thread can then run a particular subset of the fibers, establishing a one-to-many relationship between threads and fibers which simplifies synchronization. Even so there are still many difficulties with fibers. Most Win32 libraries are completely unaware of fibers, and applications that attempt to use fibers as if they were threads will encounter various failures. The kernel has no knowledge of fibers, and when a fiber enters the kernel, the thread it is executing on may block and the kernel will schedule an arbitrary thread on the processor, making it unavailable to run other fibers. For these reasons fibers are rarely used except when porting code from other systems that explicitly need the functionality provided by fibers. A summary of these abstractions is given in Fig. 11-25.

Threads

Every process normally starts out with one thread, but new ones can be created dynamically. Threads form the basis of CPU scheduling, as the operating system always selects a thread to run, not a process. Consequently, every thread has

Name	Description	Notes
Job	Collection of processes that share quotas and limits	Rarely used
Process	Container for holding resources	
Thread	Entity scheduled by the kernel	
Fiber	Lightweight thread managed entirely in user space	Rarely used

Figure 11-25. Basic concepts used for CPU and resource management.

a state (ready, running, blocked, etc), whereas processes do not have scheduling states. Threads can be created dynamically by a Win32 call that specifies the address within the enclosing process' address space it is to start running at.

Every thread has a thread ID, which is taken from the same space as the process IDs, so a single ID can never be in use for both a process and a thread at the same time. Process and thread IDs are multiples of four because they are actually allocated by the executive using a special handle table set aside for allocating IDs. The system is reusing the scalable handle management facility shown in Figs. 11-18 and 11-19. The handle table does not have references on objects, but does use the pointer field to point at the process or thread so that the lookup of a process or thread by ID is very efficient. FIFO ordering of the list of free handles is turned on for the ID table in recent versions of Windows so that IDs are not immediately reused. The problems with immediate reuse are explored in the problems at the end of this chapter.

A thread normally runs in user mode, but when it makes a system call it switches to kernel mode and continues to run as the same thread with the same properties and limits it had in user mode. Each thread has two stacks, one for use when it is in user-mode and one for use when it is in kernel mode. Whenever a thread enters the kernel, it switches to the kernel-mode stack. The values of the user-mode registers are saved in a **CONTEXT** data structure at the base of the kernel-mode stack. Since the only way for a user-mode thread to not be running is for it to enter the kernel, the CONTEXT for a thread always contains its register state when it is not running. The CONTEXT for each thread can be examined and modified from any process with a handle to the thread.

Threads normally run using the access token of their containing process, but in certain cases related to client/server computing, a thread running in a service process can impersonate its client, using a temporary access token based on the client's token so it can perform operations on the client's behalf. (In general a service cannot use the client's actual token, as the client and server may be running on different systems.)

Threads are also the normal focal point for I/O. Threads block when performing synchronous I/O, and the outstanding I/O request packets for asynchronous I/O are linked to the thread. When a thread is finished executing, it can exit.

Any I/O requests pending for the thread will be canceled. When the last thread still active in a process exits, the process terminates.

It is important to realize that threads are a scheduling concept, not a resource ownership concept. Any thread is able to access all the objects that belong to its process. All it has to do is use the handle value and make the appropriate Win32 call. There is no restriction on a thread that it cannot access an object because a different thread created or opened it. The system does not even keep track of which thread created which object. Once an object handle has been put in a process' handle table, any thread in the process can use it, even it if is impersonating a different user.

As described previously, in addition to the normal threads that run within user processes Windows has a number of system threads that run only in kernel mode and are not associated with any user process. All such system threads run in a special process called the **system process**. This process does not have a user-mode address space. It provides the environment that threads execute in when they are not operating on behalf of a specific user-mode process. We will study some of these threads later when we come to memory management. Some perform administrative tasks, such as writing dirty pages to the disk, while others form the pool of worker threads that are assigned to run specific short-term tasks delegated by executive components or drivers that need to get some work done in the system process.

11.4.2 Job, Process, Thread, and Fiber Management API Calls

New processes are created using the Win32 API function CreateProcess. This function has many parameters and lots of options. It takes the name of the file to be executed, the command-line strings (unparsed), and a pointer to the environment strings. There are also flags and values that control many details such as how security is configured for the process and first thread, debugger configuration, and scheduling priorities. A flag also specifies whether open handles in the creator are to be passed to the new process. The function also takes the current working directory for the new process and an optional data structure with information about the GUI Window the process is to use. Rather than returning just a process ID for the new process, Win32 returns both handles and IDs, both for the new process and for its initial thread.

The large number of parameters reveals a number of differences from the design of process creation in UNIX.

1. The actual search path for finding the program to execute is buried in the library code for Win32, but managed more explicitly in UNIX.
2. The current working directory is a kernel-mode concept in UNIX but a user-mode string in Windows. Windows *does* open a handle on the current directory for each process, with the same annoying effect as in UNIX: You cannot delete the directory, unless it happens to be across the network, in which case you can delete it.
3. UNIX parses the command line and passes an array of parameters, while Win32 leaves argument parsing up to the individual program. As a consequence, different programs may handle wildcards (e.g., *.txt) and other special symbols in an inconsistent way.
4. Whether file descriptors can be inherited in UNIX is a property of the handle. In Windows it is a property of both the handle and a parameter to process creation.
5. Win32 is GUI-oriented, so new processes are directly passed information about their primary window, while this information is passed as parameters to GUI applications in UNIX.
6. Windows does not have a SETUID bit as a property of the executable, but one process can create a process that runs as a different user, as long as it can obtain a token with that user's credentials.
7. The process and thread handle returned from Windows can be used to modify the new process/thread in many substantive ways, including duplication of handles and setting up the environment variables in the new process. UNIX just makes modifications to the new process between the fork and exec calls.

Some of these differences are historical and philosophical. UNIX was designed to be command-line-oriented rather than GUI-oriented like Windows. UNIX users are more sophisticated, and understand concepts like *PATH* variables. Windows Vista inherited a lot of legacy from MS-DOS.

The comparison is also skewed because Win32 is a user-mode wrapper around the native NT process execution, much as the *system* library function wraps fork/exec in UNIX. The actual NT system calls for creating processes and threads, NtCreateProcess and NtCreateThread, are much simpler than the Win32 versions. The main parameters to NT process creation are a handle on a section representing the program file to run, a flag specifying whether the new process should, by default, inherit handles from the creator, and parameters related to the security model. All the details of setting up the environment strings, and creating the initial thread, are left to user-mode code that can use the handle on the new process to manipulate its virtual address space directly.

To support the POSIX subsystem, native process creation has an option to create a new process by copying the virtual address space of another process rather than mapping a section object for a new program. This is only used to implement fork for POSIX, and not by Win32.

Thread creation passes the CPU context to use for the new thread (which includes the stack pointer and initial instruction pointer), a template for the TEB,

and a flag saying whether the thread should be immediately run or created in a suspended state (waiting for somebody to call NtResumeThread on its handle). Creation of the user-mode stack and pushing of the *argv/argc* parameters is left to user-mode code calling the native NT memory management APIs on the process handle.

In the Windows Vista release, a new native API for processes was included which moves many of the user-mode steps into the kernel-mode executive, and combines process creation with creation of the initial thread. The reason for the change was to support the use of processes as security boundaries. Normally, all processes created by a user are considered to be equally trusted. It is the user, as represented by a token, that determines where the trust boundary is. This change in Windows Vista allows processes to also provide trust boundaries, but this means that the creating process does not have sufficient rights regarding a new process handle to implement the details of process creation in user mode.

Interprocess Communication

Threads can communicate in a wide variety of ways, including pipes, named pipes, mailslots, sockets, remote procedure calls, and shared files. Pipes have two modes: byte and message, selected at creation time. Byte-mode pipes work the same way as in UNIX. Message-mode pipes are somewhat similar but preserve message boundaries, so that four writes of 128 bytes will be read as four 128-byte messages, and not as one 512-byte message, as might happen with byte-mode pipes. Named pipes also exist and have the same two modes as regular pipes. Named pipes can also be used over a network but regular pipes cannot.

Mailslots are a feature of the OS/2 operating system implemented in Windows for compatibility. They are similar to pipes in some ways, but not all. For one thing, they are one way, whereas pipes are two way. They could be used over a network but do not provide guaranteed delivery. Finally, they allow the sending process to broadcast a message to many receivers, instead of to just one receiver. Both mailslots and named pipes are implemented as file systems in Windows, rather than executive functions. This allows them to be accessed over the network using the existing remote file system protocols.

Sockets are like pipes, except that they normally connect processes on different machines. For example, one process writes to a socket and another one on a remote machine reads from it. Sockets can also be used to connect processes on the same machine, but since they entail more overhead than pipes, they are generally only used in a networking context. Sockets were originally designed for Berkeley UNIX, and the implementation was made widely available. Some of the Berkeley code and data structures are still present in Windows today, as acknowledged in the release notes for the system.

RPCs (remote procedure calls) are a way for process *A* to have process *B* call a procedure in *B*'s address space on *A*'s behalf and return the result to *A*. Various

restrictions on the parameters exist. For example, it makes no sense to pass a pointer to a different process, so data structures have to be packaged up and transmitted in a nonprocess specific way. RPC is normally implemented as an abstraction layer on top of a transport layer. In the case of Windows, the transport can be TCP/IP sockets, named pipes, or ALPC. ALPC (Advanced Local Procedure Call) is a message-passing facility in the kernel-mode executive. It is optimized for communicating between processes on the local machine and does not operate across the network. The basic design is for sending messages that generate replies, implementing a lightweight version of remote procedure call which the RPC package can build on top of to provide a richer set of features than available in ALPC. ALPC is implemented using a combination of copying parameters and temporary allocation of shared memory, based on the size of the messages.

Finally, processes can share objects. This includes section objects, which can be mapped into the virtual address space of different processes at the same time. All writes done by one process then appear in the address spaces of the other processes. Using this mechanism, the shared buffer used in producer-consumer problems can easily be implemented.

Synchronization

Processes can also use various types of synchronization objects. Just as Windows Vista provides numerous inter-process communication mechanisms, it also provides numerous synchronization mechanisms, including semaphores, mutexes, critical regions, and events. All of these mechanisms work with threads, not processes, so that when a thread blocks on a semaphore, other threads in that process (if any) are not affected and can continue to run.

A semaphore is created using the CreateSemaphore Win32 API function, which can initialize it to a given value and define a maximum value as well. Semaphores are kernel-mode objects and thus have security descriptors and handles. The handle for a semaphore can be duplicated using DuplicateHandle and passed to another process so that multiple processes can synchronize on the same semaphore. A semaphore can also be given a name in the Win32 namespace, and have an ACL set to protect it. Sometimes sharing a semaphore by name is more appropriate than duplicating the handle.

Calls for up and down exist, although they have the somewhat odd names of ReleaseSemaphore (up) and WaitForSingleObject (down). It is also possible to give WaitForSingleObject a timeout, so the calling thread can be released eventually, even if the semaphore remains at 0 (although timers reintroduce races). WaitForSingleObject and WaitForMultipleObjects are the common interfaces used for waiting on the dispatcher objects discussed in Sec. 11.3. While it would have been possible to wrap the single-object version of these APIs in a wrapper with a somewhat more semaphore-friendly name, many threads use the multiple-object

version which may include waiting for multiple flavors of synchronization objects as well as other events like process or thread termination, I/O completion, and messages being available on sockets and ports.

Mutexes are also kernel-mode objects used for synchronization, but simpler than semaphores because they do not have counters. They are essentially locks, with API functions for locking WaitForSingleObject and unlocking ReleaseMutex. Like semaphore handles, mutex handles can be duplicated and passed between processes so that threads in different processes can access the same mutex.

A third synchronization mechanism is called **critical sections**, which implement the concept of critical regions. These are similar to mutexes in Windows, except local to the address space of the creating thread. Because critical sections are not kernel-mode objects, they do not have explicit handles or security descriptors and cannot be passed between processes. Locking and unlocking are done with EnterCriticalSection and LeaveCriticalSection, respectively. Because these API functions are performed initially in user space and only make kernel calls when blocking is needed, they are much faster than mutexes. Critical sections are optimized to combine spin locks (on multiprocessors) with the use of kernel synchronization only when necessary. In many applications most critical sections are so rarely contended or have such short hold times that it is never necessary to allocate a kernel synchronization object. This results in a very significant savings in kernel memory.

The last synchronization mechanism we discuss uses kernel-mode objects called **events**. As we have described previously, there are two kinds: **notification events** and **synchronization events**. An event can be in one of two states: signaled or not-signaled. A thread can wait for an event to be signaled with WaitForSingleObject. If another thread signals an event with SetEvent, what happens depends on the type of event. With a notification event, all waiting threads are released and the event stays set until manually cleared with ResetEvent. With a synchronization event, if one or more threads are waiting, exactly one thread is released and the event is cleared. An alternative operation is PulseEvent, which is like SetEvent except that if nobody is waiting, the pulse is lost and the event is cleared. In contrast, a SetEvent that occurs with no waiting threads is remembered by leaving the event in the signaled state so a subsequent thread that calls a wait API for the event will not actually wait.

The number of Win32 API calls dealing with processes, threads, and fibers is nearly 100, a substantial number of which deal with IPC in one form or another. A summary of the ones discussed above as well as some other important ones is given in Fig. 11-26.

Note that not all of these are just system calls. While some are wrappers, others contain significant library code which maps the Win32 semantics onto the native NT APIs. Still others, like the fiber APIs, are purely user-mode functions since, as we mentioned earlier, kernel mode in Windows Vista knows nothing about fibers. The are entirely implemented by user-mode libraries.

Win32 API Function	Description
CreateProcess	Create a new process
CreateThread	Create a new thread in an existing process
CreateFiber	Create a new fiber
ExitProcess	Terminate current process and all its threads
ExitThread	Terminate this thread
ExitFiber	Terminate this fiber
SwitchToFiber	Run a different fiber on the current thread
SetPriorityClass	Set the priority class for a process
SetThreadPriority	Set the priority for one thread
CreateSemaphore	Create a new semaphore
CreateMutex	Create a new mutex
OpenSemaphore	Open an existing semaphore
OpenMutex	Open an existing mutex
WaitForSingleObject	Block on a single semaphore, mutex, etc.
WaitForMultipleObjects	Block on a set of objects whose handles are given
PulseEvent	Set an event to signaled then to nonsignaled
ReleaseMutex	Release a mutex to allow another thread to acquire it
ReleaseSemaphore	Increase the semaphore count by 1
EnterCriticalSection	Acquire the lock on a critical section
LeaveCriticalSection	Release the lock on a critical section

Figure 11-26. Some of the Win32 calls for managing processes, threads, and fibers.

11.4.3 Implementation of Processes and Threads

In this section we will get into more detail about how Windows creates a process (and the initial thread). Because Win32 is the most documented interface, we will start there. But we will quickly work our way down into the kernel and understand the implementation of the native API call for creating a new process. There are many more specific details that we will gloss over here, such as how WOW16 and WOW64 have special code in the creation path, or how the system supplies application-specific fix-ups to get around small incompatibilities and latent bugs in applications. We will focus on the main code paths that get executed whenever processes are created, as well as look at a few of the details that fill in gaps in what we have covered so far.

A process is created when another process makes the Win32 CreateProcess call. This call invokes a (user-mode) procedure in *kernel32.dll* that creates the process in several steps using multiple system calls and by performing other work.

1. Convert the executable file name given as a parameter from a Win32 pathname to an NT pathname. If the executable just has a name without a directory pathname, it is searched for in the directories listed in the default directories (which include, but are not limited to, those in the PATH variable in the environment).
2. Bundle up the process creation parameters and pass them, along with the full pathname of the executable program, to the native API NtCreateUserProcess. (This API was added in Windows Vista so that the details of process creation could be handled in kernel mode, allowing processes to be used as a trust boundary. The previous native APIs described above still exist, but are no longer used by the Win32 CreateProcess call.)
3. Running in kernel-mode, NtCreateUserProcess processes the parameters, and then opens the program image and creates a section object that can be used to map the program into the new process' virtual address space.
4. The process manager allocates and initializes the process object (the kernel data structure representing a process to both the kernel and executive layers).
5. The memory manager creates the address space for the new process by allocating and initializing the page directories and the virtual address descriptors which describe the kernel-mode portion, including the process-specific regions, such as the **self-map** page directory entry that gives each process kernel-mode access to the physical entry that gives each process kernel-mode access to the physical pages in its entire page table using kernel virtual addresses. (We will describe the self-map in more detail in Sec. 11.5.)
6. A handle table is created for the new process, and all the handles from the caller that are allowed to be inherited are duplicated into it.
7. The shared user page is mapped, and the memory manager initializes the working-set data structures used for deciding what pages to trim from a process when physical memory is low. The pieces of the executable image represented by the section object are mapped into the new process' user-mode address space.
8. The executive creates and initializes the user-mode Process Environment Block (PEB) which is used by both user-mode and the kernel to maintain process-wide state information, such as the user-mode heap pointers and the list of loaded libraries (DLLs).
9. Virtual memory is allocated in the new process, and used to pass parameters, including the environment strings and command line.

10. A process ID is allocated from the special handle table (ID table) the kernel maintains for efficiently allocating locally unique IDs for processes and threads.
11. A thread object is allocated and initialized. A user-mode stack is allocated along with the Thread Environment Block (TEB). The *CONTEXT* record which contains the thread's initial values for the CPU registers (including the instruction and stack pointers), is initialized.
12. The process object is added to the global list of processes. Handles for the process and thread objects are allocated in the caller's handle table. An ID for the initial thread is allocated from the ID table.
13. NtCreateUserProcess returns to user-mode with the new process created, containing a single thread that is ready to run but suspended.
14. If the NT API fails, the Win32 code checks to see if this might be a process belonging to another subsystem like WOW64. Or perhaps the program is marked that it should be run under the debugger. These special cases are handled with special code in the user-mode CreateProcess code.
15. If NtCreateUserProcess was successful, there is still some work to be done. Win32 processes have to be registered with the Win32 subsystem process, *csrss.exe*. Kernel32.dll sends a message to *csrss* telling it about the new process along with the process and thread handles so it can duplicate itself. The process and threads are entered into the subsystems' tables so that they have a complete list of all Win32 processes and threads. The subsystem then displays a cursor containing a pointer with an hourglass to tell the user that something is going on but that the cursor can be used in the meanwhile. When the process makes its first GUI call, usually to create a window, the cursor is removed (it times out after 2 seconds if no call is forthcoming).
16. If the process is restricted, such as low-rights Internet Explorer, the token is modified to restrict what objects the new process can access.
17. If the application program was marked as needing to be *shimmed* to run compatibly with the current version of Windows, the specified *shims* are applied. (Shims usually wrap library calls to slightly modify their behavior, such as returning a fake version number or delaying the freeing of memory).
18. Finally, call NtResumeThread to unsuspend the thread, and return the structure to the caller containing the IDs and handles for the process and thread that were just created.

Scheduling

The Windows kernel does not have any central scheduling thread. Instead, when a thread cannot run any more, the thread enters kernel-mode and calls into the scheduler itself to see which thread to switch to. The following conditions cause the currently running thread to execute the scheduler code:

1. The currently running thread blocks on a semaphore, mutex, event, I/O, etc.
2. The thread signals an object (e.g., does an up on a semaphore or causes an event to be signaled).
3. The quantum expires.

In case 1, the thread is already running in kernel-mode to carry out the operation on the dispatcher or I/O object. It cannot possibly continue, so it calls the scheduler code to pick its successor and load that thread's CONTEXT record to resume running it.

In case 2, the running thread is in the kernel, too. However, after signaling some object, it can definitely continue because signaling an object never blocks. Still, the thread is required to call the scheduler to see if the result of its action has released a thread with a higher scheduling priority that is now ready to run. If so, a thread switch occurs since Windows is fully preemptive (i.e., thread switches can occur at any moment, not just at the end of the current thread's quantum). However, in the case of a multiprocessor, a thread that was made ready may be scheduled on a different CPU and the original thread can continue to execute on the current CPU even though its scheduling priority is lower.

In case 3, an interrupt to kernel mode occurs, at which point the thread executes the scheduler code to see who runs next. Depending on what other threads are waiting, the same thread may be selected, in which case it gets a new quantum and continues running. Otherwise a thread switch happens.

The scheduler is also called under two other conditions:

1. An I/O operation completes.
2. A timed wait expires.

In the first case, a thread may have been waiting on this I/O and is now released to run. A check has to be made to see if it should preempt the running thread since there is no guaranteed minimum run time. The scheduler is not run in the interrupt handler itself (since that may keep interrupts turned off too long). Instead a DPC is queued for slightly later, after the interrupt handler is done. In the second case, a thread has done a down on a semaphore or blocked on some other object, but with a timeout that has now expired. Again it is necessary for the interrupt handler to queue a DPC to avoid having it run during the clock interrupt handler.

If a thread has been made ready by this timeout, the scheduler will be run and if the newly runnable thread has higher priority, the current thread is preempted as in case 1.

Now we come to the actual scheduling algorithm. The Win32 API provides two APIs to influence thread scheduling. First, there is a call SetPriorityClass that sets the priority class of all the threads in the caller's process. The allowed values are: real-time, high, above normal, normal, below normal, and idle. The priority class determines the relative priorities of processes. (Starting with Windows Vista the process priority class can also be used by a process to temporarily mark itself as being *background*, meaning that it should not interfere with any other activity in the system.) Note that the priority class is established for the process, but affects the actual priority of every thread in the process by setting a base priority that each thread starts with when created.

The second Win32 API is SetThreadPriority. It sets the relative priority of a thread (possibly, but not necessarily, the calling thread) with respect to the priority class of its process. The allowed values are: time critical, highest, above normal, normal, below normal, lowest, and idle. Time critical threads get the highest non-real-time scheduling priority, while idle threads get the lowest, irrespective of the priority class. The other priority values adjust the base priority of a thread with respect to the normal value determined by the priority class (+2, +1, 0, −1, −2, respectively). The use of priority classes and relative thread priorities makes it easier for applications to decide what priorities to specify.

The scheduler works as follows. The system has 32 priorities, numbered from 0 to 31. The combinations of priority class and relative priority are mapped onto 32 absolute thread priorities according to the table of Fig. 11-27. The number in the table determines the thread's **base priority**. In addition, every thread has a **current priority**, which may be higher (but not lower) than the base priority and which we will discuss shortly.

		Win32 process class priorities					
		Real-time	High	Above Normal	Normal	Below Normal	Idle
Win32 thread priorities	Time critical	31	15	15	15	15	15
	Highest	26	15	12	10	8	6
	Above normal	25	14	11	9	7	5
	Normal	24	13	10	8	6	4
	Below normal	23	12	9	7	5	3
	Lowest	22	11	8	6	4	2
	Idle	16	1	1	1	1	1

Figure 11-27. Mapping of Win32 priorities to Windows priorities.

To use these priorities for scheduling, the system maintains an array of 32 lists of threads, corresponding to priorities 0 through 31 derived from the table of Fig. 11-27. Each list contains ready threads at the corresponding priority. The basic scheduling algorithm consists of searching the array from priority 31 down to priority 0. As soon as a nonempty list is found, the thread at the head of the queue is selected and run for one quantum. If the quantum expires, the thread goes to the end of the queue at its priority level and the thread at the front is chosen next. In other words, when there are multiple threads ready at the highest priority level, they run round robin for one quantum each. If no thread is ready, the processor is idled—that is, set to a low power state waiting for an interrupt to occur.

It should be noted that scheduling is done by picking a thread without regard to which process that thread belongs. Thus the scheduler does *not* first pick a process and then pick a thread in that process. It only looks at the threads. It does not consider which thread belongs to which process except to determine if it also needs to switch address spaces when switching threads.

To improve the scalability of the scheduling algorithms onto multiprocessors with a high number of processors, the scheduler tries hard not to have to take the lock that synchronizes access to the global array of priority lists. Instead it sees if it can directly dispatch a thread that is ready to run to the processor where it should run.

For each thread the scheduler maintains the notion of its **ideal processor** and attempts to schedule it on that processor whenever possible. This improves the performance of the system, as the data used by a thread are more likely to already be available in the cache belonging to its ideal processor. The scheduler is aware of multiprocessors in which each CPU has its own memory and which can execute programs out of any memory—but at a cost if the memory is not local. These systems are called **NUMA (NonUniform Memory Access)** machines. The scheduler tries to optimize thread placement on such machines. The memory manager tries to allocate physical pages in the NUMA node belonging to the ideal processor for threads when they page fault.

The array of queue headers is shown in Fig. 11-28. The figure shows that there are actually four categories of priorities: real-time, user, zero, and idle, which is effectively −1. These deserve some comment. Priorities 16–31 are called real time, and are intended to build systems that satisfy real-time constraints, such as deadlines. Threads with real-time priorities run before any of the threads with dynamic priorities, but not before DPCs and ISRs. If a real-time application wants to run on the system it may require device drivers that are careful not to run DPCs or ISRs for any extended time as they might cause the real-time threads to miss their deadlines.

Ordinary users may not run real-time threads. If a user thread ran at a higher priority than, say, the keyboard or mouse thread and got into a loop, the keyboard or mouse thread would never run, effectively hanging the system. The right to set

the priority class to real-time requires a special privilege to be enabled in the process' token. Normal users do not have this privilege.

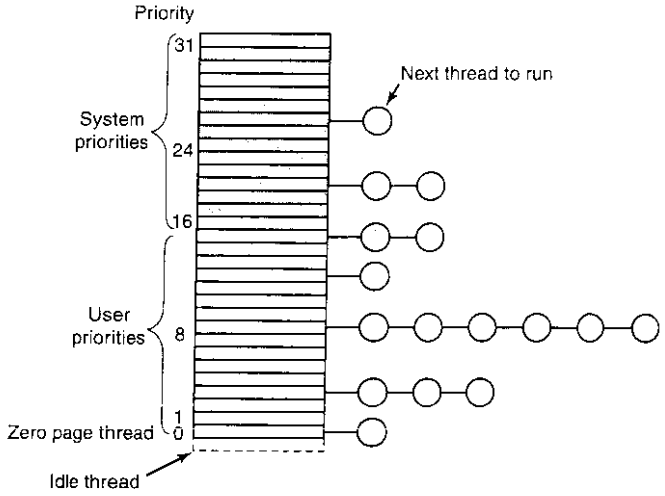

Figure 11-28. Windows Vista supports 32 priorities for threads.

Application threads normally run at priorities 1–15. By setting the process and thread priorities, an application can determine which threads get preference. The *ZeroPage* system threads run at priority 0 and convert free pages into pages of all zeroes. There is a separate *ZeroPage* thread for each real processor.

Each thread has a base priority based on the priority class of the process and the relative priority of the thread. But the priority used for determining which of the 32 lists a ready thread is queued on is determined by its current priority, which is normally the same as the base priority—but not always. Under certain conditions, the current priority of a non-real-time thread is boosted above the base priority by the kernel (but never above priority 15). Since the array of Fig. 11-28 is based on the current priority, changing this priority affects scheduling. No adjustments are ever made to real-time threads.

Let us now see when a thread's priority is raised. First, when an I/O operation completes and releases a waiting thread, the priority is boosted to give it a chance to run again quickly and start more I/O. The idea here is to keep the I/O devices busy. The amount of boost depends on the I/O device, typically 1 for a disk, 2 for a serial line, 6 for the keyboard, and 8 for the sound card.

Second, if a thread was waiting on a semaphore, mutex, or other event, when it is released, it gets boosted by 2 levels if it is in the foreground process (the process controlling the window to which keyboard input is sent) and 1 level otherwise. This fix tends to raise interactive processes above the big crowd at level 8.

Finally, if a GUI thread wakes up because window input is now available, it gets a boost for the same reason.

These boosts are not forever. They take effect immediately, and can cause rescheduling of the CPU. But if a thread uses all of its next quantum, it loses one priority level and moves down one queue in the priority array. If it uses up another full quantum, it moves down another level, and so on until it hits its base level, where it remains until it is boosted again.

There is one other case in which the system fiddles with the priorities. Imagine that two threads are working together on a producer-consumer type problem. The producer's work is harder, so it gets a high priority, say 12, compared to the consumer's 4. At a certain point, the producer has filled up a shared buffer and blocks on a semaphore, as illustrated in Fig. 11-29(a).

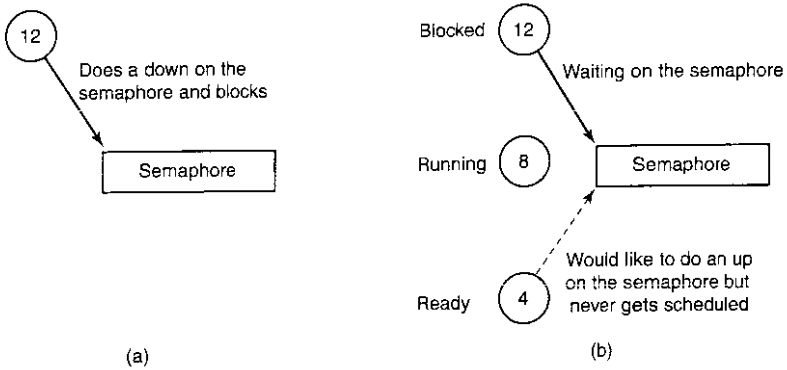

Figure 11-29. An example of priority inversion.

Before the consumer gets a chance to run again, an unrelated thread at priority 8 becomes ready and starts running, as shown in Fig. 11-29(b). As long as this thread wants to run, it will be able to, since it has a higher scheduling priority than the consumer, and the producer, though even higher, is blocked. Under these circumstances, the producer will never get to run again until the priority 8 thread gives up.

Windows solves this problem through what might be charitably called a big hack. The system keeps track of how long it has been since a ready thread ran last. If it exceeds a certain threshold, it is moved to priority 15 for two quanta. This may give it the opportunity to unblock the producer. After the two quanta are up, the boost is abruptly removed rather than decaying gradually. Probably a better solution would be to penalize threads that use up their quantum over and over by lowering their priority. After all, the problem was not caused by the starved thread, but by the greedy thread. This problem is well known under the name **priority inversion**.

An analogous problem happens if a priority 16 thread grabs a mutex and does not get a chance to run for a long time, starving more important system threads that are waiting for the mutex. This problem could have been prevented within the operating system by having a thread that needs a mutex for a short time just disable scheduling while it is busy. (On a multiprocessor, a spin lock should be used.)

Before leaving the subject of scheduling, it is worth saying a few words about the quantum. On Windows client systems the default is 20 msec. On Windows server systems it is 180 msec. The short quantum favors interactive users whereas the long quantum reduces context switches and thus provides better efficiency. These defaults can be increased manually by 2x, 4x, or 6x if desired.

One last patch to the scheduling algorithm says that when a new window becomes the foreground window, all of its threads get a longer quantum by an amount taken from the registry. This change gives them more CPU time, which usually translates to better user experience for the application whose window just moved to the foreground.

11.5 MEMORY MANAGEMENT

Windows Vista has an extremely sophisticated virtual memory system. It has a number of Win32 functions for using it, implemented by the memory manager—the largest component of the NTOS executive layer. In the following sections we will look at the fundamental concepts, the Win32 API calls, and finally the implementation.

11.5.1 Fundamental Concepts

In Windows Vista, every user process has its own virtual address space. For x86 machines, virtual addresses are 32 bits long, so each process has 4 GB of virtual address space. This can be allocated as either 2 GB of addresses for the user mode of each process, or Windows server systems can optionally configure the system to provide 3 GB for user mode. The other 2 GB (or 1 GB) is used by kernel mode. With x64 machines running in 64-bit mode, addresses can be either 32 or 64 bits. 32-bit addresses are used for processes that are running with *WOW64* for 32-bit compatibility. Since the kernel has plenty of available addresses, such 32-bit processes can actually get a full 4 GB of address space if they want. For both x86 and x64, the virtual address space is demand paged, with a fixed page size of 4 KB—though in some cases, as we will see shortly, 4-MB large pages are also used (by using a page directory only and bypassing the corresponding page table).

The virtual address space layouts for three x86 processes are shown in Fig. 11-30 in simplified form. The bottom and top 64 KB of each process' virtual

address space is normally unmapped This choice was made intentionally to help catch programming errors. Invalid pointers are often 0 or −1, so attempts to use them on Windows will cause an immediate trap instead of reading garbage or, worse yet, writing to an incorrect memory location.

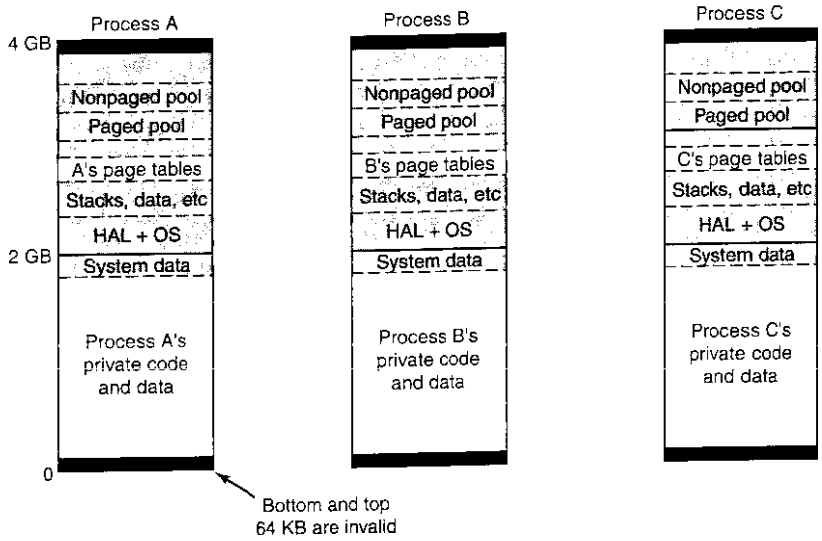

Figure 11-30. Virtual address space layout for three user processes on the x86. The white areas are private per process. The shaded areas are shared among all processes.

Starting at 64 KB comes the user's private code and data. This extends up to almost 2 GB. The upper 2 GB contains the operating system, including the code, data, and the paged and nonpaged pools. The upper 2 GB is the kernel's virtual memory, and is shared among all user processes, except for virtual memory data like the page tables and working set lists, which are per-process. Kernel virtual memory is only accessible while running in kernel mode. The reason for sharing the process' virtual memory with the kernel is that when a thread makes a system call, it traps into kernel mode and can continue running without changing the memory map. All that has to be done is switch to the thread's kernel stack. Because the process' user-mode pages are still accessible, the kernel-mode code can read parameters and access buffers without having to switch back and forth between address spaces or temporarily double-map pages into both. The trade-off here is less private address space per process in return for faster system calls.

Windows allows threads to attach themselves to other address spaces while running in the kernel. Attachment to an address space allows the thread to access all of the user-mode address space, as well as the portions of the kernel address

space that are specific to a process, such as the self-map for the page tables. Threads must switch back to their original address space before returning to user mode.

Virtual Address Allocation

Each page of virtual addresses can be in one of three states: invalid, reserved, or committed. An **invalid page** is not currently mapped to a memory section object and a reference to it causes a page fault that results in an access violation. Once code or data is mapped onto a virtual page, the page is said to be **committed**. A page fault on a committed page results in mapping the page containing the virtual address that caused the fault onto one of the pages represented by the section object or stored in the pagefile. It is often the case that this will require a physical page to be allocated, and I/O performed on the file represented by the section object to read in the data from disk. But page faults can also occur simply because the page table entry needs to be updated, as the physical page referenced is still cached in memory, in which case I/O is not required. These are called **soft faults** and we will discuss them in more detail shortly.

A virtual page can also be in the **reserved** state. A reserved virtual page is invalid, but has the property that those virtual addresses will never be allocated by the memory manager for another purpose. As an example, when a new thread is created, many pages of user-mode stack space are reserved in the process' virtual address space, but only one page is committed. As the stack grows the virtual memory manager will automatically commit additional pages under the covers, until the reservation is almost exhausted. The reserved pages function as guard pages to keep the stack from growing too far and overwriting other process data. Reserving all the virtual pages means that the stack can eventually grow to its maximum size without the risk that some of the contiguous pages of virtual address space needed for the stack might be given away for another purpose. In addition to the invalid, reserved, and committed attributes, pages also have other attributes, such as being readable, writable, and—in the case of AMD64-compatible processors—executable.

Pagefiles

An interesting trade-off occurs with assignment of backing store to committed pages that are not being mapped to specific files. These pages use the **pagefile**. The question is *how* and *when* to map the virtual page to a specific location in the pagefile. A simple strategy would be to assign each virtual page to a page in one of the paging files on disk at the time the virtual page was committed. This would guarantee that there was always a known place to write out each committed page should it be necessary to evict it from memory.

Windows uses a *just-in-time* strategy. Committed pages that are backed by the pagefile are not assigned space in the pagefile until the time that they have to be paged out. No disk space is allocated for pages that are never paged out. If the total virtual memory is less than the available physical memory, a pagefile is not needed at all. This is convenient for embedded systems based on Windows. It is also the way the system is booted, since pagefiles are not initialized until the first user-mode process, *smss.exe*, begins running.

With a preallocation strategy the total virtual memory in the system used for private data (stacks, heap, and copy-on-write code pages) is limited to the size of the pagefiles. With just-in-time allocation the total virtual memory can be almost as large as the combined size of the pagefiles and physical memory. With disks so large and cheap versus physical memory, the savings in space is not as significant as the increased performance that is possible.

With demand-paging, requests to read pages from disk need to be initiated right away, as the thread that encountered the missing page cannot continue until this *page-in* operation completes. The possible optimizations for faulting pages into memory involve attempting to prepage additional pages in the same I/O operation. However, operations that write modified pages to disk are not normally synchronous with the execution of threads. The just-in-time strategy for allocating pagefile space takes advantage of this to boost the performance of writing modified pages to the pagefile. Modified pages are grouped together and written in big chunks. Since the allocation of space in the pagefile does not happen until the pages are being written, the number of seeks required to write a batch of pages can be optimized by allocating the pagefile pages to be near each other, or even making them contiguous.

When pages stored in the pagefile are read into memory, they keep their allocation in the pagefile until the first time they are modified. If a page is never modified, it will go onto a special list of free physical pages, called the **standby list**, where it can be reused without having to be written back to disk. If it *is* modified, the memory manager will free the pagefile page and the only copy of the page will be in memory. The memory manager implements this by marking the page as read-only after it is loaded. The first time a thread attempts to write the page the memory manager will detect this situation and free the pagefile page, grant write access to the page, and then have the thread try again.

Windows supports up to 16 pagefiles, normally spread out over separate disks to achieve higher I/O bandwidth. Each one has an initial size and a maximum size it can grow to later if needed, but it is better to create these files to be the maximum size at system installation time. If it becomes necessary to grow a pagefile when the file system is much fuller, it is likely that the new space in the pagefile will be highly fragmented, reducing performance.

The operating system keeps track of which virtual page maps onto which part of which paging file by writing this information into the page table entries for the process for private pages, or into prototype page-table entries associated with the section object for shared pages. In addition to the pages that are backed by the pagefile, many pages in a process are mapped to regular files in the file system.

The executable code and read-only data in a program file (e.g., an EXE or DLL) can be mapped into the address space of whatever process is using it. Since these pages cannot be modified, they never need to be paged-out but the physical pages can just be immediately reused after the page table mappings are all marked as invalid. When the page is needed again in the future, the memory manager will read the page in from the program file.

Sometimes pages that start out as read-only end up being modified. For example, setting a breakpoint in the code when debugging a process, or fixing up code to relocate it to different addresses within a process, or making modifications to data pages that started out shared. In cases like these, Windows, like most modern operating systems, supports a type of page called **copy-on-write**. These pages start out as ordinary mapped pages, but when an attempt is made to modify any part of the page the memory manager makes a private, writable copy. It then updates the page table for the virtual page so that it points at the private copy, and has the thread retry the write—which will now succeed. If that copy later needs to be paged out, it will be written to the pagefile rather than the original file,

Besides mapping program code and data from EXE and DLL files, ordinary files can be mapped into memory, allowing programs to reference data from files without explicitly doing read and write operations. I/O operations are still needed, but they are provided implicitly by the memory manager using the section object to represent the mapping between pages in memory and the blocks in the files on disk.

Section objects do not have to refer to a file at all. They can refer to anonymous regions of memory. By mapping anonymous section objects into multiple processes, memory can be shared without having to allocate a file on disk. Since sections can be given names in the NT namespace, processes can rendezvous by opening section objects by name, as well as by duplicating handles to section objects between processes.

Addressing Large Physical Memories

Years ago, when 16-bit (or 20-bit) address spaces were standard, but machines had megabytes of physical memory, all kinds of tricks were thought up to allow programs to use more physical memory than fit in the address space. Often these tricks went under the name of **bank switching**, in which a program could substitute some block of memory above the 16-bit or 20-bit limit for a block of its own memory. When 32-bit machines were introduced, most desktop machines had only a few megabytes of physical memory. But as memory has gotten denser on integrated circuits, the amount of memory commonly available has grown dramatically. This first hits servers where applications often require more memory. The Xeon chips from Intel supported Physical Address Extensions (PAE)

which allowed physical memory to be addressed with 36 bits instead of 32, meaning that up to 64 GB of physical memory could be put on a single system. This is far more than the 2 or 3 GB that a single process can address with 32-bit user-mode virtual addresses, yet many big applications like SQL databases are designed to run in a single-process address space, so bank switching is back, now called **AWE (Address Windowing Extensions)** in Windows. This facility allows programs (running with the right privilege) to request the allocation of physical memory. The process requesting the allocation can then reserve virtual addresses and request the operating system to map regions of virtual pages directly to the physical pages. AWE is a stopgap solution until all servers use 64-bit addressing.

11.5.2 Memory Management System Calls

The Win32 API contains a number of functions that allow a process to manage its virtual memory explicitly. The most important of these functions are listed in Fig. 11-31. All of them operate on a region consisting either of a single page or a sequence of two or more pages that are consecutive in the virtual address space.

Win32 API function	Description
VirtualAlloc	Reserve or commit a region
VirtualFree	Release or decommit a region
VirtualProtect	Change the read/write/execute protection on a region
VirtualQuery	Inquire about the status of a region
VirtualLock	Make a region memory resident (i.e., disable paging for it)
VirtualUnlock	Make a region pageable in the usual way
CreateFileMapping	Create a file mapping object and (optionally) assign it a name
MapViewOfFile	Map (part of) a file into the address space
UnmapViewOfFile	Remove a mapped file from the address space
OpenFileMapping	Open a previously created file mapping object

Figure 11-31. The principal Win32 API functions for managing virtual memory in Windows.

The first four API functions are used to allocate, free, protect, and query regions of virtual address space. Allocated regions always begin on 64-KB boundaries to minimize porting problems to future architectures with pages larger than current ones. The actual amount of address space allocated can be less than 64 KB, but must be a multiple of the page size. The next two APIs give a process the ability to hardwire pages in memory so they will not be paged out and to undo this property. A real-time program might need pages with this property to avoid pagefaults to disk during critical operations, for example. A limit is enforced by

the operating system to prevent processes from getting too greedy. The pages actually can be removed from memory, but only if the entire process is swapped out. When it is brought back, all the locked pages are reloaded before any thread can start running again. Although not shown in Fig. 11-31, Windows Vista also has native API functions to allow a process to access the virtual memory of a different process over which it has been given control, that is, for which it has a handle (see Fig. 11-9).

The last four API functions listed are for managing memory-mapped files. To map a file, a file mapping object (see Fig. 11-23) must first be created with CreateFileMapping. This function returns a handle to the file mapping object (i.e., a section object) and optionally enters a name for it into the Win32 namespace so that other processes can use it too. The next two functions map and unmap views on section objects from a process' virtual address space. The last API can be used by a process to map share a mapping that another process created with CreateFileMapping, usually one created to map anonymous memory. In this way, two or more processes can share regions of their address spaces. This technique allows them to write in limited regions of each other's virtual memory.

11.5.3 Implementation of Memory Management

Windows Vista, on the x86, supports a single linear 4-GB demand-paged address space per process. Segmentation is not supported in any form. Theoretically, page sizes can be any power of 2 up to 64 KB. On the Pentium they are normally fixed at 4 KB. In addition, the operating system can use 4-MB pages to improve the effectiveness of the **TLB (Translation Lookaside Buffer)** in the processor's memory management unit. Use of 4-MB pages by the kernel and large applications significantly improves performance by improving the hit-rate for the TLB and reducing the number of times the page tables have to be walked to find entries that are missing from the TLB.

Unlike the scheduler, which selects individual threads to run and does not care much about processes, the memory manager deals entirely with processes and does not care much about threads. After all, processes, not threads, own the address space and that is what the memory manager is concerned with. When a region of virtual address space is allocated, as four of them have been for process *A* in Fig. 11-32, the memory manager creates a **VAD (Virtual Address Descriptor)** for it, listing the range of addresses mapped, the section representing the backing store file and offset where it is mapped, and the permissions. When the first page is touched, the directory of page tables is created and its physical address is inserted into the process object. An address space is completely defined by the list of its VADs. The VADs are organized into a balanced tree, so that the descriptor for a particular address can be found efficiently. This scheme supports sparse address spaces. Unused areas between the mapped regions use no resources (memory or disk) so they are essential free.

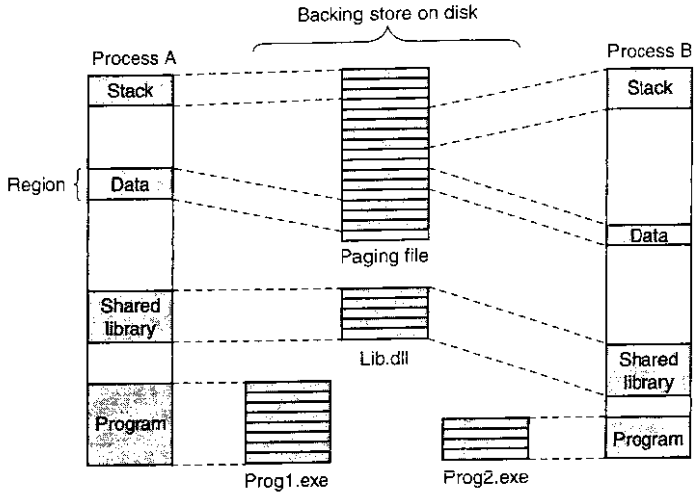

Figure 11-32. Mapped regions with their shadow pages on disk. The *lib.dll* file is mapped into two address spaces at the same time.

Page Fault Handling

When a process starts on Windows Vista, many of the pages mapping the program's EXE and DLL image files may already be in memory because they are shared with other processes. The writable pages of the images are marked *copy-on-write* so that they can be shared up to the point they need to be modified. If the operating system recognizes the EXE from a previous execution, it may have recorded the page reference pattern, using a technology Microsoft calls **SuperFetch**. SuperFetch attempts to prepage many of the needed pages even though the process has not faulted on them yet. This reduces the latency for starting up applications by overlapping the reading of the pages from disk with the execution of the initialization code in the images. It improves throughput to disk because it is easier for the disk drivers to organize the reads to reduce the seek time needed. Process prepaging is also used during boot of the system, when a background application moves to the foreground, and when restarting the system after hibernation.

Prepaging is supported by the memory manager, but implemented as a separate component of the system. The pages brought in are not inserted into the process' page table, but instead are inserted into the *standby list* from which they can quickly be inserted into the process as needed without accessing the disk.

Nonmapped pages are slightly different in that they are not initialized by reading from the file. Instead the first time a nonmapped page is accessed the memory

manager provides a new physical page, making sure the contents are all zeroes (for security reasons). On subsequent faults a nonmapped page may need to be found in memory or else must be read back from the pagefile.

Demand paging in the memory manager is driven by page faults. On each page fault, a trap to the kernel occurs. The kernel then builds a machine-independent descriptor telling what happened and passes this to the memory manager part of the executive. The memory manager then checks the access for validity. If the faulted page falls within a committed region, it looks up the address in the list of VADs and finds (or creates) the process page table entry. In the case of a shared page, the memory manager uses the prototype page table entry associated with the section object to fill in the new page table entry for the process page tables.

The format of the page table entries differs depending on the processor architecture. For the x86 and x64, the entries for a mapped page are shown in Fig. 11-33. If an entry is marked valid, its contents are interpreted by the hardware so that the virtual address can be translated into the correct physical page. Unmapped pages also have entries, but they are marked *invalid* and the hardware ignores the rest of the entry. The software format is somewhat different from the hardware format, and is determined by the memory manager. For example, for an unmapped page that must be allocated and zeroed before it may be used, that fact is noted in the page table entry.

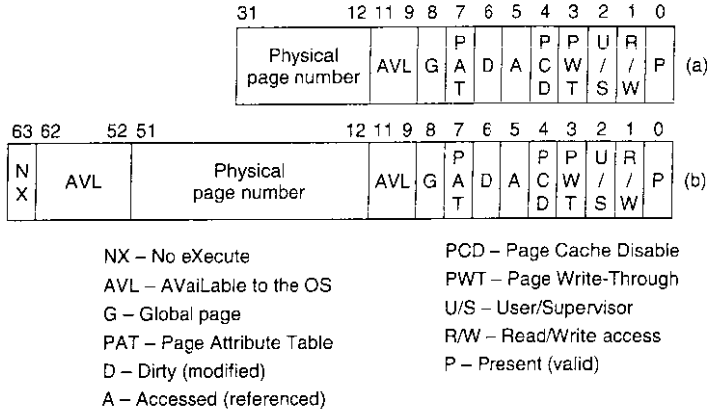

Figure 11-33. A page table entry (PTE) for a mapped page on the (a) Intel x86 and (b) AMD x64 architectures.

Two important bits in the page table entry are updated by the hardware directly. These are the access (A) and dirty (D) bits. These bits keep track of when a particular page mapping has been used to access the page and whether that access could have modified the page by writing it. This really helps the performance of the system because the memory manager can use the access bit to implement the

LRU (Least-Recently Used) style of paging. The LRU principle says that pages which have not been used the longest are the least likely to be used again soon. The access bit allows the memory manager to determine that a page has been accessed. The dirty bit lets the memory manager know that a page may have been modified. Or more significantly, that a page has *not* been modified. If a page has not been modified since being read from disk, the memory manager does not have to write the contents of the page to disk before using it for something else.

The x86 normally uses a 32-bit page table entry and the x64 uses a 64-bit page table entry, as shown in Fig. 11-33. The only difference in the fields is that the physical page number field is 30 bits instead of 20 bits. However, existing x64 processors support many fewer physical pages than can be represented by the architecture. The x86 also supports a special mode **PAE (Physical Address Extension)** which is used to allow the processor to access more than 4 GB of physical memory. The additional physical page frame bits require that the page table entries in PAE mode grow to also be 64 bits.

Each page fault can be considered as being in one of five categories:

1. The page referenced is not committed.
2. Attempted access to a page in violation of the permissions.
3. A shared copy-on-write page was about to be modified.
4. The stack needs to grow.
5. The page referenced is committed but not currently mapped in.

The first and second cases are due to programming errors. If a program attempts to use an address which is not supposed to have a valid mapping, or attempts an invalid operation (like attempting to write a read-only page) this is called an **access violation** and usually results in termination of the process. Access violations are often the result of bad pointers, including accessing memory that was freed and unmapped from the process.

The third case has the same symptoms as the second one (an attempt to write to a read-only page), but the treatment is different. Because the page has been marked as *copy-on-write*, the memory manager does not report an access violation, but instead makes a private copy of the page for the current process and then returns control to the thread that attempted to write the page. The thread will retry the write, which will now complete without causing a fault.

The fourth case occurs when a thread pushes a value onto its stack and crosses onto a page which has not been allocated yet. The memory manager is programmed to recognize this as a special case. As long as there is still room in the virtual pages reserved for the stack, the memory manager will supply a new physical page, zero it, and map it into the process. When the thread resumes running, it will retry the access and succeed this time around.

Finally, the fifth case is a normal page fault. However, it has several subcases. If the page is mapped by a file, the memory manager must search its data structures, such as the prototype page table associated with the section object to be sure that there is not already a copy in memory. If there is, say in another process or on the standby or modified page lists, it will just share it—perhaps marking it as copy-on-write if changes are not supposed to be shared. If there is not already a copy, the memory manager will allocate a free physical page and arrange for the file page to be copied in from disk.

When the memory manager can satisfy a page fault by finding the needed page in memory rather than reading it in from disk, the fault is classified as a **soft fault**. If the copy from disk is needed, it is a **hard fault**. Soft faults are much cheaper, and have little impact on application performance compared to hard faults. Soft faults can occur because a shared page has already been mapped into another process, or only a new zero page is needed, or the needed page was trimmed from the process' working set but is being requested again before it has had a chance to be reused.

When a physical page is no longer mapped by the page table in any process it goes onto one of three lists: free, modified, or standby. Pages that will never be needed again, such as stack pages of a terminating process, are freed immediately. Pages that may be faulted again go to either the modified list or the standby list, depending on whether or not the dirty bit was set for any of the page table entries that mapped the page since it was last read from disk. Pages in the modified list will be eventually written to disk, and then moved to the standby list.

The memory manager can allocate pages as needed using either the free list or the standby list. Before allocating a page and copying from disk, the memory manager always checks the standby and modified lists to see if it already has the page in memory. The prepaging scheme in Windows Vista converts future hard faults into soft faults by reading in the pages that are expected to be needed and pushing them onto the standby list. The memory manager itself does a small amount of ordinary prepaging by accessing groups of consecutive pages rather than single pages. The additional pages are immediately put on the standby list. This is not generally wasteful because the overhead in the memory manager is very much dominated by the cost of doing a single I/O. Reading a *cluster* of pages rather than a single page is negligibly more expensive.

The page table entries in Fig. 11-33 refer to physical page numbers, not virtual page numbers. To update page table (and page directory) entries, the kernel needs to use virtual addresses. Windows maps the page tables and page directories for the current process into kernel virtual address space using a **self-map** entry in the page directory, as shown in Fig. 11-34. By mapping a page directory entry to point at the page directory (the self-map), there are virtual addresses that can be used to refer to page directory entries (a) as well as page table entries (b). The self-map occupies 4 MB of kernel virtual addresses for every process (on the x86). Fortunately it is the same 4 MB. But 4 MB is not a big deal any more.

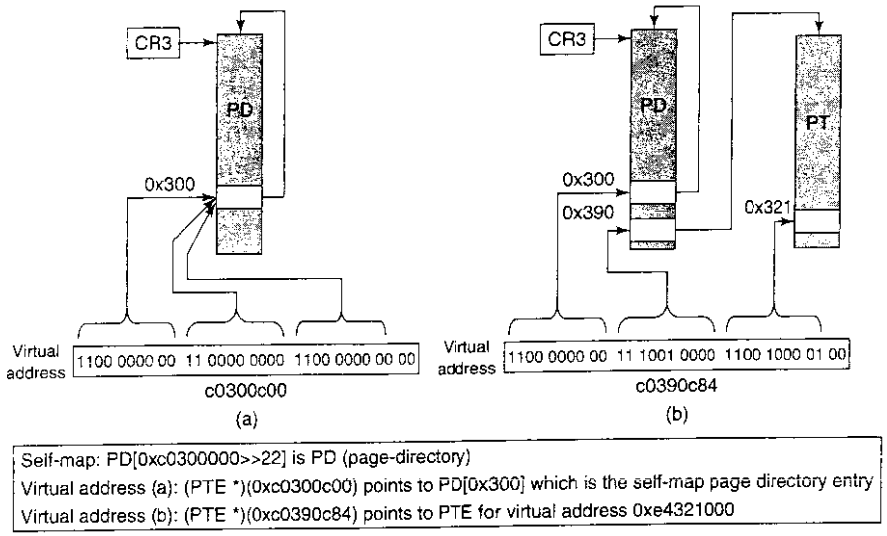

Figure 11-34. The Windows self-map entry used to map the physical pages of the page tables and page directory into kernel virtual addresses, for the x86.

The Page Replacement Algorithm

When the number of free physical memory pages starts to get low, the memory manager starts working to make more physical pages available by removing them from user-mode processes as well as the system process, which represents kernel-mode use of pages. The goal is to have the most important virtual pages present in memory and the others on disk. The trick is in determining what *important* means. In Windows this is answered by making heavy use of the working-set concept. Each process (*not* each thread) has a working set. This set consists of the mapped-in pages that are in memory and thus can be referenced without a page fault. The size and composition of the working set fluctuates as the process' threads run, of course.

Each process' working set is described by two parameters: the minimum size and the maximum size. These are not hard bounds, so a process may have fewer pages in memory than its minimum or (under certain circumstances) more than its maximum. Every process starts with the same minimum and maximum, but these bounds can change over time, or can be determined by the job object for processes contained in a job. The default initial minimum is in the range 20–50 pages and the default initial maximum is in the range 45–345 pages, depending on the total amount of physical memory in the system. The system administrator can change these defaults, however. While few home users will try, server admins might.

Working sets only come into play when the available physical memory is getting low in the system. Otherwise processes are allowed to consume memory as they choose, often far exceeding the working-set maximum. But when the system comes under **memory pressure**, the memory manager starts to squeeze processes back into their working sets, starting with processes that are over their maximum by the most. There are three levels of activity by the working-set manager, all of which is periodic based on a timer. New activity is added at each level:

1. **Lots of memory available:** Scan pages resetting access bits and using their values to represent the *age* of each page. Keep an estimate of the unused pages in each working set.
2. **Memory getting tight:** For any process with a significant proportion of unused pages, stop adding pages to the working set and start replacing the oldest pages whenever a new page is needed. The replaced pages go to the standby or modified list.
3. **Memory is tight:** Trim (i.e., reduce) working sets to be below their maximum by removing the oldest pages.

The working-set manager runs every second, called from the **balance set manager** thread. The working-set manager throttles the amount of work it does to keep from overloading the system. It also monitors the writing of pages on the modified list to disk to be sure that the list does not grow too large, waking the ModifiedPageWriter thread as needed.

Physical Memory Management

Above we mentioned three different lists of physical pages, the free list, the standby list, and the modified list. There is a fourth list which contains free pages that have been zeroed. The system frequently needs pages that contain all zeros. When new pages are given to processes, or the final partial page at the end of a file is read, a zero page is needed. It is time-consuming to write a page with zeros, so it is better to create zero pages in the background using a low-priority thread. There is also a fifth list used to hold pages that have been detected as having hardware errors (i.e., through hardware error detection).

All pages in the system are either referenced by a valid page table entry or are on one of these five lists, which are collectively called the **Page Frame Number Database (PFN database)**. Fig. 11-35 shows the structure of the PFN Database. The table is indexed by physical page frame number. The entries are fixed length, but different formats are used for different kinds of entries (e.g., shared versus private). Valid entries maintain the page's state and a count of how many page tables point to the page, so that the system can tell when the page is no longer in use. Pages that are in a working set tell which entry references them. There is also a pointer to the process page table that points to the page (for nonshared pages), or to the prototype page table (for shared pages).

Additionally there is a link to the next page on the list (if any), and various other fields and flags, such as *read in progress*, *write in progress*, an so on. To save space, the lists are linked together with fields referring to the next element by its index within the table rather than pointers. The table entries for the physical pages are also used to summarize the dirty bits found in the various page table entries that point to the physical page (i.e., because of shared pages). There is also information used to represent differences in memory pages on larger server systems which have memory that is faster from some processors than from others, namely NUMA machines.

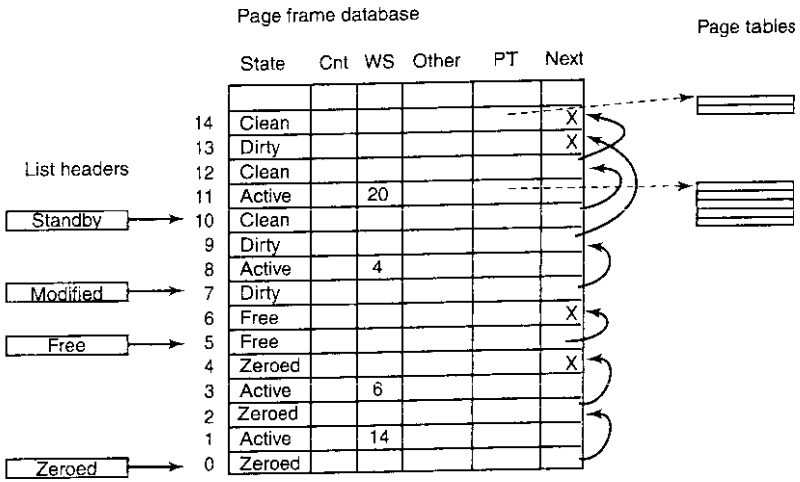

Figure 11-35. Some of the major fields in the page frame database for a valid page.

Pages are moved between the working sets and the various lists by the working-set manager and other system threads. Let us examine the transitions. When the working set manager removes a page from a working set, the page goes on the bottom of the standby or modified list, depending on its state of cleanliness. This transition is shown as (1) in Fig. 11-36.

Pages on both lists are still valid pages, so if a page fault occurs and one of these pages is needed, it is removed from the list and faulted back into the working set without any disk I/O (2). When a process exits, its nonshared pages cannot be faulted back to it, so the valid pages in its page table and any of its pages on the modified or standby lists go on the free list (3). Any pagefile space in use by the process is also freed.

Other transitions are caused by other system threads. Every 4 seconds the balance set manager thread runs and looks for processes all of whose threads have been idle for a certain number of seconds. If it finds any such processes, their

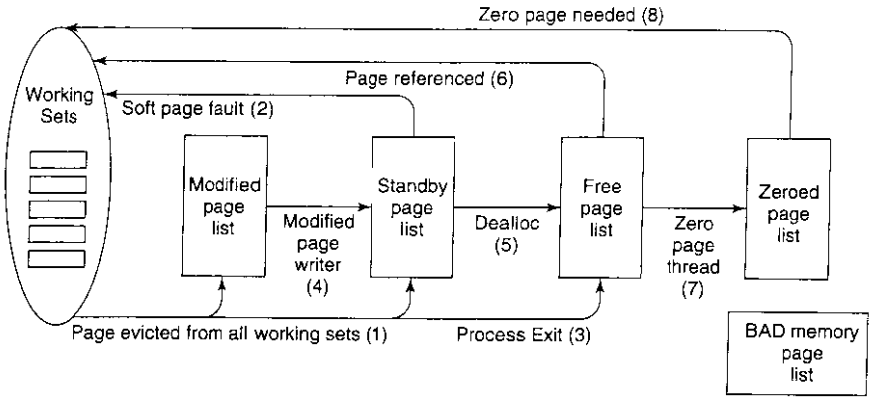

Figure 11-36. The various page lists and the transitions between them.

kernel stacks are unpinned from physical memory and their pages are moved to the standby or modified lists, also shown as (1).

Two other system threads, the **mapped page writer** and the **modified page writer**, wake up periodically to see if there are enough clean pages. If there are not, they take pages from the top of the modified list, write them back to disk, and then move them to the standby list (4). The former handles writes to mapped files and the latter handles writes to the pagefiles. The result of these writes is to transform modified (dirty) pages into standby (clean) pages.

The reason for having two threads is that a mapped file might have to grow as a result of the write, and growing it requires access to on-disk data structures to allocate a free disk block. If there is no room in memory to bring them in when a page has to be written, a deadlock could result. The other thread can solve the problem by writing out pages to a paging file.

The other transitions in Fig. 11-36 are as follows. If a process unmaps a page, the page is no longer associated with a process and can go on the free list (5), except for the case that it is shared. When a page fault requires a page frame to hold the page about to be read in, the page frame is taken from the free list (6), if possible. It does not matter that the page may still contain confidential information because it is about to be overwritten in its entirety.

The situation is different when a stack grows. In that case, an empty page frame is needed and the security rules require the page to contain all zeros. For this reason, another kernel system thread, the **ZeroPage thread**, runs at the lowest priority (see Fig. 11-28), erasing pages that are on the free list and putting them on the zeroed page list (7). Whenever the CPU is idle and there are free pages, they might as well be zeroed since a zeroed page is potentially more useful than a free page and it costs nothing to zero the page when the CPU is idle.

The existence of all these lists leads to some subtle policy choices. For example, suppose that a page has to be brought in from disk and the free list is empty. The system is now forced to choose between taking a clean page from the standby list (which might otherwise have been faulted back in later) or an empty page from the zeroed page list (throwing away the work done in zeroing it). Which is better?

The memory manager has to decide how aggressively the system threads should move pages from the modified list to the standby list. Having clean pages around is better than having dirty pages around (since they can be reused instantly), but an aggressive cleaning policy means more disk I/O and there is some chance that a newly cleaned page may be faulted back into a working set and dirtied again anyway. In general, Windows resolves these kinds of trade-offs through algorithms, heuristics, guesswork, historical precedent, rules of thumb, and administrator-controlled parameter settings.

All in all, memory management is a highly complex executive component with many data structures, algorithms, and heuristics. It attempts to be largely self tuning, but there are also many knobs that administrators can tweak to affect system performance. A number of these knobs and the associated counters can be viewed using tools in the various tool kits mentioned earlier. Probably the most important thing to remember here is that memory management in real systems is a lot more than just one simple paging algorithm like clock or aging.

11.6 CACHING IN WINDOWS VISTA

The Windows cache improves the performance of file systems by keeping recently and frequently used regions of files in memory. Rather than cache physical addressed blocks from the disk, the cache manager manages virtually addressed blocks, that is, regions of files. This approach fits well with the structure of the native NT File System (NTFS), as we will see in Sec. 11.8. NTFS stores all of its data as files, including the file system metadata.

The cached regions of files are called *views* because they represent regions of kernel virtual addresses that are mapped onto file system files. Thus the actual management of the physical memory in the cache is provided by the memory manager. The role of the cache manager is to manage the use of kernel virtual addresses for views, arrange with the memory manager to *pin* pages in physical memory, and provide interfaces for the file systems.

The Windows cache manager facilities are shared among all the file systems. Because the cache is virtually addressed according to individual files, the cache manager is easily able to perform read-ahead on a per-file basis. Requests to access cached data come from each file system. Virtual caching is convenient because the file systems do not have to first translate file offsets into physical block

numbers before requesting a cached file page. Instead the translation happens later when the memory manager calls the file system to access the page on the disk.

Besides management of the kernel virtual address and physical memory resources used for caching, the cache manager also has to coordinate with file systems regarding issues like coherency of views, flushing to disk, and correct maintenance of the end-of-file marks—particularly as files expand. One of the most difficult aspects of a file to manage between the file system, the cache manager, and the memory manager is the offset of the last byte in the file, called the *ValidDataLength*. If a program writes past the end of the file, the blocks that were skipped have to be filled with zeros, and for security reasons it is critical that the ValidDataLength recorded in the file metadata not allow access to uninitialized blocks, so the zero blocks have to be written to disk before the metadata is updated with the new length. While it is expected that if the system crashes, some of the blocks in the file might not have been updated from memory, it is not acceptable that some of the blocks might contain data previously belonging to other files.

Let us now examine how the cache manager works. When a file is referenced, the cache manager maps a 256-KB chunk of kernel virtual address space onto the file. If the file is larger than 256 KB, only a portion of the file is mapped at a time. If the cache manager runs out of 256-KB chunks of virtual address space, it must unmap an old file before mapping in a new one. Once a file is mapped, the cache manager can satisfy requests for its blocks by just copying from kernel virtual address space to the user buffer. If the block to be copied is not in physical memory, a page fault will occur and the memory manager will satisfy the fault in the usual way. The cache manager is not even aware of whether the block was in memory or not. The copy always succeeds.

The cache manager also works for pages that are mapped into virtual memory and accessed with pointers rather than being copied between kernel and user-mode buffers. When a thread accesses a virtual address mapped to a file and a page fault occurs, the memory manager may in many cases be able to satisfy the access as a soft fault. It does not need to access the disk because it finds that the page is already in physical memory because it is mapped by the cache manager.

Caching is not appropriate for all applications. Large enterprise applications, like SQL, prefer to manage their own caching and I/O. Windows allows files to be opened for **unbuffered I/O** which bypasses the cache manager. Historically, such applications would rather trade off operating systems caching for an increased user-mode virtual address space, so the system supports a configuration where it can be rebooted to provide 3 GB of user-mode address space to applications that request it, using only 1 GB for kernel mode instead of the conventional 2-GB/2-GB split. This mode of operation (called /3GB mode after the boot switch that enables it) is not as flexible as in some operating systems, which allow the user/kernel address space split to be adjusted with far more granularity. When

Windows runs in /3GB mode, only half the number of kernel virtual addresses are available. The cache manager adjusts by mapping far fewer files, which is what SQL would prefer anyway.

Windows Vista introduced a brand new form of caching in the system, called **ReadyBoost**, which is distinct from the cache manager. Users can plug flash memory sticks into USB or other ports and arrange for the operating system to use the flash memory as a write-through cache. The flash memory introduces a new layer in the memory hierarchy, which is particularly useful for increasing the amount of read-caching of disk data that is possible. Reads from flash memory are relatively fast, though not as fast as the Dynamic RAM used for normal memory. With flash being relatively inexpensive versus high-speed DRAM, this feature in Vista allows the system to get higher performance with less DRAM—and all without having to open the computer's case.

ReadyBoost compresses the data (typically 2x) and encrypts it. The implementation uses a filter driver that processes the I/O requests sent to the volume manager by the file system. Similar technology, named **ReadyBoot**, is used to speed up boot-time on some Windows Vista systems by caching data to flash. These technologies have less impact on systems with 1 GB or more of DRAM. Where they really help is on systems trying to run Windows Vista with only 512 MB of DRAM. Near 1 GB the system has enough memory that demand paging is infrequent enough that disk I/O can keep up for most usage scenarios.

The write-through approach is important to minimize data loss should a flash stick be unplugged, but future PC hardware may incorporate flash memory directly on the parentboard. Then the flash can be used without write-through, allowing the system to cache critical data that needs to persist across a system crash without having to spin up the disk. This is good not just for performance, but also to reduce power consumption (and thus increase battery life on notebooks) because the disk is spinning less. Some notebooks today go all the way and eliminate an electromechanical disk altogether, instead using lots of flash memory.

11.7 INPUT/OUTPUT IN WINDOWS VISTA

The goals of the Windows I/O manager are to provide a fundamentally extensive and flexible framework for efficiently handling a very wide variety of I/O devices and services, support automatic device discovery and driver installation (plug-and-play) and power management for devices and the CPU—all using a fundamentally asynchronous structure that allows computation to overlap with I/O transfers. There are many hundreds of thousands of devices that work with Windows Vista. For a large number of common devices it is not even necessary to install a driver, because there is already a driver that shipped with the Windows operating system. But even so, counting all the revisions, there are almost a million distinct driver binaries that run on Windows Vista. In the following sections we will examine some of the issues relating to I/O.

11.7.1 Fundamental Concepts

The I/O manager is on intimate terms with the plug-and-play manager. The basic idea behind plug and play is that of an enumerable bus. Many buses, including PC Card, PCI, PCI-x, AGP, USB, IEEE 1394, EIDE, and SATA, have been designed so that the plug-and-play manager can send a request to each slot and ask the device there to identify itself. Having discovered what is out there, the plug-and-play manager allocates hardware resources, such as interrupt levels, locates the appropriate drivers, and loads them into memory. As each driver is loaded, a **driver object** is created for it. And then for each device, at least one device object is allocated. For some buses, such as SCSI, enumeration happens only at boot time, but for other buses, such as USB, it can happen at any time, requiring close cooperation between the plug-and-play manager, the bus drivers (which actually do the enumerating), and the I/O manager.

In Windows, all the file systems, antivirus filters, volume managers, network protocol stacks, and even kernel services that have no associated hardware are implemented using I/O drivers. The system configuration must be set to cause some of these drivers to load, because there is no associated device to enumerate on the bus. Others, like the file systems, are loaded by special code that detects they are needed, such as the file system recognizer that looks at a raw volume and deciphers what type of file system format it contains.

An interesting feature of Windows is its support for **dynamic disks**. These disks may span multiple partitions and even multiple disks and may be reconfigured on the fly, without even having to reboot. In this way, logical volumes are no longer constrained to a single partition or even a single disk so that a single file system may span multiple drives in a transparent way.

The I/O to volumes can be filtered by a special Windows driver to produce **Volume Shadow Copies**. The filter driver creates a snapshot of the volume which can be separately mounted and represents a volume at a previous point in time. It does this by keeping track of changes after the snapshot point. This is very convenient for recovering files that were accidentally deleted, or traveling back in time to see the state of a file at periodic snapshots made in the past.

But shadow copies are also valuable for making accurate backups of server systems. The system works with server applications to have them reach a convenient point for making a clean backup of their persistent state on the volume. Once all the applications are ready, the system initializes the snapshot of the volume and then tells the applications that they can continue. The backup is made of the volume state at the point of the snapshot. And the applications were only blocked for a very short time rather than having to go offline for the duration of the backup.

Applications participate in the snapshot process, so the backup reflects a state that is easy to recover in case there is a future failure. Otherwise the backup might still be useful, but the state it captured would look more like the state if the

system had crashed. Recovering from a system at the point of a crash can be more difficult or even impossible, since crashes occur at arbitrary times in the execution of the application. *Murphy's Law* says that crashes are most likely to occur at the worst possible time, that is, when the application data is in a state where recovery is impossible.

Another aspect of Windows is its support for asynchronous I/O. It is possible for a thread to start an I/O operation and then continue executing in parallel with the I/O. This feature is especially important on servers. There are various ways the thread can find out that the I/O has completed. One is to specify an event object at the time the call is made and then wait on it eventually. Another is to specify a queue to which a completion event will be posted by the system when the I/O is done. A third is to provide a callback procedure that the system calls when the I/O has completed. A fourth is to poll a location in memory that the I/O manager updates when the I/O completes.

The final aspect that we will mention is prioritized I/O, which was introduced in Windows Vista. I/O priority is determined by the priority of the issuing thread, or can be explicitly set. There are five priorities specified: *critical*, *high*, *normal*, *low*, and *very low*. Critical is reserved for the memory manager to avoid deadlocks that could otherwise occur when the system experiences extreme memory pressure. Low and very low priorities are used by background processes, like the disk defragmentation service and spyware scanners and desktop search, which are attempting to avoid interfering with normal operations of the system. Most I/O gets normal priority, but multimedia applications can mark their I/O as high to avoid glitches. Multimedia apps can alternatively use **bandwidth reservation** to request guaranteed bandwidth to access time-critical files, like music or video. The I/O system will provide the application with the optimal transfer size and the number of outstanding I/O operations that should be maintained to allow the I/O system to achieve the requested bandwidth guarantee.

11.7.2 Input/Output API Calls

The system call APIs provided by the I/O manager are not very different from those offered by most operating systems. The basic operations are open, read, write, ioctl, and close, but there are also plug-and-play and power operations, operations for setting parameters, flushing system buffers, and so on. At the Win32 layer these APIs are wrapped by interfaces that provide higher-level operations specific to particular devices. At the bottom though, these wrappers open devices and perform these basic types of operations. Even some metadata operations, such as file rename, are implemented without specific system calls. They just use a special version of the ioctl operations. This will make more sense when we explain the implementation of I/O device stacks and the use of I/O request packets (IRPs) by the I/O manager.

I/O system call	Description
NtCreateFile	Open new or existing files or devices
NtReadFile	Read from a file or device
NtWriteFile	Write to a file or device
NtQueryDirectoryFile	Request information about a directory, including files
NtQueryVolumeInformationFile	Request information about a volume
NtSetVolumeInformationFile	Modify volume information
NtNotifyChangeDirectoryFile	Complete when any file in the directory or sub-tree is modified
NtQueryInformationFile	Request information about a file
NtSetInformationFile	Modify file information
NtLockFile	Lock a range of bytes in a file
NtUnlockFile	Remove a range lock
NtFsControlFile	Miscellaneous operations on a file
NtFlushBuffersFile	Flush in-memory file buffers to disk
NtCancelIoFile	Cancel outstanding I/O operations on a file
NtDeviceIoControlFile	Special operations on a device

Figure 11-37. Native NT API calls for performing I/O.

The native NT I/O system calls, in keeping with the general philosophy of Windows, take numerous parameters, and include many variations. Fig. 11-37 lists the primary system call interfaces to the I/O manager. NtCreateFile is used to open existing or new files. It provides security descriptors for new files, a rich description of the access rights requested, and gives the creator of new files some control over how blocks will be allocated. NtReadFile and NtWriteFile take a file handle, buffer, and length. They also take an explicit file offset, and allow a key to be specified for accessing locked ranges of bytes in the file. Most of the parameters are related to specifying which of the different methods to use for reporting completion of the (possibly asynchronous) I/O, as described above.

NtQueryDirectoryFile is an example of a standard paradigm in the executive where various Query APIs exist to access or modify information about specific types of objects. In this case it is file objects that refer to directories. A parameter specifies what type of information is being requested, such as a list of the names in the directory or detailed information about each file that is needed for an extended directory listing. Since this is really an I/O operation, all the standard ways of reporting that the I/O completed are supported. NtQueryVolumeInformationFile is like the directory query operation, but expects a file handle which represents an open volume which may or may not contain a file system. Unlike for directories, there are parameters than can be modified on volumes, and thus there is a separate API NtSetVolumeInformationFile.

NtNotifyChangeDirectoryFile is an example of an interesting NT paradigm. Threads can do I/O to determine whether any changes occur to objects (mainly file system directories, as in this case, or registry keys). Because the I/O is asynchronous the thread returns and continues, and is only notified later when something is modified. The pending request is queued in the file system as an outstanding I/O operation using an I/O Request Packet (IRP). Notifications are problematic if you want to remove a file system volume from the system, because the I/O operations are pending. So Windows supports facilities for canceling pending I/O operations, including support in the file system for forcibly dismounting a volume with pending I/O.

NtQueryInformationFile is the file-specific version of the system call for directories. It has a companion system call, NtSetInformationFile. These interfaces access and modify all sorts of information about file names, file features like encryption and compression and sparseness, and other file attributes and details, including looking up the internal file id or assigning a unique binary name (object id) to a file.

These system calls are essentially a form of ioctl specific to files. The set operation can be used to rename or delete a file. But note that they take handles, not file names, so a file first must be opened before being renamed or deleted. They can also be used to rename the alternative data streams on NTFS (see Sec. 11.8).

Separate APIs, NtLockFile and NtUnlockFile exist to set and remove byte-range locks on files. NtCreateFile allows access to an entire file to be restricted by using a sharing mode. An alternative is these lock APIs, which apply mandatory access restrictions to a range of bytes in the file. Reads and writes must supply a *key* matching the key provided to NtLockFile in order to operate on the locked ranges.

Similar facilities exist in UNIX, but there it is discretionary whether applications heed the range locks. NtFsControlFile is much like the preceding Query and Set operations, but is a more generic operation aimed at handling file-specific operations that do not fit within the other APIs. For example, some operations are specific to a particular file system.

Finally, there are miscellaneous calls such as NtFlushBuffersFile. Like the UNIX sync call, it forces file system data to be written back to disk, NtCancelIoFile to cancel outstanding I/O requests for a particular file, and NtDeviceIoControlFile which implements ioctl operations for devices. The list of operations is actually much longer. There are system calls for deleting files by name, and querying the attributes of a specific file—but these are just wrappers around the other I/O manager operations we have listed and did not really need to be implemented as separate system calls. There are also system calls for dealing with **I/O completion ports**, a queuing facility in Windows that helps multithreaded servers make efficient use of asynchronous I/O operations by readying threads by demand and reducing the number of context switches required to service I/O on dedicated threads.

11.7.3 Implementation of I/O

The Windows I/O system consists of the plug-and-play services, the power manager, the I/O manager, and the device driver model. Plug-and-play detects changes in hardware configuration and builds or tears down the device stacks for each device, as well as causing the loading and unloading of device drivers. The power manager adjusts the power state of the I/O devices to reduce system power consumption when devices are not in use. The I/O manager provides support for manipulating I/O kernel objects, and IRP-based operations like IoCallDrivers and IoCompleteRequest. But most of the work required to support Windows I/O is implemented by the device drivers themselves.

Device Drivers

To make sure that device drivers work well with the rest of Windows Vista, Microsoft has defined the **WDM (Windows Driver Model)** that device drivers are expected to conform with. The WDM was designed to work with both Windows 98 and NT-based Windows beginning with Windows 2000, allowing carefully written drivers to be compatible with both systems. There is a development kit (the Windows Driver Kit) that is designed to help driver writers produce conformant drivers. Most Windows drivers start out by copying an appropriate sample driver and modifying it.

Microsoft also provides a **driver verifier** which validates many of the actions of drivers to be sure that they conform to the WDM requirements for the structure and protocols for I/O requests, memory management, and so on. The verifier ships with the system, and administrators can control it by running *verifier.exe*, which allows them to configure which drivers are to be checked and how extensive (i.e., expensive) the checks should be.

Even with all the support for driver development and verification, it is still very difficult to write even simple drivers in Windows, so Microsoft has built a system of wrappers called the **WDF (Windows Driver Foundation)** that runs on top of WDM and simplifies many of the more common requirements, mostly related to correct interaction with power management and plug-and-play operations.

To further simplify driver writing, as well as increase the robustness of the system, WDF includes the **UMDF (User-Mode Driver Framework)** for writing drivers as services that execute in processes. And there is the **KMDF (Kernel-Mode Driver Framework)** for writing drivers as services that execute in the kernel, but with many of the details of WDM made automagical. Since underneath it is the WDM that provides the driver model, that is what we will focus on in this section.

Devices in Windows are represented by device objects. Device objects are also used to represent hardware, such as buses, as well as software abstractions like file systems, network protocol engines, and kernel extensions, like antivirus

filter drivers. All these are organized by producing what Windows calls a *device stack*, as was previously shown in Fig. 11-16.

I/O operations are initiated by the I/O manager calling an executive API IoCallDriver with pointers to the top device object and to the IRP representing the I/O request. This routine finds the driver object associated with the device object. The operation types that are specified in the IRP generally correspond to the I/O manager system calls described above, such as CREATE, READ, and CLOSE.

Fig. 11-38 shows the relationships for a single level of the device stack. For each of these operations a driver must specify an entry point. IoCallDriver takes the operation type out of the IRP, uses the device object at the current level of the device stack to find the driver object, and indexes into the driver dispatch table with the operation type to find the corresponding entry point into the driver. The driver is then called and passed the device object and the IRP.

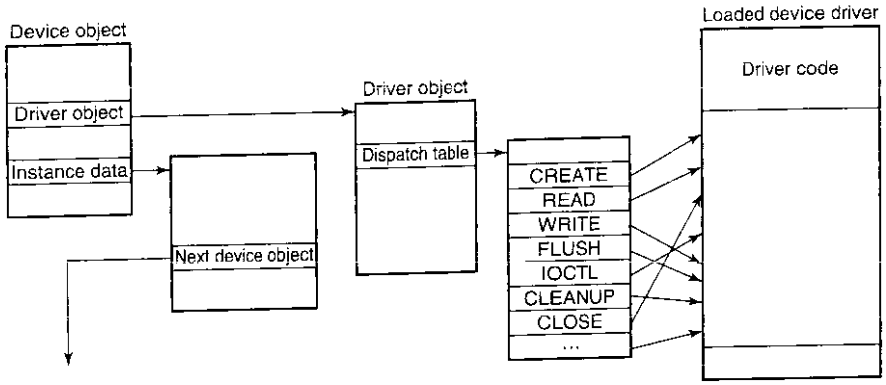

Figure 11-38. A single level in a device stack.

Once a driver has finished processing the request represented by the IRP, it has three options. It can call IoCallDriver again, passing the IRP and the next device object in the device stack. It can declare the I/O request to be completed and return to its caller. Or it can queue the IRP internally and return to its caller, having declared that the I/O request is still pending. This latter case results in an asynchronous I/O operation, at least if all the drivers above in the stack agree and also return to their callers.

I/O Request Packets

Fig. 11-39 shows the major fields in the IRP. The bottom of the IRP is a dynamically sized array containing fields that can be used by each driver for the device stack handling the request. These *stack* fields also allow a driver to specify

the routine to call when completing an I/O request. During completion each level of the device stack is visited in reverse order, and the completion routine assigned by each driver is called in turn. At each level the driver can continue to complete the request or decide there is still more work to do and leave the request pending, suspending the I/O completion for the time being.

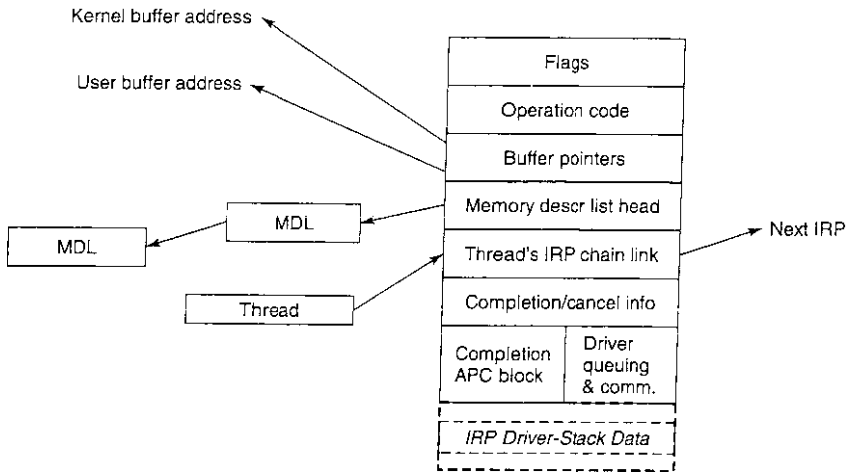

Figure 11-39. The major fields of an I/O Request Packet.

When allocating an IRP, the I/O manager has to know how deep the particular device stack is so that it can allocate a sufficiently large IRP. It keeps track of the stack depth in a field in each device object as the device stack is formed. Note that there is no formal definition of what the next device object is in any stack. That information is held in private data structures belonging to the previous driver on the stack. In fact the stack does not really have to be a stack at all. At any layer a driver is free to allocate new IRPs, continue to use the original IRP, send an I/O operation to a different device stack, or even switch to a system worker thread to continue execution.

The IRP contains flags, an operation code for indexing into the driver dispatch table, buffer pointers for possibly both kernel and user buffers, and a list of **MDL**s (**Memory Descriptor Lists**) which are used to describe the physical pages represented by the buffers, that is, for DMA operations. There are fields used for cancellation and completion operations. The fields in the IRP that are used to queue the IRP to devices while it is being processed are reused when the I/O operation has finally completed to provide memory for the APC control object used to call the I/O manager's completion routine in the context of the original thread. There is also a link field used to link all the outstanding IRPs to the thread that initiated them.

Device Stacks

A driver in Windows Vista may do all the work by itself, as the printer driver does in Fig. 11-40. On the other hand, drivers may also be stacked, which means that a request may pass through a sequence of drivers, each doing part of the work. Two stacked drivers are also illustrated in Fig. 11-40.

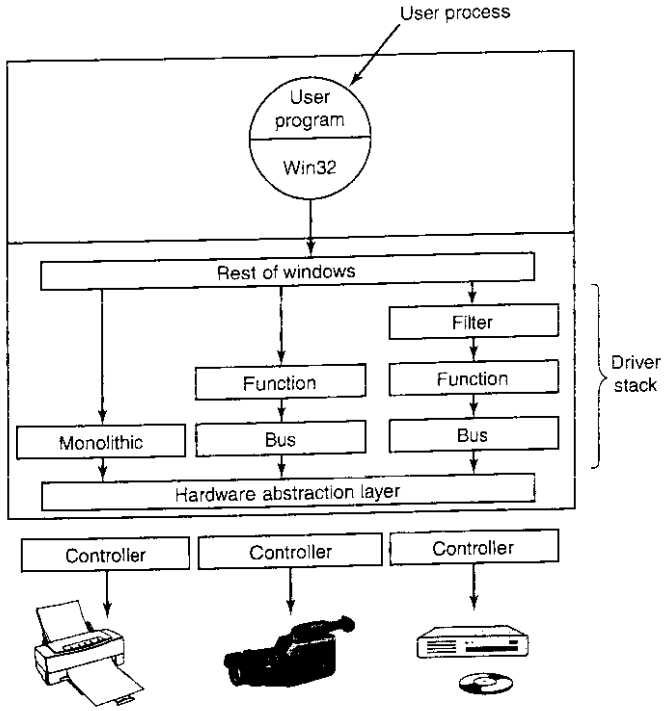

Figure 11-40. Windows allows drivers to be stacked to work with a specific instance of a device. The stacking is represented by device objects.

One common use for stacked drivers is to separate the bus management from the functional work of controlling the device. Bus management on the PCI bus is quite complicated on account of many kinds of modes and bus transactions. By separating this work from the device-specific part, driver writers are freed from learning how to control the bus. They can just use the standard bus driver in their stack. Similarly, USB and SCSI drivers have a device-specific part and a generic part, with common drivers being supplied by Windows for the generic part.

Another use of stacking drivers is to be able to insert **filter drivers** into the stack. We have already looked at the use of file system filter drivers, which are

inserted above the file system. Filter drivers are also used for managing physical hardware. A filter driver performs some transformation on the operations as the IRP flows down the device stack, as well as during the completion operation with the IRP flows back up through the completion routines each driver specified. For example, a filter driver could compress data on the way to the disk or encrypt data on the way to the network. Putting the filter here means that neither the application program nor the true device driver have to be aware of it, and it works automatically for all data going to (or coming from) the device.

Kernel-mode device drivers are a serious problem for the reliability and stability of Windows. Most of the kernel crashes in Windows are due to bugs in device drivers. Because kernel-mode device drivers all share the same address space with the kernel and executive layers, errors in the drivers can corrupt system data structures, or worse. Some of these bugs are due to the astonishingly large numbers of device drivers that exist for Windows, or to the development of drivers by less experienced system programmers. The bugs are also due to the large amount of detail involved in writing a correct driver for Windows.

The I/O model is powerful and flexible, but all I/O is fundamentally asynchronous, so race conditions can abound. Windows 2000 added the plug-and-play and power management facilities from the Win9x systems to the NT-based Windows for the first time. This put a large number of requirements on drivers to deal correctly with devices coming and going while I/O packets are in the middle of being processed. Users of PCs frequently dock/undock devices, close the lid and toss notebooks into briefcases, and generally do not worry about whether the little green activity light happens to still be on. Writing device drivers that function correctly in this environment can be very challenging, which is why Windows Driver Foundation was developed to simplify the Windows Driver Model.

The **power manager** rides herd on power usage throughout the system. Historically management of power consumption consisted of shutting off the monitor display and stopping the disk drives from spinning. But the issue is rapidly becoming more complicated due to requirements for extending how long notebooks can run on batteries, and energy conservation concerns related to desktop computers being left on all the time and the high cost of supplying power to the huge server farms that exist today (companies like Microsoft and Google are building their server farms next to hydroelectric facilities to get low rates).

Newer power management facilities include reducing the power consumption of components when the system is not in use by switching individual devices to standby states, or even powering them off completely using *soft* power switches. Multiprocessors shut down individual CPUs when they are not needed, and even the clock rates of the running CPUs can be adjusted downward to reduce power consumption. When a processor is idle, its power consumption is also reduced since it needs to do nothing except wait for an interrupt to occur.

Windows supports a special mode of shutdown called **hibernation** which copies all of physical memory to disk and then reduces power consumption to a

small trickle (notebooks can run weeks in a hibernated state) with little battery drain. Because all the memory state is written to disk, you can even replace the battery on a notebook while it is hibernated. When the system reboots after hibernation it restores the saved memory state (and reinitializes the devices). This brings the computer back into the same state it was before hibernation, without having to logon again and start up all the applications and services that were running. Even though Windows tries to optimize this process (including ignoring unmodified pages backed by disk already and compressing other memory pages to reduce the amount of I/O required), it can still take many seconds to hibernate a notebook or desktop system with gigabytes of memory.

An alternative to hibernation is a mode called **standby mode** where the power manager reduces the entire system to the lowest power state possible, using just enough power to the refresh the dynamic RAM. Because memory does not need to be copied to disk, this is much faster than hibernation. But standby is not as reliable because work will be lost if a desktop system loses power, or the battery is swapped on a notebook, or due to bugs in various device drivers which reduce devices to low-power state but are then unable to reinitialize them. In developing Windows Vista, Microsoft expended a lot of effort improving the operation of standby mode, with the cooperation of many in the hardware device community. They also stopped the practice of allowing applications to veto the system going into standby mode (which sometimes resulted in superheated notebooks for inattentive users who tossed them in briefcases without waiting for the light to blink).

There are many books available about the Windows Driver Model and the newer Windows Driver Foundation (Cant, 2005; Oney, 2002; Orwick & Smith, 2007; and Viscarola et al., 2007).

11.8 THE WINDOWS NT FILE SYSTEM

Windows Vista supports several file systems, the most important of which are **FAT-16**, **FAT-32**, and **NTFS (NT File System)**. FAT-16 is the old MS-DOS file system. It uses 16-bit disk addresses, which limits it to disk partitions no larger than 2 GB. Mostly it is used to access floppy disks, for customers that still use them. FAT-32 uses 32-bit disk addresses and supports disk partitions up to 2 TB. There is no security in FAT-32, and today it is only really used for transportable media, like flash drives. NTFS is the file system developed specifically for the NT version of Windows. Starting with Windows XP it became the default file system installed by most computer manufacturers, greatly improving the security and functionality of Windows. NTFS uses 64-bit disk addresses and can (theoretically) support disk partitions up to 2^{64} bytes, although other considerations limit it to smaller sizes.

In this chapter we will examine the NTFS file system because it is a modern file system with many interesting features and design innovations. It is a large and complex file system and space limitations prevent us from covering all of its features, but the material presented below should give a reasonable impression of it.

11.8.1 Fundamental Concepts

Individual file names in NTFS are limited to 255 characters; full paths are limited to 32,767 characters. File names are in Unicode, allowing people in countries not using the Latin alphabet (e.g., Greece, Japan, India, Russia, and Israel) to write file names in their native language. For example, φιλε is a perfectly legal file name. NTFS fully supports case-sensitive names (so *foo* is different from *Foo* and *FOO*). The Win32 API does not fully support case-sensitivity for file names and not at all for directory names. The support for case-sensitivity exists when running the POSIX subsystem in order to maintain compatibility with UNIX. Win32 is not case-sensitive, but it is case-preserving, so file names can have different case letters in them. Though case-sensitivity is a feature that is very familiar to users of UNIX, it is largely inconvenient to ordinary users who do not make such distinctions normally. For example, the Internet is largely case-insensitive today.

An NTFS file is not just a linear sequence of bytes, as FAT-32 and UNIX files are. Instead, a file consists of multiple attributes, each of which is represented by a stream of bytes. Most files have a few short streams, such as the name of the file and its 64-bit object ID, plus one long (unnamed) stream with the data. However, a file can also have two or more (long) data streams as well. Each stream has a name consisting of the file name, a colon, and the stream name, as in *foo:stream1*. Each stream has its own size and is lockable independently of all the other streams. The idea of multiple streams in a file is not new in NTFS. The file system on the Apple Macintosh uses two streams per file, the data fork and the resource fork. The first use of multiple streams for NTFS was to allow an NT file server to serve Macintosh clients. Multiple data streams are also used to represent metadata about files, such as the thumbnail pictures of JPEG images that are available in the Windows GUI. But alas, the multiple data streams are fragile and frequently fall off of files when they are transported to other file systems, transported over the network, or even when backed up and later restored, because many utilities ignore them.

NTFS is a hierarchical file system, similar to the UNIX file system. The separator between component names is "\", however, instead of "/", a fossil inherited from the compatibility requirements with CP/M when MS-DOS was created. Unlike UNIX the concept of the current working directory, hard links to the current directory (.) and the parent directory (..) are implemented as conventions rather than as a fundamental part of the file system design. Hard links are supported, but only used for the POSIX subsystem, as is NTFS support for traversal checking on directories (the 'x' permission in UNIX).

Symbolic links in NTFS were not supported until Windows Vista. Creation of symbolic links is normally restricted to administrators to avoid security issues like spoofing, as UNIX experienced when symbolic links were first introduced in 4.2BSD. The implementation of symbolic links in Vista uses an NTFS feature called **reparse points** (discussed later in this section). In addition, compression, encryption, fault tolerance, journaling, and sparse files are also supported. These features and their implementations will be discussed shortly.

11.8.2 Implementation of the NT File System

NTFS is a highly complex and sophisticated file system that was developed specifically for NT as an alternative to the HPFS file system that had been developed for OS/2. While most of NT was designed on dry land, NTFS is unique among the components of the operating system in that much of its original design took place aboard a sailboat out on the Puget Sound (following a strict protocol of work in the morning, beer in the afternoon). Below we will examine a number of features of NTFS, starting with its structure, then moving on to file name lookup, file compression, journaling, and file encryption.

File System Structure

Each NTFS volume (e.g., disk partition) contains files, directories, bitmaps, and other data structures. Each volume is organized as a linear sequence of blocks (clusters in Microsoft's terminology), with the block size being fixed for each volume and ranging from 512 bytes to 64 KB, depending on the volume size. Most NTFS disks use 4-KB blocks as a compromise between large blocks (for efficient transfers) and small blocks (for low internal fragmentation). Blocks are referred to by their offset from the start of the volume using 64-bit numbers.

The main data structure in each volume is the **MFT (Master File Table)**, which is a linear sequence of fixed-size 1-KB records. Each MFT record describes one file or one directory. It contains the file's attributes, such as its name and timestamps, and the list of disk addresses where its blocks are located. If a file is extremely large, it is sometimes necessary to use two or more MFT records to contain the list of all the blocks, in which case the first MFT record, called the **base record**, points to the other MFT records. This overflow scheme dates back to CP/M, where each directory entry was called an extent. A bitmap keeps track of which MFT entries are free.

The MFT is itself a file and as such can be placed anywhere within the volume, thus eliminating the problem with defective sectors in the first track. Furthermore, the file can grow as needed, up to a maximum size of 2^{48} records.

The MFT is shown in Fig. 11-41. Each MFT record consists of a sequence of (attribute header, value) pairs. Each attribute begins with a header telling which attribute this is and how long the value is. Some attribute values are variable

length, such as the file name and the data. If the attribute value is short enough to fit in the MFT record, it is placed there. This is called an **immediate file** (Mullender and Tanenbaum, 1984). If it is too long, it is placed elsewhere on the disk and a pointer to it is placed in the MFT record. This makes NTFS very efficient for small fields, that is, those that can fit within the MFT record itself.

The first 16 MFT records are reserved for NTFS metadata files, as illustrated in Fig. 11-41. Each of the records describes a normal file that has attributes and data blocks, just like any other file. Each of these files has a name that begins with a dollar sign to indicate that it is a metadata file. The first record describes the MFT file itself. In particular, it tells where the blocks of the MFT file are located so that the system can find the MFT file. Clearly, Windows needs a way to find the first block of the MFT file in order to find the rest of the file system information. The way it finds the first block of the MFT file is to look in the boot block, where its address is installed when the volume is formatted with the file system.

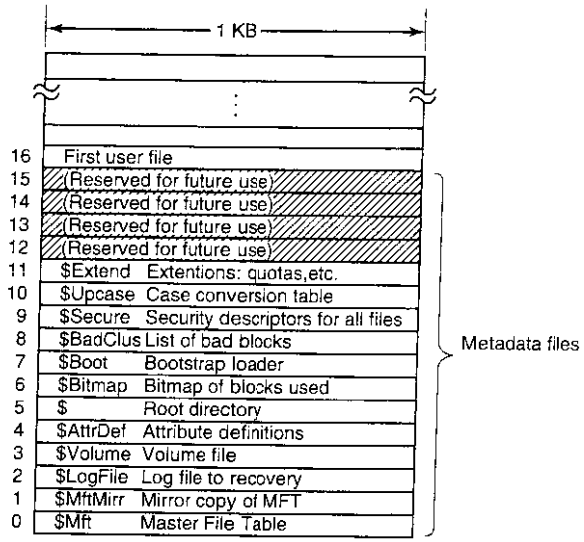

Figure 11-41. The NTFS master file table.

Record 1 is a duplicate of the early part of the MFT file. This information is so precious that having a second copy can be critical in the event one of the first blocks of the MFT ever goes bad. Record 2 is the log file. When structural changes are made to the file system, such as adding a new directory or removing an existing one, the action is logged here before it is performed, in order to increase the chance of correct recovery in the event of a failure during the operation, such

as a system crash. Changes to file attributes are also logged here. In fact, the only changes not logged here are changes to user data. Record 3 contains information about the volume, such as its size, label, and version.

As mentioned above, each MFT record contains a sequence of (attribute header, value) pairs. The *$AttrDef* file is where the attributes are defined. Information about this file is in MFT record 4. Next comes the root directory, which itself is a file and can grow to arbitrary length. It is described by MFT record 5.

Free space on the volume is kept track of with a bitmap. The bitmap is itself a file, and its attributes and disk addresses are given in MFT record 6. The next MFT record points to the bootstrap loader file. Record 8 is used to link all the bad blocks together to make sure they never occur in a file. Record 9 contains the security information. Record 10 is used for case mapping. For the Latin letters A-Z case mapping is obvious (at least for people who speak Latin). Case mapping for other languages, such as Greek, Armenian, or Georgian (the country, not the state), is less obvious to Latin speakers, so this file tells how to do it. Finally, record 11 is a directory containing miscellaneous files for things like disk quotas, object identifiers, reparse points, and so on. The last four MFT records are reserved for future use.

Each MFT record consists of a record header followed by the (attribute header, value) pairs. The record header contains a magic number used for validity checking, a sequence number updated each time the record is reused for a new file, a count of references to the file, the actual number of bytes in the record used, the identifier (index, sequence number) of the base record (used only for extension records), and some other miscellaneous fields.

NTFS defines 13 attributes that can appear in MFT records. These are listed in Fig. 11-42. Each attribute header identifies the attribute and gives the length and location of the value field along with a variety of flags and other information. Usually, attribute values follow their attribute headers directly, but if a value is too long to fit in the MFT record, it may be put in separate disk blocks. Such an attribute is said to be a **nonresident attribute**. The data attribute is an obvious candidate. Some attributes, such as the name, may be repeated, but all attributes must appear in a fixed order in the MFT record. The headers for resident attributes are 24 bytes long; those for nonresident attributes are longer because they contain information about where to find the attribute on disk.

The standard information field contains the file owner, security information, the timestamps needed by POSIX, the hard link count, the read-only and archive bits, and so on. It is a fixed-length field and is always present. The file name is a variable-length Unicode string. In order to make files with non–MS-DOS names accessible to old 16-bit programs, files can also have an 8 + 3 MS-DOS **short name**. If the actual file name conforms to the MS-DOS 8 + 3 naming rule, a secondary MS-DOS name is not needed.

In NT 4.0, security information was put in an attribute, but in Windows 2000 and later, security information all goes into a single file so that multiple files can

Attribute	Description
Standard information	Flag bits, timestamps, etc.
File name	File name in Unicode; may be repeated for MS-DOS name
Security descriptor	Obsolete. Security information is now in $Extend$Secure
Attribute list	Location of additional MFT records, if needed
Object ID	64-bit file identifier unique to this volume
Reparse point	Used for mounting and symbolic links
Volume name	Name of this volume (used only in $Volume)
Volume information	Volume version (used only in $Volume)
Index root	Used for directories
Index allocation	Used for very large directories
Bitmap	Used for very large directories
Logged utility stream	Controls logging to $LogFile
Data	Stream data; may be repeated

Figure 11-42. The attributes used in MFT records.

share the same security descriptions. This results in significant savings in space within most MFT records and in the file system overall because the security info for so many of the files owned by each user are identical.

The attribute list is needed in case the attributes do not fit in the MFT record. This attribute then tells where to find the extension records. Each entry in the list contains a 48-bit index into the MFT telling where the extension record is and a 16-bit sequence number to allow verification that the extension record and base records match up.

NTFS files have an ID associated with them that is like the i-node number in UNIX. Files can be opened by ID, but the ID's assigned by NTFS are not always useful when the ID must be persisted because it is based on the MFT record and can change if the record for the file moves (e.g., if the file is restored from backup). NTFS allows a separate object ID attribute which can be set on a file and never needs to change. It can be kept with the file if it is copied to a new volume, for example.

The reparse point tells the procedure parsing the file name to do something special. This mechanism is used for explicitly mounting file systems and for symbolic links. The two volume attributes are only used for volume identification. The next three attributes deal with how directories are implemented. Small ones are just lists of files but large ones are implemented using B+ trees. The logged utility stream attribute is used by the encrypting file system.

Finally, we come to the attribute that is the most important of all: the data stream (or in some cases, streams). An NTFS file has one or more data streams

associated with it. This is where the payload is. The **default data stream** is unnamed (i.e., *dirpath\filename::$DATA*), but the **alternate data streams** each have a name, for example, *dirpath\filename:streamname:$DATA*.

For each stream, the stream name, if present, goes in this attribute header. Following the header is either a list of disk addresses telling which blocks the stream contains, or for streams of only a few hundred bytes (and there are many of these), the stream itself. Putting the actual stream data in the MFT record is called an **immediate file** (Mullender and Tanenbaum, 1984).

Of course, most of the time the data does not fit in the MFT record, so this attribute is usually nonresident. Let us now take a look at how NTFS keeps track of the location of nonresident attributes, in particular data.

Storage Allocation

The model for keeping track of disk blocks is that they are assigned in runs of consecutive blocks, where possible, for efficiency reasons. For example, if the first logical block of a stream is placed in block 20 on the disk, then the system will try hard to place the second logical block in block 21, the third logical block in 22, and so on. One way to achieve these runs is to allocate disk storage several blocks at a time, when possible.

The blocks in a stream are described by a sequence of records, each one describing a sequence of logically contiguous blocks. For a stream with no holes in it, there will be only one such record. Streams that are written in order from beginning to end all belong in this category. For a stream with one hole in it (e.g., only blocks 0–49 and blocks 60–79 are defined), there will be two records. Such a stream could be produced by writing the first 50 blocks, then seeking forward to logical block 60 and writing another 20 blocks. When a hole is read back, all the missing bytes are zeros. Files with holes are called **sparse files**.

Each record begins with a header giving the offset of the first block within the stream. Next comes the offset of the first block not covered by the record. In the example above, the first record would have a header of (0, 50) and would provide the disk addresses for these 50 blocks. The second one would have a header of (60,80) and would provide the disk addresses for these 20 blocks.

Each record header is followed by one or more pairs, each giving a disk address and run length. The disk address is the offset of the disk block from the start of its partition; the run length is the number of blocks in the run. As many pairs as needed can be in the run record. Use of this scheme for a three-run, nine-block stream is illustrated in Fig. 11-43.

In this figure we have an MFT record for a short stream of nine blocks (header 0–8). It consists of the three runs of consecutive blocks on the disk. The first run is blocks 20–23, the second is blocks 64–65, and the third is blocks 80–82. Each of these runs is recorded in the MFT record as a (disk address, block count) pair. How many runs there are depends on how well the disk block allocator did

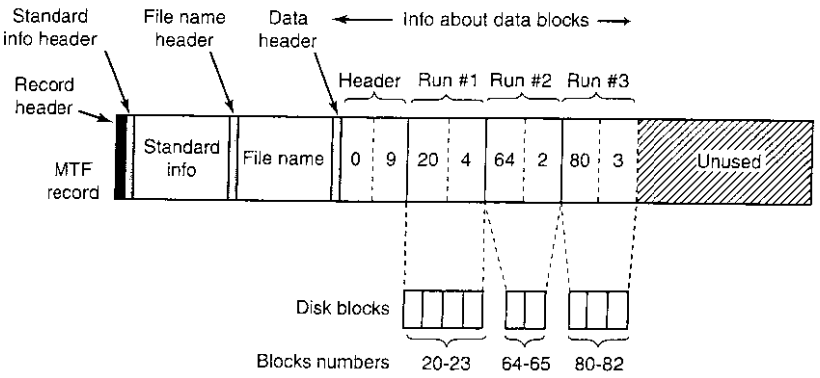

Figure 11-43. An MFT record for a three-run, nine-block stream.

in finding runs of consecutive blocks when the stream was created. For an *n*-block stream, the number of runs can be anything from *1* through *n*.

Several comments are worth making here. First, there is no upper limit to the size of streams that can be represented this way. In the absence of address compression, each pair requires two 64-bit numbers in the pair for a total of 16 bytes. However, a pair could represent 1 million or more consecutive disk blocks. In fact, a 20 MB stream consisting of 20 separate runs of 1 million 1-KB blocks each fits easily in one MFT record, whereas a 60-KB stream scattered into 60 isolated blocks does not.

Second, while the straightforward way of representing each pair takes 2×8 bytes, a compression method is available to reduce the size of the pairs below 16. Many disk addresses have multiple high-order zero-bytes. These can be omitted. The data header tells how many are omitted, that is, how many bytes are actually used per address. Other kinds of compression are also used. In practice, the pairs are often only 4 bytes.

Our first example was easy: all the file information fit in one MFT record. What happens if the file is so large or highly fragmented that the block information does not fit in one MFT record? The answer is simple: use two or more MFT records. In Fig. 11-44 we see a file whose base record is in MFT record 102. It has too many runs for one MFT record, so it computes how many extension records it needs, say, two, and puts their indices in the base record. The rest of the record is used for the first *k* data runs.

Note that Fig. 11-44 contains some redundancy. In theory, it should not be necessary to specify the end of a sequence of runs because this information can be calculated from the run pairs. The reason for "overspecifying" this information is to make seeking more efficient: to find the block at a given file offset, it is only necessary to examine the record headers, not the run pairs.

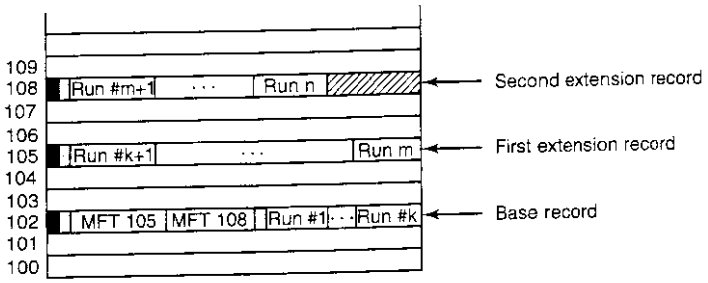

Figure 11-44. A file that requires three MFT records to store all its runs.

When all the space in record 102 has been used up, storage of the runs continues with MFT record 105. As many runs are packed in this record as fit. When this record is also full, the rest of the runs go in MFT record 108. In this way many MFT records can be used to handle large fragmented files.

A problem arises if so many MFT records are needed that there is no room in the base MFT to list all their indices. There is also a solution to this problem: the list of extension MFT records is made nonresident (i.e., stored in other disk blocks instead of in the base MFT record). Then it can grow as large as needed.

An MFT entry for a small directory is shown in Fig. 11-45. The record contains a number of directory entries, each of which describes one file or directory. Each entry has a fixed-length structure followed by a variable-length file name. The fixed part contains the index of the MFT entry for the file, the length of the file name, and a variety of other fields and flags. Looking for an entry in a directory consists of examining all the file names in turn.

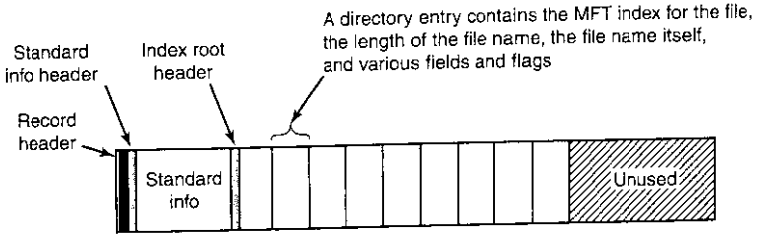

Figure 11-45. The MFT record for a small directory.

Large directories use a different format. Instead of listing the files linearly, a B+ tree is used to make alphabetical lookup possible and to make it easy to insert new names in the directory in the proper place.

We now have enough information to finish describing how file name lookup occurs for a file *\??\C:\foo\bar*. In Fig. 11-22 we saw how the Win32, the

native NT system calls, and the object and I/O managers cooperated to open a file by sending an I/O request to the NTFS device stack for the *C:* volume. The I/O request asks NTFS to fill in a file object for the remaining pathname, *\foo\bar*.

The NTFS parsing of the path *\foo\bar* begins at the root directory for *C:*, whose blocks can be found from entry 5 in the MFT (see Fig. 11-41). The string "foo" is looked up in the root directory, which returns the index into the MFT for the directory *foo*. This directory is then searched for the string "bar", which refers to the MFT record for this file. NTFS performs access checks by calling back into the security reference monitor, and if everything is cool it searches the MFT record for the attribute *::$DATA*, which is the default data stream.

Having found file *bar* NTFS will set pointers to its own metadata in the file object passed down from the I/O manager. The metadata includes a pointer to the MFT record, information about compression and range locks, various details about sharing, and so on. Most of this metadata is in data structures shared across all file objects referring to the file. A few fields are specific only to the current open, such as whether the file should be deleted when it is closed. Once the open has succeeded, NTFS calls IoCompleteRequest to pass the IRP back up the I/O stack to the I/O and object managers. Ultimately a handle for the file object is put in the handle table for the current process, and control is passed back to user mode. On subsequent ReadFile calls, an application can provide the handle, specifying that this file object for *C:\foo\bar* should be included in the read request that gets passed down the *C:* device stack to NTFS.

In addition to regular files and directories, NTFS supports hard links in the UNIX sense, and also symbolic links using a mechanism called **reparse points**. NTFS supports tagging a file or directory as a reparse point and associating a block of data with it. When the file or directory is encountered during a file name parse, the operation fails and the block of data is returned to the object manager. The object manager can interpret the data as representing an alternative pathname and then update the string to parse and retry the I/O operation. This mechanism is used to support both symbolic links and mounted file systems, redirecting the search to a different part of the directory hierarchy or even to a different partition.

Reparse points are also used to tag individual files for file system filter drivers. In Fig. 11-22 we showed how file system filters can be installed between the I/O manager and the file system. I/O requests are completed by calling IoCompleteRequest, which passes control to the completion routines each driver represented in the device stack inserted into the IRP as the request was being made. A driver that wants to tag a file associates a reparse tag and then watches for completion requests for file open operations that failed because they encountered a reparse point. From the block of data that is passed back with the IRP, the driver can tell if this is a block of data that the driver itself has associated with the file. If so the driver will stop processing the completion and continue processing the original I/O request. Generally, this will involve proceeding with the openrequest, but there is a flag that tells NTFS to ignore the reparse point and open the file.

File Compression

NTFS supports transparent file compression. A file can be created in compressed mode, which means that NTFS automatically tries to compress the blocks as they are written to disk and automatically uncompresses them when they are read back. Processes that read or write compressed files are completely unaware of the fact that compression and decompression are going on.

Compression works as follows. When NTFS writes a file marked for compression to disk, it examines the first 16 (logical) blocks in the file, irrespective of how many runs they occupy. It then runs a compression algorithm on them. If the resulting data can be stored in 15 or fewer blocks, the compressed data are written to the disk, preferably in one run, if possible. If the compressed data still take 16 blocks, the 16 blocks are written in uncompressed form. Then blocks 16–31 are examined to see if they can be compressed to 15 blocks or fewer, and so on.

Figure 11-46(a) shows a file in which the first 16 blocks have successfully compressed to eight blocks, the second 16 blocks failed to compress, and the third 16 blocks have also compressed by 50%. The three parts have been written as three runs and stored in the MFT record. The "missing" blocks are stored in the MFT entry with disk address 0 as shown in Fig. 11-46(b). Here the header (0, 48) is followed by five pairs, two for the first (compressed) run, one for the uncompressed run, and two for the final (compressed) run.

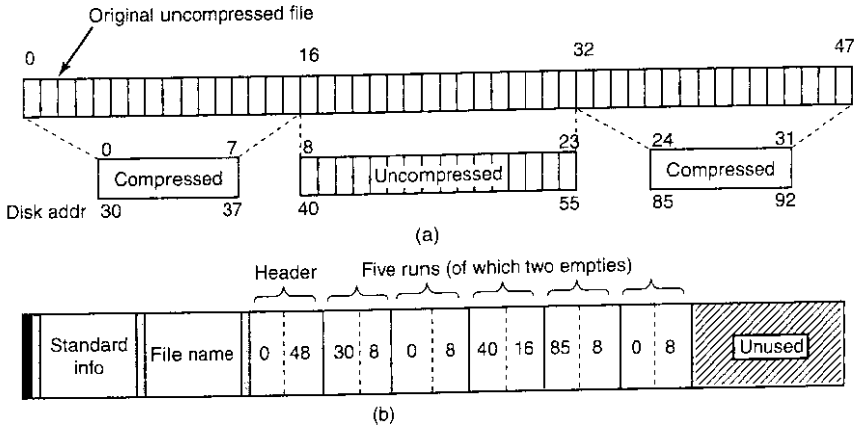

Figure 11-46. (a) An example of a 48-block file being compressed to 32 blocks. (b) The MFT record for the file after compression.

When the file is read back, NTFS has to know which runs are compressed and which ones are not. It can tell based on the disk addresses. A disk address of 0 indicates that it is the final part of 16 compressed blocks. Disk block 0 may not be used for storing data, to avoid ambiguity. Since block 0 on the volume contains the boot sector, using it for data is impossible anyway.

Random access to compressed files is possible, but tricky. Suppose that a process does a seek to block 35 in Fig. 11-46. How does NTFS locate block 35 in a compressed file? The answer is that it has to read and decompress the entire run first. Then it knows where block 35 is and can pass it to any process that reads it. The choice of 16 blocks for the compression unit was a compromise. Making it shorter would have made the compression less effective. Making it longer would have made random access more expensive.

Journaling

NTFS supports two mechanisms for programs to detect changes to files and directories on a volume. The first mechanism is an I/O operation that is called NtNotifyChangeDirectoryFile that passes a buffer to the system, which returns when a change is detected to a directory or directory sub-tree. The result of the I/O is that the buffer has been filled with a list of *change records*. With luck, the buffer is large enough. Otherwise the records which did not fit are lost.

The second mechanism is the NTFS change journal. NTFS keeps a list of all the change records for directories and files on the volume in a special file, which programs can read using special file system control operations, that is, the *FSCTL_QUERY_USN_JOURNAL* option to the NtFsControlFile API. The journal file is normally very large, and there is little likelihood that entries will be reused before they can be examined.

File Encryption

Computers are used nowadays to store all kinds of sensitive data, including plans for corporate takeovers, tax information, and love letters, which the owners do not especially want revealed to anyone. Information loss can happen when a notebook computer is lost or stolen, a desktop system is rebooted using an MS-DOS floppy disk to bypass Windows security, or a hard disk is physically removed from one computer and installed on another one with an insecure operating system.

Windows addresses these problems by providing an option to encrypt files, so that even in the event the computer is stolen or rebooted using MS-DOS, the files will be unreadable. The normal way to use Windows encryption is to mark certain directories as encrypted, which causes all the files in them to be encrypted, and new files moved to them or created in them to be encrypted as well. The actual encryption and decryption are not managed by NTFS itself, but by a driver called **EFS (Encryption File System)**, which registers callbacks with NTFS.

EFS provides encryption for specific files and directories. There is also another encryption facility in Windows Vista called **BitLocker** which encrypts almost all the data on a volume, which can help protect data no matter what—as long as the user takes advantage of the mechanisms available for strong keys. Given the number of systems that are lost or stolen all the time, and the great sensitivity to the issue of identity theft, making sure secrets are protected is very important. An amazing number of notebooks go missing every day. Major Wall Street companies supposedly average losing one notebook per week in taxicabs in New York City alone.

11.9 SECURITY IN WINDOWS VISTA

Having just looked at encryption, this is a good time to examine security in general. NT was originally designed to meet the U.S. Department of Defense's C2 security requirements (DoD 5200.28-STD), the Orange Book, which secure DoD systems must meet. This standard requires operating systems to have certain properties in order to be classified as secure enough for certain kinds of military work. Although Windows Vista was not specifically designed for C2 compliance, it inherits many security properties from the original security design of NT, including the following:

1. Secure login with anti-spoofing measures.
2. Discretionary access controls.
3. Privileged access controls.
4. Address space protection per process.
5. New pages must be zeroed before being mapped in.
6. Security auditing.

Let us review these items briefly

Secure login means that the system administrator can require all users to have a password in order to log in. Spoofing is when a malicious user writes a program that displays the login prompt or screen and then walks away from the computer in the hope that an innocent user will sit down and enter a name and password. The name and password are then written to disk and the user is told that login has failed. Windows Vista prevents this attack by instructing users to hit CTRL-ALT-DEL to log in. This key sequence is always captured by the keyboard driver, which then invokes a system program that puts up the genuine login screen. This procedure works because there is no way for user processes to disable CTRL-ALT-DEL processing in the keyboard driver. But NT can and does disable use of the CTRL-ALT-DEL secure attention sequence in some cases. This idea

came from Windows XP and Windows 2000, which used in order to have more compatibility for users switching from Windows 98.

Discretionary access controls allow the owner of a file or other object to say who can use it and in what way. Privileged access controls allow the system administrator (superuser) to override them when needed. Address space protection simply means that each process has its own protected virtual address space not accessible by any unauthorized process. The next item means that when the process heap grows, the pages mapped in are initialized to zero so that processes cannot find any old information put there by the previous owner (hence the zeroed page list in Fig. 11-36, which provides a supply of zeroed pages for this purpose). Finally, security auditing allows the administrator to produce a log of certain security-related events.

While the Orange Book does not specify what is to happen when someone steals your notebook computer, in large organizations one theft a week is not unusual. Consequently, Windows Vista provides tools that a conscientious user can use to minimize the damage when a notebook is stolen or lost (e.g., secure login, encrypted files, etc.). Of course, conscientious users are precisely the ones who do not lose their notebooks—it is the others who cause the trouble.

In the next section we will describe the basic concepts behind Windows Vista security. After that we will look at the security system calls. Finally, we will conclude by seeing how security is implemented.

11.9.1 Fundamental Concepts

Every Windows Vista user (and group) is identified by an **SID (Security ID)**. SIDs are binary numbers with a short header followed by a long random component. Each SID is intended to be unique worldwide. When a user starts up a process, the process and its threads run under the user's SID. Most of the security system is designed to make sure that each object can be accessed only by threads with authorized SIDs.

Each process has an **access token** that specifies an SID and other properties. The token is normally created by *winlogon*, as described below. The format of the token is shown in Fig. 11-47. Processes can call GetTokenInformation to acquire this information. The header contains some administrative information. The expiration time field could tell when the token ceases to be valid, but it is currently not used. The *Groups* field specifies the groups to which the process belongs, which is needed for the POSIX subsystem. The default **DACL (Discretionary ACL)** is the access control list assigned to objects created by the process if no other ACL is specified. The user SID tells who owns the process. The restricted SIDs are to allow untrustworthy processes to take part in jobs with trustworthy processes but with less power to do damage.

Finally, the privileges listed, if any, give the process special powers denied ordinary users, such as the right to shut the machine down or access files to which

access would otherwise be denied. In effect, the privileges split up the power of the superuser into several rights that can be assigned to processes individually. In this way, a user can be given some superuser power, but not all of it. In summary, the access token tells who owns the process and which defaults and powers are associated with it.

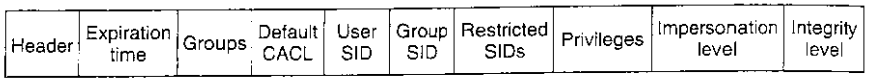

Figure 11-47. Structure of an access token.

When a user logs in, *winlogon* gives the initial process an access token. Subsequent processes normally inherit this token on down the line. A process' access token initially applies to all the threads in the process. However, a thread can acquire a different access token during execution, in which case the thread's access token overrides the process' access token. In particular, a client thread can pass its access rights to a server thread to allow the server to access the client's protected files and other objects. This mechanism is called **impersonation**. It is implemented by the transport layers (i.e., ALPC, named pipes, and TCP/IP), used by RPC to communicate from clients to servers. The transports use internal interfaces in the kernel's security reference monitor component to extract the security context for the current thread's access token and ship it to the server side, where it is used to construct a token which can be used by the server to impersonate the client.

Another basic concept is the **security descriptor**. Every object has a security descriptor associated with it that tells who can perform which operations on it. The security descriptors are specified when the objects are created. The NTFS file system and the registry maintain a persistent form of security descriptor, which is used to create the security descriptor for File and Key objects (the object manager objects representing open instances of files and keys).

A security descriptor consists of a header followed by a DACL with one or more **ACEs** (**Access Control Entries**). The two main kinds of elements are Allow and Deny. An allow element specifies an SID and a bitmap that specifies which operations processes that SID may perform on the object. A deny element works the same way, except a match means the caller may not perform the operation. For example, Ida has a file whose security descriptor specifies that everyone has read access, Elvis has no access. Cathy has read/write access, and Ida herself has full access. This simple example is illustrated in Fig. 11-48. The SID Everyone refers to the set of all users, but it is overridden by any explicit ACEs that follow.

In addition to the DACL, a security descriptor also has a **SACL** (**System Access Control list**), which is like a DACL except that it specifies not who may use the object, but which operations on the object are recorded in the system-wide

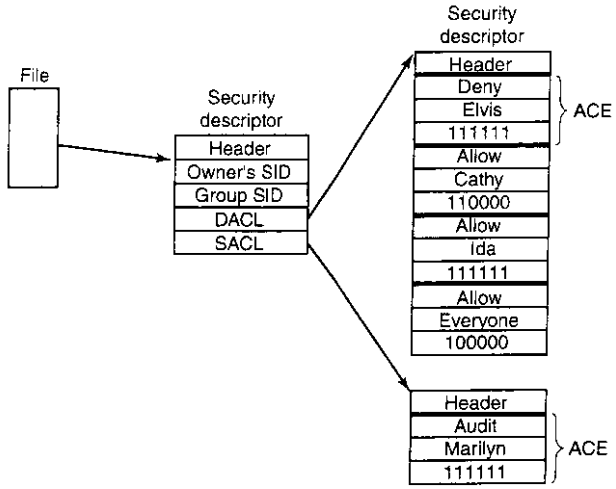

Figure 11-48. An example security descriptor for a file.

security event log. In Fig. 11-48, every operation that Marilyn performs on the file will be logged. The SACL also contains the **integrity level**, which we will describe shortly.

11.9.2 Security API Calls

Most of the Windows Vista access control mechanism is based on security descriptors. The usual pattern is that when a process creates an object, it provides a security descriptor as one of the parameters to the CreateProcess, CreateFile, or other object creation call. This security descriptor then becomes the security descriptor attached to the object, as we saw in Fig. 11-48. If no security descriptor is provided in the object creation call, the default security in the caller's access token (see Fig. 11-47) is used instead.

Many of the Win32 API security calls relate to the management of security descriptors, so we will focus on those here. The most important calls are listed in Fig. 11-49. To create a security descriptor, storage for it is first allocated and then initialized using InitializeSecurityDescriptor. This call fills in the header. If the owner SID is not known, it can be looked up by name using LookupAccountSid. It can then be inserted into the security descriptor. The same holds for the group SID, if any. Normally, these will be the caller's own SID and one of the caller's groups, but the system administrator can fill in any SIDs.

At this point the security descriptor's DACL (or SACL) can be initialized with InitializeAcl. ACL entries can be added using AddAccessAllowedAce, and

Win32 API function	Description
InitializeSecurityDescriptor	Prepare a new security descriptor for use
LookupAccountSid	Look up the SID for a given user name
SetSecurityDescriptorOwner	Enter the owner SID in the security descriptor
SetSecurityDescriptorGroup	Enter a group SID in the security descriptor
InitializeAcl	Initialize a DACL or SACL
AddAccessAllowedAce	Add a new ACE to a DACL or SACL allowing access
AddAccessDeniedAce	Add a new ACE to a DACL or SACL denying access
DeleteAce	Remove an ACE from a DACL or SACL
SetSecurityDescriptorDacl	Attach a DACL to a security descriptor

Figure 11-49. The principal Win32 API functions for security.

AddAccessDeniedAce. These calls can be repeated multiple times to add as many ACE entries as are needed. DeleteAce can be used to remove an entry, that is, when modifying an existing ACL rather than when constructing a new ACL. When the ACL is ready, SetSecurityDescriptorDacl can be used to attach it to the security descriptor. Finally, when the object is created, the newly minted security descriptor can be passed as a parameter to have it attached to the object.

11.9.3 Implementation of Security

Security in a standalone Windows Vista system is implemented by a number of components, most of which we have already seen (networking is a whole other story and beyond the scope of this book). Logging in is handled by *winlogon* and authentication is handled by *lsass*. The result of a successful login is a new GUI shell *explorer.exe*) with its associated access token. This process uses the SECURITY and SAM hives in the registry. The former sets the general security policy and the latter contains the security information for the individual users, as discussed in Sec. 11.2.3.

Once a user is logged in, security operations happen when an object is opened for access. Every OpenXXX call requires the name of the object being opened and the set of rights needed. During processing of the open, the security reference monitor (see Fig. 11-13) checks to see if the caller has all the rights required. It performs this check by looking at the caller's access token and the DACL associated with the object. It goes down the list of ACEs in the ACL in order. As soon as it finds an entry that matches the caller's SID or one of the caller's groups, the access found there is taken as definitive. If all the rights the caller needs are available, the open succeeds; otherwise it fails.

DACLs can have Deny entries as well as Allow entries, as we have seen. For this reason, it is usual to put entries denying access in front of entries granting access in the ACL, so that a user who is specifically denied access cannot get in via a back door by being a member of a group that has legitimate access.

After an object has been opened, a handle to it is returned to the caller. On subsequent calls, the only check that is made is whether the operation now being tried was in the set of operations requested at open time, to prevent a caller from opening a file for reading and then trying to write on it. Additionally, calls on handles may result in entries in the audit logs, as required by the SACL.

Windows Vista added another security facility to deal with common problems securing the system by ACLs. There are new mandatory **Integrity-level SIDs** in the process token, and objects specify an integrity-level ACE in the SACL. The integrity level prevents write-access to objects no matter what ACEs are in the DACL. In particular the integrity-level scheme is used to protect against an Internet Explorer process that has been compromised by an attacker (perhaps the user ill-advisedly downloading code from an unknown Website). **Low-rights IE**, as it is called, runs with an integrity level set to *low*. By default all files and registry keys in the system have an integrity level of *medium*, so IE running with low-integrity level cannot modify them.

A number of other security features have been added to Windows in recent years. For service pack 2 of Windows XP, much of the system was compiled with a flag (/GS) that did validation against many kinds of stack buffer overflows. Additionally a facility in the AMD64 architecture, called NX, was used to limit execution of code on stacks. The NX bit in the processor is available even when running in x86 mode. NX stands for *no execute* and allows pages to be marked so that code cannot be executed from them. Thus if an attacker uses a buffer overflow vulnerability to insert code into a process, it is not so easy to jump to the code and start executing it.

Windows Vista introduced even more security features to foil attackers. Code loaded into kernel mode is checked (by default on x64 systems) and only loaded if it is properly signed.. The addresses that DLLs and EXEs are loaded at, as well as stack allocations, are shuffled quite a bit on each system to make it less likely that an attacker can successfully use buffer overflows to branch into a well-known address and begin executing sequences of code that can be weaved into an elevation of privilege. A much smaller fraction of systems will be able to be attacked by relying on binaries being at standard addresses. Systems are far more likely to just crash, converting a potential elevation attack into a less dangerous denial-of-service attack.

Yet another change was the introduction of what Microsoft calls **UAC (User Account Control)**. This is to address the chronic problem in Windows where most users run as administrators. The design of Windows does not require users to run as administrators, but neglect over many releases had made it just about impossible to use Windows successfully if you were not an administrator. Being an administrator all the time is dangerous. Not only can user errors easily damage the system, but if the user is somehow fooled or attacked and runs code that is

trying to compromise the system, the code will have administrative access, and can bury itself deep in the system.

With UAC, if an attempt is made to perform an operation requiring administrator access, the system overlays a special desktop and takes control so that only input from the user can authorize the access (similar to how CTRL-ALT-DEL works for C2 security). Of course, without becoming administrator it is possible for an attacker to destroy what the user really cares about, namely his personal files. But UAC does help foil existing types of attacks, and it is always easier to recover a compromised system if the attacker was unable to modify any of the system data or files.

The final security feature in Windows Vista is one we have already mentioned. There is support to create *protected processes* which provide a security boundary. Normally, the user (as represented by a token object) defines the privilege boundary in the system. When a process is created, the user has access to process through any number of kernel facilities for process creation, debugging, pathnames, thread injection, and so on. Protected processes are shut off from user access. The only use of this facility in Vista is to allow Digital Rights Management software to better protect content. Perhaps use of protected processes will be expanded in future releases to more user-friendly purposes, like securing the system against attackers rather than securing content against attacks by the system owner.

Microsoft's efforts to improve the security of Windows have accelerated in recent years as more and more attacks have been launched against systems around the world. Some of these attacks have been very successful, taking entire countries and major corporations offline, and incurring costs of billions of dollars. Most of the attacks exploit small coding errors that lead to buffer overruns, allowing the attacker to insert code by overwriting return addresses, exception pointers, and other data that control the execution of programs. Many of these problems could be avoided if type-safe languages were used instead of C and C++. And even with these unsafe languages many vulnerabilities could be avoided if students were better trained to understand the pitfalls of parameter and data validation. After all, many of the software engineers who write code at Microsoft were students a few years earlier, just like many of you reading this case study are now. There are many books available on the kinds of small coding errors that are exploitable in pointer-based languages and how to avoid them (e.g., Howard and LeBlank, 2007).

11.10 SUMMARY

Kernel mode in Windows Vista is structured in the HAL, the kernel and executive layers of NTOS, and a large number of device drivers implementing everything from device services to file systems and networking to graphics. The HAL

hides certain differences in hardware from the other components. The kernel layer manages the CPUs to support multithreading and synchronization, and the executive implements most kernel-mode services.

The executive is based on kernel-mode objects that represent the key executive data structures, including processes, threads, memory sections, drivers, devices, and synchronization objects—to mention a few. User processes create objects by calling system services and get back handle references which can be used in subsequent system calls to the executive components. The operating system also creates objects internally. The object manager maintains a name space into which objects can be inserted for subsequent lookup.

The most important objects in Windows are processes, threads, and sections. Processes have virtual address spaces and are containers for resources. Threads are the unit of execution and are scheduled by the kernel layer using a priority algorithm in which the highest-priority ready thread always runs, preempting lower-priority threads as necessary. Sections represent memory objects, like files, that can be mapped into the address spaces of processes. EXE and DLL program images are represented as sections, as is shared memory.

Windows supports demand-paged virtual memory. The paging algorithm is based on the working-set concept. The system maintains several types of page lists, to optimize the use of memory. The various page lists are fed by trimming the working sets using complex formulas that try to reuse physical pages that have not been referenced in a long time. The cache manager manages virtual addresses in the kernel that can be used to map files into memory, dramatically improving I/O performance for many applications because read operations can be satisfied without accessing disk.

I/O is performed by device drivers, which follow the Windows Driver Model. Each driver starts out by initializing a driver object that contains the addresses of the procedures that the system can call to manipulate devices. The actual devices are represented by device objects, which are created from the configuration description of the system or by the plug-and-play manager as it discovers devices when enumerating the system buses. Devices are stacked and I/O request packets are passed down the stack and serviced by the drivers for each device in the device stack. I/O is inherently asynchronous, and drivers commonly queue requests for further work and return back to their caller. File system volumes are implemented as devices in the I/O system.

The NTFS file system is based on a master file table, which has one record per file or directory. All the metadata in an NTFS file system is itself part of an NTFS file. Each file has multiple attributes, which can either be in the MFT record or nonresident (stored in blocks outside the MFT). NTFS supports Unicode, compression, journaling, and encryption among many other features.

Finally, Windows Vista has a sophisticated security system based on access control lists and integrity levels. Each process has an authentication token that tells the identity of the user and what special privileges the process has, if any.

Each object has a security descriptor associated with it. The security descriptor points to a discretionary access control list that contains access control entries that can allow or deny access to individuals or groups. Windows has added numerous security features in recent releases, including BitLocker for encrypting entire volumes, and address space randomization, nonexecutable stacks, and other measures to make buffer overflow attacks more difficult.

PROBLEMS

1. The HAL keeps track of time starting in the year 1601. Give an example of an application where this feature is useful.
2. In Sec. 11.3.2 we described the problems caused by multithreaded applications closing handles in one thread while still using them in another. One possibility for fixing this would be to insert a sequence field. How could this help? What changes to the system would be required?
3. Win32 does not have signals. If they were to be introduced, they could be per process, per thread, both, or neither. Make a proposal and explain why it is a good idea.
4. An alternative to using DLLs is to statically link each program with precisely those library procedures it actually calls, no more and no less. If this scheme were to be introduced, would it make more sense on client machines or on server machines?
5. What are some reasons why a thread has separate user-mode and kernel-mode stacks in Windows?
6. Windows uses 4-MB pages because it improves the effectiveness of the TLB, which can have a profound impact on performance. Why is this?
7. Is there any limit on the number of different operations that can be defined on an executive object? If so, where does this limit come from? If not, why not?
8. The Win32 API call WaitForMultipleObjects allows a thread to block on a set of synchronization objects whose handles are passed as parameters. As soon as any one of them is signaled, the calling thread is released. Is it possible to have the set of synchronization objects include two semaphores, one mutex, and one critical section? Why or why not? *Hint*: This is not a trick question but it does require some careful thought.
9. Name three reasons why a process might be terminated.
10. As described in Sec. 11.4, there is a special handle table used to allocate IDs for processes and threads. The algorithms for handle tables normally allocate the first available handle (maintaining the free list in LIFO order). In recent releases of Windows this was changed so that the ID table always keeps the free list in FIFO order. What is the problem that the LIFO ordering potentially causes for allocating process IDs, and why does not .UX have this problem?

11. Suppose that the quantum is set to 20 msec and the current thread, at priority 24, has just started a quantum. Suddenly an I/O operation completes and a priority 28 thread is made ready. About how long does it have to wait to get to run on the CPU?
12. In Windows Vista, the current priority is always greater than or equal to the base priority. Are there any circumstances in which it would make sense to have the current priority be lower than the base priority? If so, give an example. If not, why not?
13. In Windows it was easy to implement a facility where threads running in the kernel can temporarily attach to the address space of a different process. Why is this so much harder to implement in user mode? Why might it be interesting to do so?
14. Even when there is plenty of free memory available, and the memory manager does not need to trim working sets, the paging system can still frequently be writing to disk. Why?
15. Why does the self-map used to access the physical pages of the page directory and page tables for a process always occupy the same 4 MB of kernel virtual addresses (on the x86)?
16. If a region of virtual address space is reserved but not committed, do you think a VAD is created for it? Defend your answer.
17. Which of the transitions shown in Fig. 11-36 are policy decisions, as opposed to required moves forced by system events (e.g., a process exiting and freeing its pages)?
18. Suppose that a page is shared and in two working sets at once. If it is evicted from one of the working sets, where does it go in Fig. 11-36? What happens when it is evicted from the second working set?
19. When a process unmaps a clean stack page, it makes the transition (5) in Fig. 11-36. Where does a dirty stack page go when unmapped? Why is there no transition to the modified list when a dirty stack page is unmapped?
20. Suppose that a dispatcher object representing some type of exclusive lock (like a mutex) is marked to use a notification event instead of a synchronization event to announce that the lock has been released. Why would this be bad? How much would the answer depend on lock hold times, the length of quantum, and whether the system was a multiprocessor?
21. A file has the following mapping. Give the MFT run entries.

Offset	0	1	2	3	4	5	6	7	8	9	10
Disk address	50	51	52	22	24	25	26	53	54	-	60

22. Consider the MFT record of Fig. 11-43. Suppose that the file grew and a 10th block was assigned to the end of the file. The number of this block is 66. What would the MFT record look like now?
23. In Fig. 11-46(b), the first two runs are each of length 8 blocks. Is it just an accident that they are equal, or does this have to do with the way compression works? Explain your answer.
24. Suppose that you wanted to build Windows Vista Lite. Which of the fields of Fig. 11-47 could be removed without weakening the security of the system?

25. An extension model used by many programs (Web browsers, Office, COM servers) involves *hosting* DLLs to hook and extend their underlying functionality. Is this a reasonable model for an RPC-based service to use as long as it is careful to impersonate clients before loading the DLL? Why not?

26. When running on a NUMA machine, whenever the Windows memory manager needs to allocate a physical page to handle a page fault it attempts to use a page from the NUMA node for the current thread's ideal processor. Why? What if the thread is currently running on a different processor?

27. Give a couple of examples where an application might be able to recover easily from a backup based on a volume shadow copy rather the state of the disk after a system crash.

28. In Sec. 11.9, providing new memory to the process heap was mentioned as one of the scenarios that require a supply of zeroed pages in order to satisfy security requirements. Give one or more other examples of virtual memory operations that require zeroed pages.

29. The *regedit* command can be used to export part or all of the registry to a text file under all current versions of Windows. Save the registry several times during a work session and see what changes. If you have access to a Windows computer on which you can install software or hardware, find out what changes when a program or device is added or removed.

30. Write a UNIX program that simulates writing an NTFS file with multiple streams. It should accept a list of one or more files as arguments and write an output file that contains one stream with the attributes of all arguments and additional streams with the contents of each of the arguments. Now write a second program for reporting on the attributes and streams and extracting all the components.

12

CASE STUDY 3: SYMBIAN OS

In the previous two chapters, we have examined two operating systems popular on desktops and notebooks: Linux and Windows Vista. However, more than 90% of the CPUs in the world are not in desktops and notebooks. They are in embedded systems like cell phones, PDAs, digital cameras, camcorders, game machines, iPods, MP3 players, CD players, DVD recorders, wireless routers, TV sets, GPS receivers, laser printers, cars, and many more consumer products. Most of these use modern 32-bit and 64-bit chips, and nearly all of them run a full-blown operating system. But few people are even aware of the existence of these operating systems. In this chapter we will take a close look at one operating system popular in the embedded-systems world: Symbian OS.

Symbian OS is an operating system that runs on mobile "smartphone" platforms from several different manufacturers. Smartphones are so named because they run fully featured operating systems and utilize the features of desktop computers. Symbian OS is designed so that it can be the basis of a wide variety of smartphones from several different manufacturers. It was carefully designed specifically to run on smartphone platforms: general-purpose computers with limited CPU, memory and storage capacity, focused on communication.

Our discussion of Symbian OS will start with its history. We will then provide an overview of the system to give an idea of how it is designed and what uses the designers intended for it. Next we will examine the various aspects of Symbian OS design as we have for Linux and for Windows: we will look at processes, memory management, I/O, the file system, and security. We will conclude with a look at how Symbian OS addresses communication in smartphones.

12.1 THE HISTORY OF SYMBIAN OS

UNIX has a long history, almost ancient in terms of computers. Windows has a moderately long history. Symbian OS, on the other hand, has a fairly short history. It has roots in systems that were developed in the 1990s and its debut was in 2001. This should not be surprising, since the smartphone platform upon which Symbian OS runs has evolved only recently as well.

Symbian OS has its roots in handheld devices and has seen rapid development through several versions.

12.1.1 Symbian OS Roots: Psion and EPOC

The heritage of Symbian OS begins with some of the first handheld devices. Handheld devices evolved in the late 1980s as a means to capture the usefulness of a desktop device in a small, mobile package. The first attempts at a handheld computer did not meet with much excitement; the Apple Newton was a well-designed device that was popular with only a few users. Despite this slow start, the handheld computers developed by the mid-1990s were better tailored to the user and the way that people used mobile devices. Handheld computers were originally designed as PDAs—personal digital assistants that were essentially electronic planners—but evolved to embrace many types of functionality. As they developed, they began to function like desktop computers and they started to have the same needs as well. They needed to multitask; they added storage capabilities in multiple forms; they needed to be flexible in areas of input and output.

Handheld devices also grew to embrace communication. As these personal devices grew, personal communication was also developing. Mobile phones saw a dramatic increase in use in the late 1990s. Thus, it was natural to merge handheld devices with mobile phones to form smartphones. The operating systems that ran handheld devices had to develop as this merger took place.

In the 1990s, Psion Computers manufactured devices that were PDAs. In 1991, Psion produced the Series 3: a small computer with a half-VGA, monochrome screen that could fit into a pocket. The Series 3 was followed by the Series 3c in 1996, with additional infrared capability, and the Series 3mx in 1998, with a faster processor and more memory. Each of these devices was a great success, primarily because of good power management and interoperability with other computers, including PCs and other handheld devices. Programming was based in the language C, had an object-oriented design, and employed **application engines**, a signature part of Symbian OS development. This engine approach was a powerful feature. It borrowed from microkernel design to focus functionality in engines—which functioned like servers—that managed functions in response to requests from applications. This approach made it possible to standardize an API and to use object abstraction to remove the application programmer from worrying about tedious details like data formats.

In 1996, Psion started to design a new 32-bit operating system that supported pointing devices on a touch screen, used multimedia, and was more communication rich. The new system was also more object-oriented, and was to be portable to different architectures and device designs. The result of Psion's effort was the introduction of the system as EPOC Release 1. EPOC was programmed in C++ and was designed to be object-oriented from the ground up. It again used the engine approach and expanded this design idea into a series of servers that coordinated access to system services and peripheral devices. EPOC expanded the communication possibilities, opened up the operating system to multimedia, introduced new platforms for interface items like touch screens, and generalized the hardware interface.

EPOC was further developed into two more releases: EPOC Release 3 (ER3) and EPOC Release 5 (ER5). These ran on new platforms like the Psion Series 5 and Series 7 computers.

Psion also looked to emphasize the ways that its operating system could be adapted to other hardware platforms. Around the year 2000, the most opportunities for new handheld development were in the mobile phone business, where manufacturers were already searching for a new, advanced operating system for its next generation of devices. To take advantage of these opportunities, Psion and the leaders in the mobile phone industry, including Nokia, Ericsson, Motorola, and Matsushita (Panasonic), formed a joint venture, called Symbian, which was to take ownership of and further develop the EPOC operating system core. This new core design was now called Symbian OS.

12.1.2 Symbian OS Version 6

Since EPOC's last version was ER5, Symbian OS debuted at version 6 in 2001. It took advantage of the flexible properties of EPOC and was targeted at several different generalized platforms. It was designed to be flexible enough to meet the requirements for developing a variety of advanced mobile devices and phones, while allowing manufacturers the opportunity to differentiate their products.

It was also decided that Symbian OS would actively adopt current, state-of-the-art key technologies as they became available. This decision reinforced the design choices of object orientation and a client-server architecture as these were becoming widespread in the desktop and Internet worlds.

Symbian OS version 6 was called "open" by its designers. This was different than the "open source" properties often attributed to UNIX and Linux. By "open," Symbian OS designers meant that the structure of the operating system was published and available to all. In addition, all system interfaces were published to foster third-party software design.

12.1.3 Symbian OS Version 7

Symbian OS version 6 looked very much like its EPOC and version 6 predecessors in design and function. The design focus had been to embrace mobile telephony. However, as more and more manufacturers designed mobile phones, it became obvious that even the flexibility of EPOC, a handheld operating system, would not be able to address the plethora of new phones that needed to use Symbian OS.

Symbian OS version 7 kept the desktop functionality of EPOC, but most system internals were rewritten to embrace many kinds of smartphone functionality. The operating system kernel and operating system services were separated from the user interface. The same operating system could now be run on many different smartphone platforms, each of which used a different user interface system. Symbian OS could now be extended to address new and unpredicted messaging formats, for example, or could be used on different smartphones that used different phone technologies. Symbian OS version 7 was released in 2003.

12.1.4 Symbian OS Today

Symbian OS version 7 was a very important release because it built abstraction and flexibility into the operating system. However, this abstraction came at a price. The performance of the operating system soon became an issue that needed to be addressed.

A project was undertaken to completely rewrite the operating system again, this time focusing on performance. The new operating system design was to retain the flexibility of Symbian OS version 7 while enhancing performance and making the system more secure. Symbian OS version 8, released in 2004, enhanced the performance of Symbian OS, particularly for its real-time functions. Symbian OS version 9, released in 2005, added concepts of capability-based security and gatekeeping installation. Symbian OS version 9 also added the flexibility for hardware that Symbian OS version 7 added for software. A new binary model was developed that allowed hardware developers to use Symbian OS without redesigning the hardware to fit a specific architectural model.

12.2 AN OVERVIEW OF SYMBIAN OS

As the previous section demonstrates, Symbian OS has evolved from a handheld operating system to an operating system that specifically targets real-time performance on a smartphone platform. This section will provide a general introduction to the concepts embodied in the design of Symbian OS. These concepts directly correspond to how the operating system is used.

Symbian OS is unique among operating systems in that it was designed with smartphones as the target platform. It is not a generic operating system shoehorned (with great difficulty) into a smartphone, nor is it an adaptation of a larger operating system for a smaller platform. It does, however, have many of the features of other, larger operating systems, from multitasking to memory management to security issues.

The predecessors to Symbian OS have given it their best features. Symbian OS is object-oriented, inherited from EPOC. It uses a microkernel design, which minimizes kernel overhead and pushes nonessential functionality to user-level processes, as introduced in version 6. It uses a client/server architecture, which mimics the engine model built into EPOC. It supports many desktop features, including multitasking and multithreading, and an extensible storage system. It also inherited a multimedia and communication emphasis from EPOC and the move to Symbian OS.

12.2.1 Object Orientation

Object orientation is a term that implies abstraction. An object oriented design is a design that creates an abstract entity called an **object** of the data and functionality of a system component. An object provides specified data and functionality but hides the details of implementation. A properly implemented object can be removed and replaced by a different object as long as the way that other pieces of the system use that object, that is, as long as its interface remains the same.

When applied to operating system design, object orientation means that all use of system calls and kernel-side features is through interfaces with no access to actual data or reliance on any type of implementation. An object-oriented kernel provides kernel services through objects. Using kernel-side objects usually means that an application obtains a **handle**, that is, a reference, to an object, then accesses that object's interface through this handle.

Symbian OS is object-oriented by design. Implementations of system facilities are hidden; usage of system data is done through defined interfaces on system objects. Where an operating system like Linux might create a file descriptor and use that descriptor as a parameter in an open call, Symbian OS would create a file object and call the open method connected to the object. For example, in Linux , it is widely known that file descriptors are integers that index a table in the operating system's memory; in Symbian OS, the implementation of file system tables is unknown, and all file system manipulation is done through objects of a specific file class.

Note that Symbian OS differs from other operating systems that use object-oriented concepts in design. For example, many operating system designs use abstract data types; one could even argue that the whole idea of a system call implements abstraction by hiding the details of system implementation from user

programs. In Symbian OS, object orientation is designed into the entire operating system framework. Operating system functionality and system calls are always associated with system objects. Resource allocation and protection is focused on the allocation of objects, not on implementation of system calls.

12.2.2 Microkernel Design

Building upon the object-oriented nature of the operating system, the kernel structure of Symbian OS has a microkernel design. Minimal system functions and data are in the kernel; many system functions have been pushed out to user-space servers. The servers do their work by obtaining handles to system objects and making system calls through these objects into the kernel when necessary. User-space applications interact with these servers rather than make system calls.

Microkernel-based operating systems typically take up much less memory upon boot and their structure is more dynamic. Servers can be started as needed; not all servers are required at boot time. Microkernels usually implement a pluggable architecture with support for system modules that can be loaded needed and plugged into the kernel. Thus, microkernels are very flexible: code to support new functionality (for example, new hardware drivers) can be loaded and plugged in any time.

Symbian OS was designed as a microkernel-based operating system. Access to system resources is done by opening connections to resource servers that in turn coordinate access to the resources themselves. Symbian OS sports a pluggable architecture for new implementations. New implementations for system functions can be designed as system objects and dynamically inserted into the kernel. For example, new file systems can be implemented and added to the kernel as the operating system is running.

This microkernel design carries some issues. Where a single system call is sufficient for a conventional operating system, a microkernel uses message passing. Performance can suffer because of the added overhead of communication between objects. The efficiency of functions that stay in kernel space in conventional operating systems is diminished when those functions are moved to user space. For example, the overhead of multiple function calls to schedule processes can diminish performance when compared to process scheduling in Windows kernel that has direct access to kernel data structures. Since messages pass between user space and kernel space objects, switches in privilege levels are likely to occur, further complicating performance. Finally, where system calls work in a single address space for conventional designs, this message passing and privilege switching implies that two or more address spaces must be used to implement a microkernel service request.

These performance issues have forced the designers of Symbian OS (as well as other microkernel based systems) to pay careful attention to design and implementation details. The emphasis of design is for minimal, tightly focused servers.

12.2.3 The Symbian OS Nanokernel

Symbian OS designers have addressed microkernel issues by implementing a **nanokernel** structure at the core of the operating system's design. Just as certain system functions are pushed into user-space servers in microkernels, the design of Symbian OS separates functions that require complicated implementation into the Symbian OS kernel and keeps only the most basic functions in the nanokernel, the operating system's core.

The nanokernel provides some of most basic functions in Symbian OS. In the nanokernel, simple threads operating in privileged mode implement services that are very primitive. Included among the implementations at this level are scheduling and synchronization operations, interrupt handling, and synchronization objects, such as mutexes and semaphores. Most of the functions implemented at this level are preemptible. Functions at this level are very primitive (so that they can be fast). For example, dynamic memory allocation is a function too complicated for a nanokernel operation.

This nanokernel design requires a second level to implement more complicated kernel functions. The **Symbian OS kernel layer** provides the more complicated kernel functions that are needed by the rest of the operating system. Each operation at the Symbian OS kernel level is a privileged operation and combines with the primitive operations of the nanokernel to implement more complex kernel tasks. Complex object services, user-mode threads, process scheduling and context switching, dynamic memory, dynamically loaded libraries, complex synchronization, objects and interprocess communication are just some of the operations implemented by this layer. This layer is fully preemptible, and interrupts can cause this layer to reschedule any part of its execution even in the middle of context switching!

Fig. 12-1 shows a diagram of the complete Symbian OS kernel structure.

12.2.4 Client/Server Resource Access

As we mentioned, Symbian OS exploits its microkernel design and uses a client/server model to access system resources. Applications that need to access system resources are the clients; servers are programs that the operating system runs to coordinate access to these resources. Where in Linux one might call open to open a file or in Windows one might use a Microsoft API to create a window, in Symbian OS both sequences are the same: first a connection must be made to a server, the server must acknowledge the connection, and requests are made to the server to perform certain functions. So opening a file means finding the file server, calling connect to set up a connection to the server, and then sending the server an open request with the name of a specific file.

There are several advantages of this way of protecting resources. First, it fits with the design of the operating system—both as an object oriented system and as

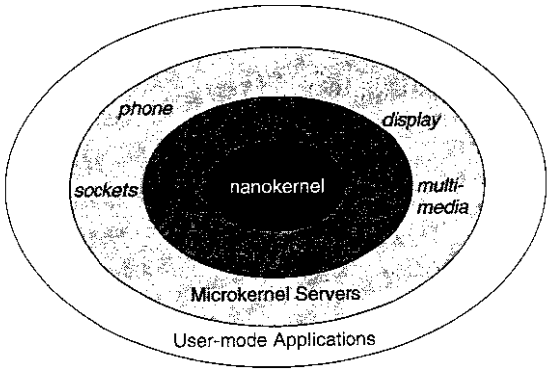

Figure 12-1. The Symbian OS kernel structure has many layers.

a microkernel-based system. Second, this type of architecture is quite effective for managing the multiple accesses to system resources that a multitasking and multithreaded operating system would require. Finally, each server is able to focus on the resources it must manage and can be easily upgraded and swapped out for new designs.

12.2.5 Features of a Larger Operating System

Despite the size of its target computers, Symbian OS has many of the features of its larger siblings. While you can expect to see the kind of support you see on larger operating systems like Linux and Windows, you should expect to find these features in a different form. Symbian OS has many features in common with larger operating systems.

Processes and Threads: Symbian OS is a multitasking and multithreaded operating system. Many processes can run concurrently, can communicate with each other, and can utilize multiple threads that run internal to each process.

Common File System Support: Symbian OS organizes access to system storage using a file system model, just like larger operating systems. It has a default file system compatible with Windows (by default, it uses a FAT-32 file system); it supports other file system implementations using a plug-in style interface. Symbian OS supports several different types of file systems, including FAT-16 and FAT-32, NTFS, and many storage card formats (for example, JFFS).

Networking: Symbian OS supports TCP/IP networking as well as several other communication interfaces, such as serial, infrared, and Bluetooth.

Memory management: Although Symbian OS does not use (or have the facilities for) mapped virtual memory, it organizes memory access in pages and allows for the replacement of pages, that is, bringing pages in, but not swapping them out.

12.2.6 Communication and Multimedia

Symbian OS was built to facilitate communication in many forms. We can hardly provide an overview of it without mentioning communication features. The modeling of communication conforms to both object orientation and a microkernel, client/server architecture. Communication structures in Symbian OS are built in modules, allowing new communication mechanisms to be grafted into the operating system easily. Modules can be written to implement anything from user-level interfaces to new protocol implementations to new device drivers. Because of the microkernel design, new modules can be introduced and loaded into the operation of the system dynamically.

Symbian OS has some unique features that come from its focus on the smartphone platform. It has a pluggable messaging architecture—one where new message types can be invented and implemented by developing modules that are dynamically loaded by the messaging server. The messaging system has been designed in layers, with specific types of object implementing the various layers. For example, message transport objects are separate from message type objects. A form of message transport, say, cellular wireless transport (like CDMA), could transport several different types of messages (standard text message types, SMS types, or system commands like BIO messages). New transport methods can be introduced by implementing a new object and loading it into the kernel.

Symbian OS has been designed at its core with APIs specialized for multimedia. Multimedia devices and content are handled by special servers and by a framework that lets the user implement modules that describe new and existing content and what to do with it. In much the same way messaging is implemented, multimedia is supported by various forms of objects, designed to interact with each other. The way sound is played is designed as an object that interacts with the way each sound format is implemented.

12.3 PROCESSES AND THREADS IN SYMBIAN OS

Symbian OS is a multitasking operating system that uses the concepts of processes and threads much like other operating systems do. However, the structure of the Symbian OS kernel and the way it approaches the possible scarcity of resources influences the way that it views these multitasking objects.

12.3.1 Threads and Nanothreads

Instead of processes as the basis for multitasking, Symbian OS favors threads and is built around the thread concept. Threads form the central unit of multitasking. A process is simply seen by the operating system as a collection of threads with a process control block and some memory space.

Thread support in Symbian OS is based in the nanokernel with **nanothreads**. The nanokernel provides only simple thread support; each thread is supported by a nanokernel-based nanothread. The nanokernel provides for nanothread scheduling, synchronization (interthread communication), and timing services. Nanothreads run in privileged mode and need a stack to store their run-time environment data. Nanothreads cannot run in user mode. This fact means that the operating system can keep close, tight control over each one. Each nanothread needs a very minimal set of data to run: basically, the location of its stack and how big that stack is. The operating system keeps control of everything else, such as the code each thread uses, and stores a thread's context on its run-time stack.

Nanothreads have thread states like processes have states. The model used by the Symbian OS nanokernel adds a few states to the basic model. In addition to the basic states, nanothreads can be in the following states:

Suspended. This is when a thread suspends another thread and is meant to be different from the waiting state, where a thread is blocked by some upper layer object (e.g., a Symbian OS thread).

Fast Semaphore Wait. A thread in this state is waiting for a fast semaphore—a type of sentinel variable—to be signaled. Fast semaphores are nanokernel level semaphores.

DFC Wait. A thread in this state is waiting for a delayed function call or DFC to be added to the DFC queue. DFCs are used in device driver implementation. They represent calls to the kernel that can be queued and scheduled for execution by the Symbian OS kernel layer.

Sleep. Sleeping threads are waiting for a specific amount of time to elapse.

Other. There is a generic state that is used when developers implement extra states for nanothreads. Developers do this when they extend the nanokernel functional for new phone platforms (called personality layers). Developers who do this must also implement how states are transitioned to and from their extended implementations.

Compare the nanothread idea with the conventional idea of a process. A nanothread is essentially an ultra light-weight process. It has a mini-context that gets switched as nanothreads get moved onto and out of the processor. Each nanothread has a state, as do processes. The keys to nanothreads are the tight control that the nanokernel has over them and the minimal data that make up the context of each one.

Symbian OS threads build upon nanothreads; the kernel adds support beyond what the nanokernel provides. User mode threads that are used for standard applications are implemented by Symbian OS threads. Each Symbian OS thread contains a nanothread and adds its own run-time stack to the stack the nanothread uses. Symbian OS threads can operate in kernel mode via system calls. Symbian OS also add exception handling and exit signaling to the implementation.

Symbian OS threads implement their own set of states on top of the nanothread implementation. Because Symbian OS threads add some functionality to the minimal nanothread implementation, the new states reflect the new ideas built into Symbian OS threads. Symbian OS adds seven new states that Symbian OS threads can be in, focused on special blocking conditions that can happen to a Symbian OS thread. These special states include waiting and suspending on (normal) semaphores, mutex variables, and condition variables. Remember that, because of the implementation of Symbian OS threads on top of nanothreads, these states are implemented in terms of nanothread states, mostly by using the suspended nanothread state in various ways.

12.3.2 Processes

Processes in Symbian OS, then, are Symbian OS threads grouped together under a single process control block structure with a single memory space. There may be only a single thread of execution or there may be many threads under one process control block. Concepts of process state and process scheduling have already been defined by Symbian OS threads and nanothreads. Scheduling a process, then, is really implemented by scheduling a thread and initializing the right process control block to use for its data needs.

Symbian OS threads organized under a single process work together in several ways. First, there is a single main thread that is marked as the starting point for the process. Second, threads share scheduling parameters. Changing parameters, that is, the method of scheduling, for the process changes the parameters for all threads. Third, threads share memory space objects, including device and other object descriptors. Finally, when a process is terminated, the kernel terminates all threads in the process.

12.3.3 Active Objects

Active objects are specialized forms of threads, implemented in a such a way as to lighten the burden they place on the operating environment. The designers of Symbian OS recognized the fact that there would be many situations where a

thread in an application would block. Since Symbian OS is focused on communication, many applications have a similar pattern of implementation: they write data to a communication socket or send information through a pipe, and then they block as they wait for a response from the receiver. Active objects are designed so that when they are brought back from this blocked state, they have a single entry point into their code that is called. This simplifies their implementation. Since they run in user space, active objects have the properties of Symbian OS threads. As such they have their own nanothread and can join with other Symbian OS threads to form a process to the operating system.

If active objects are just Symbian OS threads, one can ask what advantage the operating system gains from this simplified thread model. The key to active objects is in scheduling. While waiting for events, all active objects reside within a single process and can act as a single thread to the system. The kernel does not need to continually check each active object to see if it can be unblocked. Active objects in a single process, therefore, can be coordinated by a single scheduler implemented in a single thread. By combining code that would otherwise be implemented as multiple threads into one thread, by building fixed entry points into the code, and by using a single scheduler to coordinate their execution, active objects form an efficient and lightweight version of standard threads.

It is important to realize where active objects fit into the Symbian OS process structure. When a conventional thread makes a system call that blocks its execution while in the waiting state, the operating system still needs to check the thread. Between context switches, the operating system will spend time checking blocked processes in the wait state, determining if any needs to move to the ready state. Active objects place themselves in the wait state and wait for a specific event. Therefore, the operating system does not need to check them but moves them when their specific event has been triggered. The result is less thread checking and faster performance.

12.3.4 Interprocess Communication

In a multithreaded environment like Symbian OS, interprocess communication is crucial to system performance. Threads, especially in the form of system servers, communicate constantly.

A **socket** is the basic communication model used by Symbian OS. It is an abstract communication pipeline between two endpoints. The abstraction is used to hide both the methods of transport and the management of data between the endpoints. The concept of a socket is used by Symbian OS to communicate between clients and servers, from threads to devices, and between threads themselves.

The socket model also forms the basis of device I/O. Again abstraction is the key to making this model so useful. All the mechanics of exchanging data with a device are managed by the operating system rather than by the application. For

example, sockets that work over TCP/IP in a networking environment can be easily adapted to work over a Bluetooth environment by changing parameters in the type of socket used. Most of the rest of the data exchange work in such a switchover is done by the operating system.

Symbian OS implements the standard synchronization primitives that one would find in a general purpose operating system. Several forms of semaphores and mutexes are in wide use across the operating system. These provide for synchronizing processes and threads.

12.4 MEMORY MANAGEMENT

Memory management in systems like Linux and Windows employs many of the concepts we have written about to implement management of memory resources. Concepts such as virtual memory pages built from physical memory frames, demand-paged virtual memory, and dynamic page replacement combine to give the illusion of near limitless memory resources, where physical memory is supported and extended by storage such as hard disk space.

As an effective general-purpose operating system, Symbian OS must also provide a memory management model. However, since storage on smartphones is usually quite limited, the memory model is restricted and does not use a virtual memory/swap space model for its memory management. It does, however, use mostof the other mechanisms we have discussed for managing memory, including hardware MMUs.

12.4.1 Systems with No Virtual Memory

Many computer systems do not have the facilities to provide full-blown virtual memory with demand paging. The only storage available to the operating system on these platforms is memory; they do not come with a disk drive. Because of this, most smaller systems, from PDAs to smartphones to higher-level handheld devices, do not support a demand-paged virtual memory.

Consider the memory space used in most small platform devices. Typically, these systems have two types of storage: RAM and flash memory. RAM stores the operating system code (to be used when the system boots); flash memory is used for both operating memory and permanent (file) storage. Often, it is possible to add extra flash memory to a device (such as a Secure Digital card), and this memory is used exclusively for permanent storage.

The absence of demand-paged virtual memory does not mean the absence of memory management. In fact, most smaller platforms are built on hardware that includes many of the management features of larger systems. This includes features such as paging, address translation, and virtual /physical address abstraction. The absence of virtual memory simply means that pages cannot be swapped from

memory and stored in external storage, but the abstraction of memory pages is still used. Pages are replaced, but the page being replaced is just discarded. This means that only code pages can be replaced, since only they are backed on the flash memory.

Memory management consists of the following tasks:

Management of application size: The size of an application—both code and data—has a strong effect on how memory is used. It requires skill and discipline to create small software. The push to use object-oriented design can be an obstacle here (more objects means more dynamic memory allocation, which means larger heap sizes). Most operating systems for smaller platforms heavily discourage static linking of any modules.

Heap management: The heap—the space for dynamic memory allocation—must be managed very tightly on a smaller platform. Heap space is typically bounded on smaller platforms to force programmers to reclaim and reuse heap space as much as possible. Venturing beyond the boundaries results in errors in memory allocation.

Execution in-place: Platforms with no disk drives usually support execution in-place. What this means is that the flash memory is mapped into the virtual address space and programs can be executed directly from flash memory, without copying them into RAM first. Doing so reduces load time to zero, allowing applications to start instantly, and also does not require tying up scarce RAM.

Loading DLLs: The choice of when to load DLLs can affect the perception of system performance. Loading all DLLs when an application is first loaded into memory, for example, is more acceptable than loading them at sporadic times during execution. Users will better accept lag time in loading an application than delays in execution. Note that DLLs may not need to be loaded. This might be the case if (a) they are already in memory or (b) they are contained on external flash storage (in which case, they can be executed in place).

Offload memory management to hardware: If there is an available MMU, it is used to its fullest extent. In fact, the more functionality that can be put into an MMU, the better off system performance will be.

Even with the execution in-place rule, small platforms still need memory that is reserved for operating system operation. This memory is shared with permanent storage and is typically managed in one of two ways. First, a very simple approach is taken by some operating systems and memory is not paged at all. In these types of systems, context switching means allocating operating space, heap space, for instance, and sharing this operating space between all processes. This

method uses little to no protection between process memory areas and trusts processes to function well together. Palm OS takes this simple approach to memory management. The second method takes a more disciplined approach. In this method, memory is sectioned into pages and these pages are allocated to operating needs. Pages are kept in a free list managed by the operating system and are allocated as needed to both the operating system and user processes. In this approach (because there is no virtual memory) when the free list of pages is exhausted, the system is out of memory and no more allocation can take place. Symbian OS is an example of this second method.

12.4.2 How Symbian OS Addresses Memory

Since Symbian OS is a 32-bit operating system, addresses can range up to 4 GB. It employs the same abstractions as larger systems: programs must use virtual addresses, which get mapped by the operating system to physical addresses. As with most systems, Symbian OS divides memory into virtual pages and physical frames. Frame size is usually 4 KB, but can be variable.

Since there can be up to 4 GB of memory, a frame size of 4 KB means a page table with over a million entries. With limited sizes of memory, Symbian OS cannot dedicate 1 MB to the page table. In addition, the search and access times for such a large table would be a burden to the system. To solve this, Symbian OS adopts a two-level page table strategy, as shown in Fig. 12-2. The first level, called the **page directory**, provides a link to the second level and is indexed by a portion of the virtual address (first 12 bits). This directory is kept in memory and is pointed to by the **TTBR** (**translation table base register**). A page directory entry points into the second level, which is a collection of page tables. These tables provide a link to a specific page in memory and are indexed by a portion of the virtual address (middle 8 bits). Finally, the word in the page referenced is indexed by the low-order 12 bits of the virtual address. Hardware assists in this virtual-to-physical address mapping calculation. While Symbian OS cannot assume the existence of any kind of hardware assistance, most of the architectures it is implemented for have MMUs. The ARM processor, for example, has an extensive MMU, with a translation lookaside buffer to assist in address computation.

When a page is not in memory, an error condition occurs because all application memory pages should be loaded when the application is started (no demand paging). Dynamically loaded libraries are pulled into memory explicitly by small stubs of code linked into the application executable, not by page faults.

Despite the lack of swapping, memory is surprisingly dynamic in Symbian OS. Applications are context switched through memory and, as stated above, have their memory requirements loaded into memory when they start execution. The memory pages each application requires can be statically requested from the operating system upon loading into memory. Dynamic space—that is, for the heap—is bounded, so static requests can be made for dynamic space as well. Memory

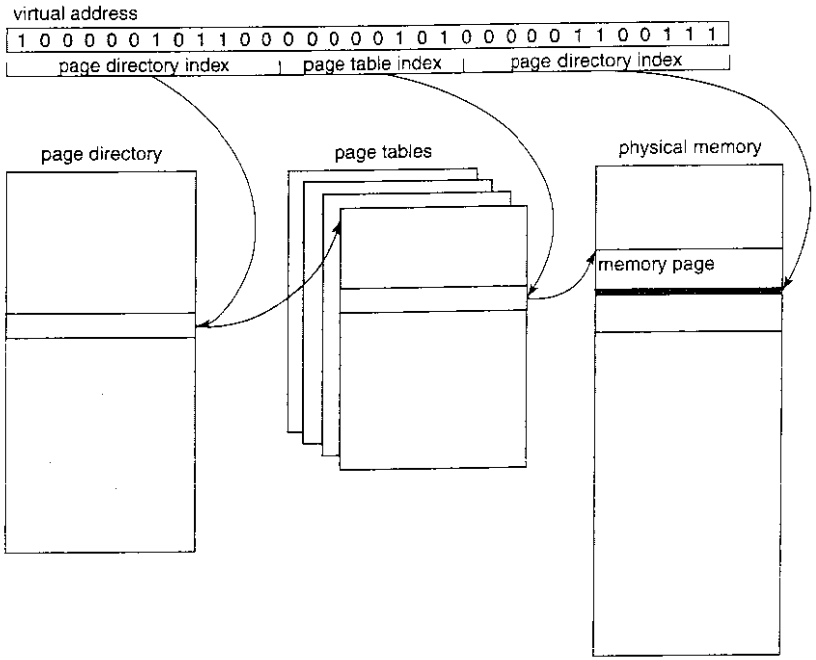

Figure 12-2. Symbian OS uses a two-level page table to reduce table access time and storage.

frames are allocated to pages from a list of free frames; if no free frames are available, then an error condition is raised. Memory frames that are used cannot be replaced with pages from an incoming application, even if the frames are for an application that is not executing currently. This is because there is no swapping in Symbian OS and there is no place to copy the evicted pages to, as the (very limited) flash memory is only for user files.

There are actually four different versions of the memory implementation model that Symbian OS uses. Each model was designed for certain types of hardware configuration. A brief listing of these is below:

The moving model: This model was designed for early ARM architectures. The page directory in the moving model is 4-KB long, and each entry holds 4 bytes, giving the directory a 16-KB size. Memory pages are protected by access bits associated with memory frames and by labeling memory access with a domain. Domains are recorded in the page directory and the MMU enforces access permissions for each domain. While segmentation is not explicitly used, there is an organization to the layout of memory: there is a data section for user-allocated data and a kernel section for kernel-allocated data.

The multiple model: This model was developed for versions 6 and later of the ARM architecture. The MMU in these versions differs from that used in earlier versions. For example, the page directory requires different handling, since it can be sectioned into two pieces, each referencing two different sets of page tables. These two are used for user page tables and for kernel page tables. The new version of the ARM architecture revised and enhanced the access bits on each page frame and deprecated the domain concept.

The direct model: The direct memory model assumes that there is no MMU at all. This model is rarely used and is not allowed on smartphones. The lack of an MMU would cause severe performance issues. This model is useful for development environments where the MMU must be disabled for some reason.

The emulator model: This model was developed to support the Windows-hosted Symbian OS emulator. The emulator has few restrictions in comparison to a real target CPU. The emulator runs as a single Windows process, therefore the address space is restricted to 2 GB, not 4 GB. All memory provided to the emulator is accessible to any Symbian OS process and therefore no memory protection is available. Symbian OS libraries are provided as Windows-format DLLs and therefore Windows handles the allocation and management of memory.

12.5 INPUT AND OUTPUT

Symbian OS's input/output structure mirrors that of other operating system designs. This section will point out some of the unique characteristics that Symbian OS uses to focus on its target platform.

12.5.1 Device Drivers

In Symbian OS, device drivers execute as kernel-privileged code to give user-level code access to system-protected resources. As with Linux and Windows, device drivers represent software access to hardware.

A device driver in Symbian OS is split into two levels: a logical device driver (LDD) and a physical device driver (PDD). The LDD presents an interface to upper layers of software, while the PDD interacts directly with hardware. In this model, the LDD can use the same implementation for a specific class of devices, while the PDD changes with each device. Symbian OS supplies many standard LDDs. Sometimes, if the hardware is fairly standard or common, Symbian OS will also supply a PDD.

Consider an example of a serial device. Symbian OS defines a generic serial LDD that defines the program interfaces for accessing the serial device. The LDD supplies an interface to the PDD, which provides the interface to serial devices. The PDD implements buffering and the flow control mechanisms necessary to help regulate the differences in speed between the CPU and serial devices. A single LDD (the user side) can connect to any of the PDDs that might be used to run serial devices. On a specific smartphone, these might include an infrared port or even an RS-232 port. These two are good examples; they use the same serial LDD, but different PDDs.

LDDs and PDDs can be dynamically loaded by user programs if they are not already existing in memory. Programming facilities are provided to check to see if loading is necessary.

12.5.2 Kernel Extensions

Kernel extensions are device drivers that are loaded by Symbian OS at boot time. Because they are loaded at boot time, they are special cases that need to be treated differently than normal device drivers.

Kernel extensions are different from normal device drivers. Most device drivers are implemented as LDDs, paired with PDDs, and are loaded when needed by user-space applications. Kernel extensions are loaded at boot time and are specifically targeted at certain devices, typically not paired with PDDs.

Kernel extensions are built into the boot procedure. These special device drivers are loaded and started just adter the scheduler starts. These implement functions that are crucial to operating systems: DMA services, display management, bus control to peripheral devices (e.g., the USB bus). These are provided for two reasons. First, it matches the object-oriented design abstractions we have come to see as characteristic of microkernel design. Second, it allows the separate platforms that Symbian OS runs on to run specialized device drivers that enable the hardware for each platform without recompiling the kernel.

12.5.3 Direct Memory Access

Device drivers frequently make use of DMA and Symbian OS supports the use of DMA hardware. DMA hardware consists of a controller that controls a set of DMA channels. Each channel provides a single direction of communication between memory and a device; therefore, bidirectional transmission of data requires two DMA channels. At least one pair of DMA channels is dedicated to the screen LCD controller. In addition, most platforms provide a certain number of general DMA channels.

Once a device has transmitted data to memory, a system interrupt is triggered. The DMA service provided by DMA hardware is used by the PDD for the transmitting device—the part of the device driver that interfaces with the hardware. Between the PDD and the DMA controller, Symbian OS implements two layers of software: a software DMA layer and a kernel extension that interfaces with the DMA hardware. The DMA layer is itself split up into a platform-independent layer and a platform-dependent layer. As a kernel extension, the DMA layer is one of the first device drivers to be started by kernel during the boot procedure.

Support for DMA is complicated for a special reason. Symbian OS supports many difference hardware configurations and no single DMA configuration can be assumed. The interface to the DMA hardware is standardized across platforms, and is supplied in the platform-independent layer. The platform-dependent layer and the kernel extension are supplied by the manufacturer, thus treating the DMA hardware the same way as Symbian OS treats any other device: with a device driver in LDD and PDD components. Since the DMA hardware is viewed as a device in its own right, this way of implementing support makes sense because it parallels the way Symbian OS supports all devices.

12.5.4 Special Case: Storage Media

Media drivers are a special form of PDD in Symbian OS that are used exclusively by the file server to implement access to storage media devices. Because smartphones can contain both fixed and removable media, the media drivers must recognize and support a variety of forms of storage. Symbian OS support for media includes a standard LDD and an interface API for users.

The file server in Symbian OS can support up to 26 different drives at the same time. Local drives are distinguished by their drive letter, as in Windows.

12.5.5 Blocking I/O

Symbian OS deals with blocking I/O through active objects. The designers realized that the weight of all threads waiting on an I/O event affects the other threads in the system. Active objects allow blocking I/O calls to be handled by the operating system rather than the process itself. Active objects are coordinated by a single scheduler and implemented in a single thread.

When the active object uses a blocking I/O call, it signals the operating system and suspends itself. When the blocking call completes, the operating system wakes up the suspended process, and that process continues execution as if a function had returned with data. The difference is one of perspective for the active object. It cannot call a function and expect a return value. It must call a special function and let that function set up the blocking I/O but return immediately. The operating system takes over the waiting.

12.5.6 Removable Media

Removable media pose an interesting dilemma for operating system designers. When a Secure Digital card is inserted in its reader slot, it is a device just like all others. It needs a controller, a driver, a bus structure, and will probably communicate to the CPU through DMA. However, the fact that you remove the media is a serious problem to this device model: how does the operating system detect insertion and removal, and how should the model accommodate the absence of a media card? To get even more complicated, some device slots can accommodate more than one kind of device. For example, an SD card, a miniSD card (with an adapter), and a MultiMediaCard all use the same kind of slot.

Symbian OS starts its implementation of removable media with their similarities. Each type of removable media have features common to all of them:

1. All devices must be inserted and removed.
2. All removable media can be removed "hot," that is, while being used.
3. Each medium can report its capabilities.
4. Incompatible cards must be rejected.
5. Each card needs power.

To support removable media, Symbian OS provides software controllers that control each supported card. The controllers work with device drivers for each card, also in software. There is a socket object created when a card is inserted and this object forms the channel over which data flows. To accommodate the changes in the card's state, Symbian OS provides a series of events that occur when state changes happen. Device drivers are configured like active objects to listen for and respond to these events.

12.6 STORAGE SYSTEMS

Like all user-oriented operating systems, Symbian OS has a file system. We will describe it below.

12.6.1 File Systems for Mobile Devices

In terms of file systems and storage, mobile phone operating systems have many of the requirements of desktop operating systems. Most are implemented in 32-bit environments; most allow users to give arbitrary names to files; most store many files that require some kind of organized structure. This means that a hierarchical directory-based file system is desirable. And while designers of mobile

operating systems have many choices for file systems, one more characteristic influences their choice: most mobile phones have storage media that can be shared with a Windows environment.

If mobile phone systems did not have removable media, then any file system would be usable. However, for systems that use flash memory, there are special circumstances to consider. Block sizes are typically from 512 bytes to 2048 bytes. Flash memory cannot simply overwrite memory; it must erase first, then write. In addition, the unit of erasure is rather coarse: bytes cannot be erased but entire blocks must be erased at a time. Erase times for flash memory are relatively long.

To accommodate these characteristics, flash memory works best when there are specifically designed file systems that spread writes over the media and deal with the long erase times. The basic concept is that when the flash store is to be updated, the file system will write a new copy of the changed data over to a fresh block, remap the file pointers, and then erase the old block later when it has time.

One of the earliest flash file systems was Microsoft's FFS2 for use with MS-DOS in the early 1990s. When the PCMCIA industry group approved the Flash Translation Layer specification for flash memory in 1994, flash devices could look like a FAT file system. Linux also has specially designed file systems, from JFFS to YAFFS (the Journaling Flash File System and the Yet Another Flash Filing System).

However, mobile platforms must share their media with other computers, which demands that some form of compatibility be in place. Most often, FAT file systems are used. Specifically, FAT-16 is used for its shorter allocation table (rather than FAT-32) and for its reduced usage of long files.

12.6.2 Symbian OS File Systems

Being a mobile smartphone operating system, Symbian OS needs to implement at least the FAT-16 file system. Indeed, it provides support for FAT-16 and uses that file system for most of its storage medium.

However, the Symbian OS file server implementation is built on an abstraction much like the Linux virtual file system. Object orientation allows objects that implement various operating systems to be plugged into the Symbian OS file server, thus allowing many different file system implementations to be used. Different implementations may even co-exist in the same file server.

Implementations of NFS and SMB file systems have been created for Symbian OS.

12.6.3 File System Security and Protection

Smartphone security is an interesting variation on general computer security. There are several aspects of smartphones that make security something of a challenge. Symbian OS has made several design choices that differentiate it from

general-purpose desktop systems and from other smartphone platforms. We will focus on those aspects that pertain to file system security; other issues are dealt with in the next section.

Consider the environment for smartphones. They are single-user devices and require no user identification to use. A phone user can execute applications, dial the phone, and access networks—all without identification. In this environment, using permissions-based security is challenging, because the lack of identification means only one set of permissions is possible—the same set for everyone.

Instead of user permissions, security often takes advantage of other types of information. In Symbian OS version 9 and later, applications are given a set of capabilities when they are installed. (The process that grants which capabilities an application has is covered in the next section.) This capability set for an application is matched against the access that the application requests. If the access is in the capability set, then access is granted; otherwise, it is refused. Capability matching requires some overhead—matching occurs at every system call that involves access to a resource—but the overhead of matching file ownership with a file's owner is gone. The trade-off works well for Symbian OS.

There are some other forms of file security on Symbian OS. There are areas of the Symbian OS storage medium that applications cannot access without special capability. This special capability is only provided to the application that installs software onto the system. The effect of this is that new applications, after being installed, are protected from nonsystem access (meaning that nonsystem malicious programs, such as viruses, cannot infect installed applications). In addition, there are areas of the file system reserved specifically for certain types of data manipulation by application (this is called data caging; see the next section).

For Symbian OS, the use of capabilities has worked as well as file ownership for protecting access to files.

12.7 SECURITY IN SYMBIAN OS

Smartphones provide a difficult environment to make secure. As we discussed previously, they are single-user devices and require no user authentication to use basic functions. Even more complicated functions (such as installing applications) require authorization but no authentication. However, they run on complex operating systems with many ways to bring data (including executing programs) in and out. Safeguarding these environments is complicated.

Symbian OS is a good example of this difficulty. Users expect that Symbian OS smartphones will allow any kind of use without authentication—no logging in or verifying your identity. Yet, as you have undoubtably experienced, an operating system as complicated as Symbian OS is very capable yet also susceptible to viruses, worms, and other malicious programs. The versions of Symbian OS prior to version 9 offered a gatekeeper type of security: the system asked the user for

permission to install every installed application. The thinking in this design was that only user-installed applications could cause system havoc and an informed user would know what programs he intended to install and what programs were malicious. The user was trusted to use them wisely.

This gatekeeper design has a lot of merit. For example, a new smartphone with no user-installed applications would be a system that could run without error. Installing only applications that a user knew were not malicious would maintain the security of the system. The problem with this design is that users do not always know the complete ramifications of the software they are installing. There are viruses that masquerade as useful programs, performing useful functions while silently installing malicious code. Normal users are unable to verify the complete trustworthiness of all the software available.

This verification of trust is what prompted a complete redesign of platform security for Symbian OS version 9. This version of the operating system keeps the gatekeeper model, but takes the responsibility for verifying software away from the user. Each software developer is now responsible for verifying its own software through a process called **signing** and the system verifies the developer's claim. Not all software requires such verification, only those that access certain system functions. When an application requires signing, this is done through a series of steps:

1. The software developer must obtain a vendor ID from a trusted third party. These trusted parties are certified by Symbian.
2. When a developer has developed a software package and wants to distribute it, the developer must submit the package to a trusted third party for validation. The developer submits his vendor ID, the software, and a list of ways that the software accesses the system.
3. The trusted third party then verifies that the list of software access types is complete and that no other type of access occurs. If the third party can make this verification, the software is then signed by that third party. This means that the installation package has a special amount of information that details what it will do to a Symbian OS system and that it may actually do.
4. This installation package is sent back to the software developer and may now be distributed to users. Note that this method depends on how software accesses system resources. Symbian OS says that in order to access a system resource, a program must have the capability to access the resource. This idea of capabilities is built into the kernel of Symbian OS. When a process is created, part of its process control block records the capabilities granted to the process. Should the process try to perform an access that was not listed in these capabilities, the access would be denied by the kernel.

The result of this seemingly elaborate process to distribute signed applications is a trust system in which an automated gatekeeper built into Symbian OS can verify software to be installed. The install process checks the signage of the installation package. If the signing of the package is valid, the capabilities granted the software are recorded and these are the capabilities granted to the application by the kernel when it executes.

The diagram in Fig. 12-3 depicts the trust relationships in Symbian OS version 9. Note here that there are several levels of trust built into the system. There are some applications that do not access system resources at all, and therefore do not require signing. An example of this might be a simple application that only displays something on the screen. These applications are not trusted, but they do not need to be. The next level of trust is made up of user-level signed applications. These signed applications are only granted the capabilities they need. The third level of trust is made up of system servers. Like user-level applications, these servers may only need certain capabilities to perform their duties. In a microkernel architecture like Symbian OS, these servers run at the user level and are trusted like user-level applications. Finally, there is a class of programs that requires full trust of the system. This set of programs has the full ability to change the system and is made up of kernel code.

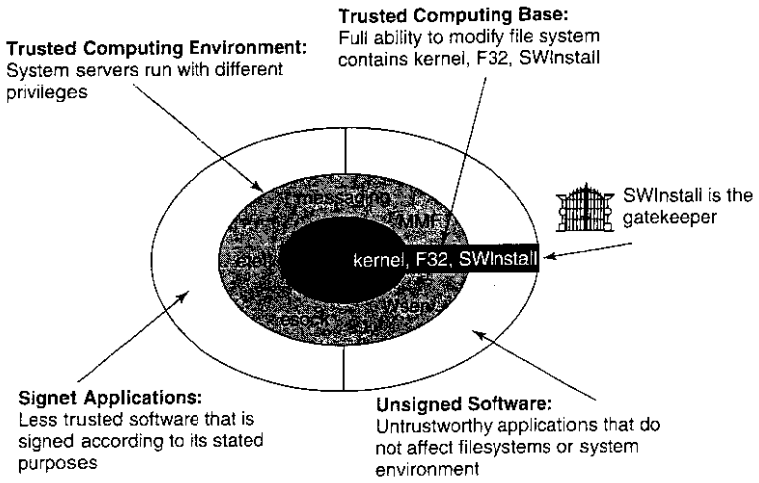

Figure 12-3. Symbian OS uses trust relationships to implement security.

There are several aspects to this system that might seem questionable. For example, is this elaborate process really necessary (especially when it costs some money to do)? The answer is yes: the Symbian signing system replaces users as the verifier of software integrity, and there must be real verification done. This process might seem to make development difficult: does each test on real hardware require a new signed installation package? To answer this, Symbian OS recognizes special signing for developers. A developer must get a special signed digital certificate that is time limited (usually for 6 months) and specific to a particular smartphone. The developer can then build its own installation packages with the digital certificate.

In addition to this gatekeeping function in version 9, Symbian OS also employs something called **data caging**, which organizes data into certain directories. Executable code only exists in one directory, for example, that is writable only by the software installation application. In addition, data written by applications can only be written in one directory, which is private and inaccessible from other programs.

12.8 COMMUNICATION IN SYMBIAN OS

Symbian OS is designed with specific criteria in mind and can be characterized by event-driven communications using client/server relationships and stack-based configurations.

12.8.1 Basic Infrastructure

The Symbian OS communication infrastructure is built on basic components. Consider a very generic form of this infrastructure shown in Fig. 12-4. Consider this diagram as a starting point for an organizational model. At the bottom of the stack is a physical device, connected in some way to the computer. This device could be a mobile phone modem or a Bluetooth radio transmitter embedded in a communicator. We are not concerned with the details of hardware here, so we will treat this physical device as an abstract unit that responds to commands from software in the appropriate manner.

The next level, and the first level we are concerned with, is the device driver level. We have already pointed out the structure of device drivers; software at this level is concerned with working directly with the hardware via the LDD and PDD structures. The software at this level is hardware specific, and every new piece of hardware requires a new software device driver to interface with it. Different drivers are needed for different hardware units, but they all must implement the same interface to the upper layers. The protocol implementation layer will expect the same interface no matter what hardware unit is used.

The next layer is the protocol implementation layer, containing implementations of the protocols supported by Symbian OS. These implementations assume a device driver interface with the layer beneath and supply a single, unified interface to the application layer above. This is the layer that implements the Bluetooth and TCP/IP protocol suites, for example, along with other protocols.

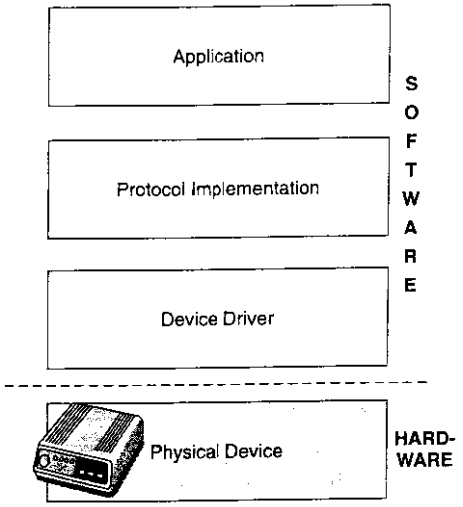

Figure 12-4. Communication in Symbian OS has block oriented structure.

Finally, the application layer is the topmost layer. This layer contains the application that must utilize the communication infrastructure. The application does not know much about how communications are implemented. However, it does do the work necessary to inform the operating system of which devices it will use. Once the drivers are in place, the application does not access them directly, but depends on the protocol implementation layer APIs to drive the real devices.

12.8.2 A Closer Look at the Infrastructure

A closer look at the layers in this Symbian OS communication infrastructure is shown in Fig. 12-5. This diagram is based on the generic model in Fig. 12-4. The blocks from Fig. 12-4 have been subdivided into operational units that depict those used by Symbian OS.

The Physical Device

First, notice that the device has not been changed. As we stated before, Symbian OS has no control over hardware. Therefore, it accommodates hardware through this layered API design, but does not specify how the hardware itself is designed and constructed. This is actually an advantage to Symbian OS and its developers. By viewing hardware as an abstract unit and designing communication around this abstraction, the designers of Symbian OS have ensured that Symbian OS can handle the wide variety of devices that are available now and can also accommodate the hardware of the future.

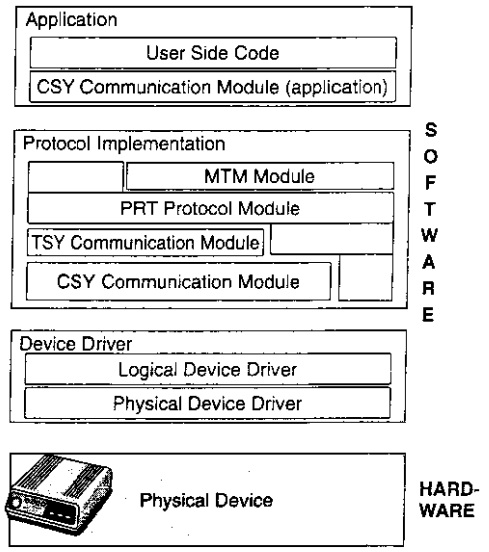

Figure 12-5. Communication structure in Symbian OS has a rich set of features.

The Device Driver Layer

The device driver layer has been divided into two layers in Fig. 12-5. The PDD layer interfaces directly with the physical device, as we mentioned before, through a specific hardware port. The LDD layer interfaces with the protocol implementation layer and implements Symbian OS policies as they relate to the device. These policies include input and output buffering, interrupt mechanisms, and flow control.

The Protocol Implementation Layer

Several sublayers have been added to the protocol implementation layer in Fig. 12-5. Four types of modules are used for protocol implementation; these are itemized below:

CSY Modules: The lowest level in the protocol implementation layers is the communication server, or CSY, module. A CSY module communicates directly with the hardware through the PDD portion of the device driver, implementing the various low-level aspects of protocols. For instance, a protocol may require raw data transfer to the hardware device or it may specify 7-bit or 8-bit buffer transfer. These modes would be handled by the CSY module.

TSY Modules: Telephony comprises a large part of the communications infrastructure, and special modules are used to implement it. Telephony server (TSY) modules implement the telephony functionality. Basic TSYs may support standard telephony functions, such as making and terminating calls, on a wide range of hardware. More advanced TSYs may support advanced phone hardware, such as those supporting GSM functionality.

PRT Modules: The central modules used for protocol implementation are protocol modules, or PRT modules. PRT modules are used by servers to implement protocols. A server creates an instance of a PRT module when it attempts to use the protocol. The TCP/IP suite of protocols, for instance, is implemented by the TCPIP.PRT module. Bluetooth protocols are implemented by the BT.PRT module.

MTMs: As Symbian OS has been designed specifically for messaging, its architects built a mechanism to handle messages of all types. These message handlers are called message type modules, or MTMs. Message handling has many different aspects, and MTMs must implement each of these aspects. User Interface MTMs must implement the various ways users will view and manipulate messages, from how a user reads a message to how a user is notified of the progress of sending a message. Client-side MTMs handle addressing, creating, and responding to messages. Server-side MTMs must implement server-oriented manipulation of messages, including folder manipulation and message-specific manipulation.

These modules build on each other in various ways, depending on the type of communication that is being used. Implementations of protocols using Bluetooth, for example, will use only PRT modules on top of device drivers. Certain IrDA protocols will do this as well. TCP/IP implementations that use PPP will use PRT modules and both a TSY and a CSY module. TCP/IP implementations without PPP will typically not use either a TSY module or a CSY module, but will link a PRT module directly to a network device driver.

Infrastructure Modularity

The modularity of this stack-based model is useful to implementers. The abstract quality of the layered design should be evident from the examples just given. Consider the TCP/IP stack implementation. A PPP connection can go directly to a CSY module or choose a GSM or regular modem TSY implementation, which in turn goes through a CSY module. When the future brings a new telephony technology, this existing structure will still work, and we only need to add a TSY module for the new telephony implementation. In addition, fine tuning the TCP/IP protocol stack does not require altering any of the modules it depends on; we simply tune up the TCP/IP PRT module and leave the rest alone. This extensive modularity means that new code plugs into the infrastructure easily, old code is easily discarded, and existing code can be modified without shaking the whole system or requiring any extensive reinstalls.

Finally, Fig. 12-5 has added sublayers to the application layer. There are CSY modules that applications use to interface with protocol modules in the protocol implementations. While we can consider these as parts of protocol implementations, it is a bit cleaner to think of them as assisting applications. An example here might be an application that uses IR to send SMS messages through a mobile phone. This application would use an IRCOMM CSY module on the application side that uses an SMS implementation wrapped in a protocol implementation layer. Again, the modularity of this entire process is a big advantage for applications that need to focus on what they do best and not on the communications process.

12.9 SUMMARY

Symbian OS was designed as an object-oriented operating system for smartphone platforms. It has a microkernel design that utilizes a very small nanokernel core, implementing only the fastest and most primitive kernel functions. Symbian OS uses a client/server architecture that coordinates access to system resources with user-space servers. While designed for smartphones, Symbian OS has many features of a general-purpose operating system: processes and threads, memory management, file system support, and a rich communication infrastructure. Symbian OS implements some unique features; for example, active objects make waiting on external events much more efficient, the lack of virtual memory makes memory management more challenging, and support for object orientation in device drivers uses a two-layer abstract design.

PROBLEMS

1. For each of the service examples below, describe whether each should be considered a kernel-space or a user-space operation (e.g., in a system server) for a microkernel operating system like Symbian OS:

1. Scheduling a thread for execution
2. Printing a document
3. Answering a Bluetooth discovery query
4. Managing thread access to the screen

5. Playing a sound when a text message arrives
6. Interrupting execution to answer a phone call

2. Itemize three efficiency improvements brought on by a microkernel design.
3. Itemize three efficiency problems brought on by a microkernel design.
4. Symbian OS split its kernel design into two layers: the nanokernel and the Symbian OS kernel. Services like dynamic memory management were deemed too complicated for the nanokernel. Describe the complicated components of dynamic memory management and why they might not work in a nanokernel.
5. We discussed active objects as a way to make I/O processing more efficient. Do you think an application could use *multiple* active objects at the same time? How would the system react when multiple I/O events required action?
6. Security in Symbian OS is focused on installation and Symbian signing of applications? Is this enough? Would there ever be scenario where an application could be placed in storage for execution without being installed? (*Hint*: Think about all possible data entry points for a mobile phone.)
7. In Symbian OS, server-based protection of shared resources is used extensively. List three advantages that this type of resource coordination has in a microkernel environment. Speculate as to how each of your advantages might affect a different kernel architecture.

13

OPERATING SYSTEM DESIGN

In the past 12 chapters, we have covered a lot of ground and taken a look at many concepts and examples relating to operating systems. But studying existing operating systems is different from designing a new one. In this chapter we are going to take a quick look at some of the issues and trade-offs that operating systems designers have to consider when designing and implementing a new system.

There is a certain amount of folklore about what is good and what is bad floating around in the operating systems community, but surprisingly little has been written down. Probably the most important book is Fred Brooks' classic *The Mythical Man Month* in which he relates his experiences in designing and implementing IBM's OS/360. The 20th anniversary edition revises some of that material and adds four new chapters (Brooks, 1995).

Three classic papers on operating system design are "Hints for Computer System Design" (Lampson, 1984), "On Building Systems That Will Fail" (Corbató, 1991), and "End-to-End Arguments in System Design" (Saltzer et al., 1984). Like Brooks' book, all three papers have survived the years extremely well; most of their insights are still as valid now as when they were first published.

This chapter draws upon these sources, plus the author's personal experience as designer or co-designer of three systems: Amoeba (Tanenbaum et al., 1990), MINIX (Tanenbaum and Woodhull, 1997), and Globe (Van Steen et al., 1999a). Since no consensus exists among operating system designers about the best way to design an operating system, this chapter will thus be more personal, speculative, and undoubtedly more controversial than the previous ones.