10

CASE STUDY 1: UNIX AND 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 UNIX because it runs on a wider variety of computers than any other operating system. It is the dominant operating system on high-end workstations and servers, but it is also used on systems ranging from notebook computers to supercomputers. It was carefully designed with a clear goal in mind, and despite its age, is still modern and elegant. Many important design principles are illustrated by UNIX. Quite a few of these have been copied by other systems.

Our discussion of UNIX will start with its history and evolution of the system. Then we will provide an overview of the system, 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 UNIX. For each topic we will first discuss the fundamental concepts, then the system calls, and finally the implementation.

One problem that we will encounter is that there are many versions and clones of UNIX, including AIX, BSD, 1BSD, HP-UX, Linux, MINIX, OSF/1, SCO UNIX, System V, Solaris, XENIX, and various others, and each of these has gone through many versions. 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. In this chapter we will draw upon several examples when discussing implementation, primarily 4.4BSD (which forms the basis for FreeBSD), System V Release 4, and Linux. Additional information about various implementations can be found in (Beck et al., 1998; Goodheart and Cox, 1994; Maxwell, 1999; McKusick et al., 1996; Pate, 1996; and Vahalia, 1996).

10.1 HISTORY OF UNIX

UNIX has a long and interesting history, so we will begin our study there. What started out as the pet project of one young researcher has become a multimillion 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 among 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 do. 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 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. 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 being 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 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 few 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 terminals, 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, a great deal of interest arose in the possibility of connecting these two machines together electronically. UNIX networking traces its roots to this link. 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 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 then 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 complete 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 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 (including 4.1BSD, 4.2BSD, 4.3BSD, and 4.4BSD) 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 two areas in which it most strongly deviates from Version 7 are signals (which is largely taken from BSD) and terminal handling, which is new. 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.

Unfortunately, a funny thing happened on the way back from the standards meeting. Now that the System V versus BSD split had been dealt with, another one appeared. A group of vendors led by IBM, DEC, Hewlett-Packard, and many others did not like the idea that AT&T had control of the rest of UNIX, so they set up a consortium known as **OSF (Open Software Foundation)** to produce a system that met all the IEEE and other standards, but also contained a large number of additional features, such as a windowing system (X11), a graphical user interface (MOTIF), distributed computing (DCE), distributed management (DME), and much more.

AT&T's reaction was to set up its own consortium, **UI (UNIX International)** to do precisely the same thing. UI's version of UNIX was based on System V. The net result is that the world then had two powerful industry groups each offering their own version of UNIX, so the users were no closer to a standard than they were in the beginning. However, the marketplace decided that System V was a better bet than the OSF system, so the latter gradually vanished. Some companies have their own variants of UNIX, such as Sun's Solaris (based on System V).

10.1.6 MINIX

One property that all these 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 to the author of this book writing 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 is in the kernel). Other well-known microkernel designs of this era were Mach (Accetta et al., 1986) and Chorus (Rozier et al., 1988). A discussion of microkernel performance issues is given in (Bricker et al., 1991).

Within a few months of its appearance, MINIX became a bit of a cult item, with its own 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. A book about operating systems principles illustrated using the 500-page MINIX source code given in an appendix and on an accompanying CD-ROM is (Tanenbaum and Woodhull, 1997). MINIX is also available for free on the World Wide Web at URL *www.cs.vu.nl/~ast/minix.html* .

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 (intentionally) lacking. The first version of Linux, 0.01, was released in 1991. It was cross-developed on a MINIX machine and borrowed some 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 size totaled 9,300 lines of C and 950 lines of assembler, roughly similar to MINIX version in size and also roughly comparable in functionality.

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 with no extra features.

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.

A large array of standard UNIX software has been ported to Linux, including over 1000 utility programs, X Windows 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 2000 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 (Free Software

Foundation) online communities. However, as Linux evolves, a steadily smaller fraction of the Linux community want to hack source code (witness 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 forego the free distribution on the Internet to buy one of many CD-ROM distributions available from numerous competing commercial companies. A Web site listing over 50 companies that sell different Linux packages is *www.linux.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 a little.

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 simultaneously 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, this legal action kept FreeBSD off the market just long enough for Linux to get well established. Had the lawsuit not happened, starting around 1993 there would have been a 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?

Given this history, strict POSIX conformance, and overlap between the user communities, it should not come as a surprise that many of Linux' features, system calls, programs, libraries, algorithms, and internal data structures are very similar to those of UNIX. For example, over 80% of the ca. 150 Linux system calls are exact copies of the corresponding system calls in POSIX, BSD, or System V. Thus to a first approximation, much of the description of UNIX given in this chapter also applies to Linux. In places where there are substantial algorithmic differences between UNIX and Linux (e.g., the scheduling algorithm), we will point that out and cover both. Where they are essentially the same, for the sake of brevity we will just refer to UNIX. Be warned that Linux is evolving rapidly, and with thousands of people hacking away on it, some of this material (based on version 2.2) is sure to become obsolete before long.

10.2 OVERVIEW OF UNIX

In this section we will provide a general introduction to UNIX and how it is used, for the benefit of readers not already familiar with it. Although different versions of UNIX differ in subtle ways, the material presented here applies to all of them. The focus here is how UNIX appears at the terminal. Subsequent sections will focus on system calls and how it works inside.

10.2.1 UNIX Goals

UNIX is 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.

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, etc. (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 UNIX 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 UNIX 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 ask if that is correct. While this kind of user interface may or may not be suitable for rank novices, it irritates skilled programmers no end. What they want is a servant, not a nanny.

10.2.2 Interfaces to UNIX

A UNIX 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, terminals, and other devices. Running on the bare hardware is the UNIX 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.

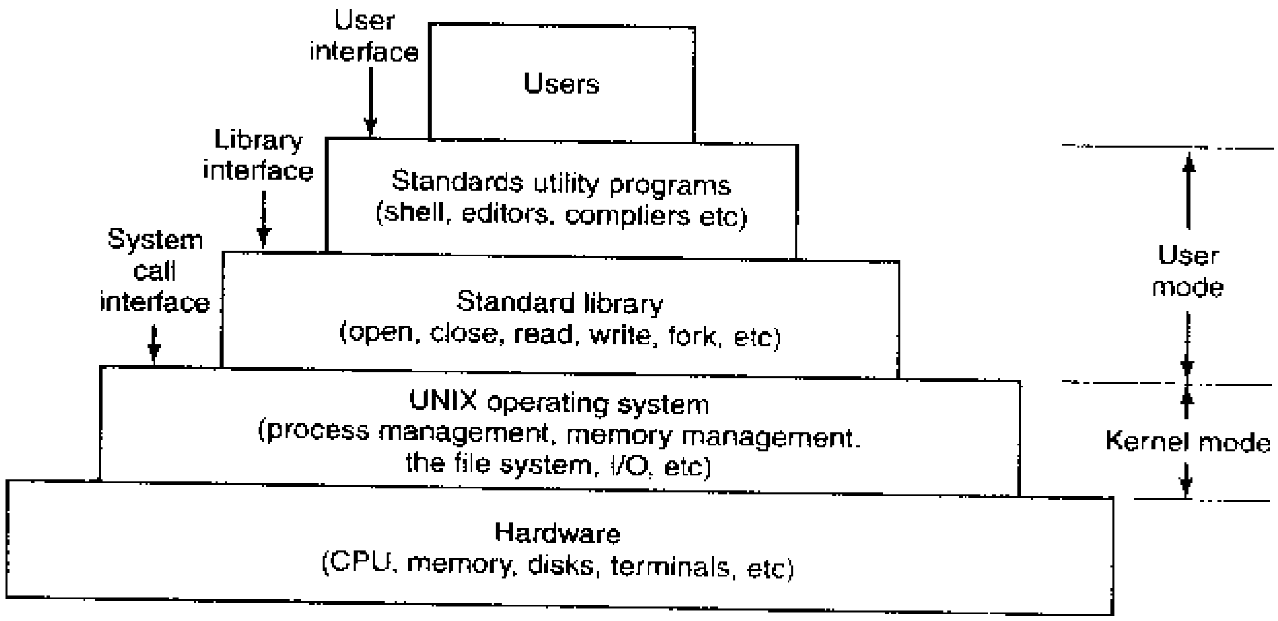

User interface
Users
Library interface
Standards utility programs (shell, editors, compliers etc)
System call interface
Standard library (open, close, read, write, fork, etc)
UNIX operating system (process management, memory management, the file system, I/O, etc)
Hardware (CPU, memory, disks, terminals, etc)
User mode
Kernel mode

Figure 10-1. The layers in a UNIX 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 to start up UNIX. 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 UNIX supply a large number of standard programs, some of which are specified by the POSIX 1003.2 standard, and some of which differ between UNIX versions. These include the command processor (shell), compilers, editors, text processing programs, and file manipulation utilities. It is these programs that a user at a terminal invokes.

Thus we can speak of three different interfaces to UNIX: the true system call interface, the library interface, and the interface formed by the set of standard utility programs. While the latter is what the casual user thinks of as "UNIX," in fact, it has almost nothing to do with the operating system itself and can easily be replaced.

Some versions of UNIX, for example, 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 UNIX so popular and has allowed it to survive numerous changes in the underlying technology so well.

10.2.3 The UNIX Shell

Many UNIX systems have a graphical user interface of the kind made popular by the Macintosh and later Windows. However, real programmers still prefer a command line interface, called the **shell**. It 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 Bourne shell (*sh*). Since then, many new shells have been written (*ksh*, *bash*, etc.). Although UNIX fully supports a graphical environment (X Windows), even in this world many programmers simply make multiple console windows and act as if they have half a dozen ASCII terminals each running the shell.

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 and write to the terminal, 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 UNIX 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 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 UNIX 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, UNIX 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 UNIX 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.

UNIX 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, various other people have written and distributed a variety of other shells.

10.2.4 UNIX Utility Programs

The user interface to UNIX consists not only of the shell, but also 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 UNIX systems. In addition to these standard utilities, there are many application programs as well, of course, such as Web browsers, image viewers, etc.

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 *cc*, 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 UNIX 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 one. All UNIX systems have these programs, and many more.

10.2.5 Kernel Structure

In Fig. 10-1 we saw the overall structure of a UNIX system. Now let us zoom in and look more closely at the kernel before examining the various parts. Showing the kernel structure is slightly tricky since there are many different versions of UNIX, but although the diagram of Fig. 10-3 describes 4.4BSD, it also applies to many other versions with perhaps small changes here and there.

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
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 UNIX utility programs required by POSIX.

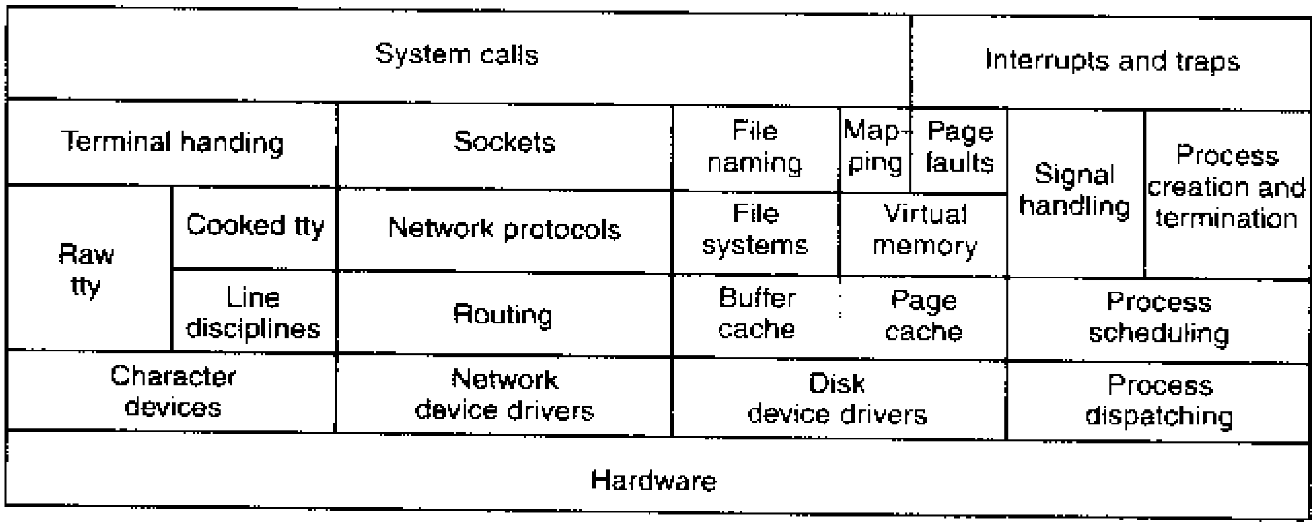


Figure 10-3. Structure of the 4.4BSD kernel.

The bottom layer of the kernel consists of the device drivers plus process dispatching. All UNIX drivers are classified as either character device drivers or block device drivers, with the main difference that seeks are allowed on block devices and not on character devices. Technically, network devices are character devices, but they are handled so differently that it is probably clearer to separate them, as has been done in the figure. Process dispatching occurs when an interrupt happens. The low-level code here stops the running process, saves its state in the kernel process table, and starts the appropriate driver. Process dispatching also happens when the kernel is finished and it is time to start up a user process again. Dispatching code is in assembler and is quite distinct from scheduling.

Above the bottom level, the code is different in each of the four "columns" of Fig. 10-3. At the left, we have the character devices. There are two ways they are used. 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 (*sh*), is line oriented and allows users to edit the current line before hitting ENTER to send it to the program. This software uses cooked mode and line disciplines.

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 the right packet goes to the right device or protocol handler. Most UNIX systems contain the full functionality of an Internet router within the kernel, although the performance is less than that of a hardware router, but this code predated modern hardware routers. Above the router code is the actual protocol stack, always including IP and TCP, but sometimes additional protocols as well. 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 are the file system's buffer cache and the page cache. In early UNIX systems, the buffer cache was a fixed chunk of memory, with the rest of memory for user pages. In many modern UNIX systems, there is no longer a fixed boundary, and any page of memory can be grabbed for either function, depending on what is needed more.

On top of the buffer cache come the file systems. Most UNIX systems support multiple file systems, including the Berkeley fast file system, log-structured file system, and various System V file systems. All of these file systems share the same buffer cache. On top of the file systems come file naming, directory management, hard link and symbolic link management, and other file system properties that are the same for all file systems.

On top of the page cache is the virtual memory system. All the paging logic is here, such as the page replacement algorithm. On top of it is the code for mapping files onto virtual memory and the high-level page fault management code. This is the code that figures out what to do when a page fault occurs. It first checks if the memory reference is valid, and if so, where the needed page is located and how it can be obtained.

The last column deals with process management. Above the dispatcher is the process scheduler, which chooses which process to run next. If threads are managed in the kernel, thread management is also here, although threads are managed in user space on some UNIX systems. Above the scheduler comes the code for processing signals and sending them to the correct destination, as well as the process creation and termination code.

The top layer is the interface into the system. On the left is the system call interface. All system calls come here and are directed to one of the lower modules, depending on the nature of the call. On right part of the top layer is the entrance for traps and interrupts, including signals, page faults, processor exceptions of all kinds, and I/O interrupts.

10.3 PROCESSES IN UNIX

In the previous sections, we started out by looking at UNIX as viewed from the keyboard, that is, what the user sees at the terminal. 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 UNIX 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, allocate memory, open files, and do I/O.

Unfortunately, with so many versions of UNIX 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 only active entities in a UNIX system are the processes. UNIX 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. Most versions of UNIX allow a process to create additional threads once it starts executing.

UNIX is a multiprogramming system, so multiple, independent processes may be running at the same time. 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 automatically 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 UNIX 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 UNIX because each one is a separate process, independent of all other processes.

Processes are created in UNIX 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 as well.

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.

Processes in UNIX 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 call-

```
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 UNIX.

ed **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 up, the system stops running *sort* until *head* has removed some data from the pipe.

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 a SIGFPE (floating-point exception) signal. The signals that are required by POSIX are listed in Fig. 10-5. Many UNIX systems have additional signals as well, but programs using them may not be portable to other versions of UNIX.

10.3.2 Process Management System Calls in UNIX

Let us now look at the UNIX system calls dealing with process management. The main ones are listed in Fig. 10-6. Fork is a good place to start the discussion. Fork is the only way to create a new process in UNIX systems. It creates an exact

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.

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 core image is copied to create the child, subsequent changes in one of them do not affect the other one. 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.

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.

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 name suggests.

```
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) {
          printf("Unable to fork0);           /* error condition */
          continue;                           /* repeat the loop */
     }

     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.

These will be described shortly. Various library procedures, including *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 5-character string "file1" and *argv*[2] would point to the 5-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 such as the terminal type and home directory name to a program. In Fig. 10-7, no environment is passed to the child, so 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 accidently 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 UNIX to suspend the process until the next signal arrives.

Thread Management System Calls

The first versions of UNIX did not have threads. That feature was added many years later. Initially there were many threads packages in use, but the proliferation of threads packages made writing portable code difficult. Eventually, the system calls used to manage threads were standardized as part of POSIX (P1003.1c).

The POSIX specification did not take a position on whether threads should be implemented in the kernel or in user space. The advantage of having user-space threads is that they can be implemented without having to change the kernel and thread switching is very efficient. The disadvantage of user-space threads is that if one thread blocks (e.g., on I/O, a semaphore, or a page fault), all the threads in the process block because the kernel thinks there is only one thread and does not schedule the process until the blocking thread is released. Thus the calls defined in P1003.1c were carefully chosen to be implementable either way. As long as user programs adhere carefully to the P1003.1c semantics, both implementations should work correctly. The most commonly-used thread calls are listed in Fig. 10-8. When kernel threads are used, these calls are true system calls; when user threads are used, these calls are implemented entirely in a user-space runtime library.

(For the truly alert reader, note that we have a typographical problem now. If the kernel manages threads, then calls such as "pthread_create," are system calls and following our convention should be set in Helvetica, like this: pthread_create. However, if they are simply user-space library calls, our convention for all procedure names is to use Times Italics, like this: *pthread_create*. Without prejudice, we will simply use Helvetica, also in the next chapter, in which it is never clear which Win32 API calls are really system calls. It could be worse: in the Algol 68 Report there was a period that changed the grammar of the language slightly when printed in the wrong font.)

Thread call	Description
pthread_create	Create a new thread in the caller's address space
pthread_exit	Terminate the calling thread
pthread_join	Wait for a thread to terminate
pthread_mutex_init	Create a new mutex
pthread_mutex_destroy	Destroy a mutex
pthread_mutex_lock	Lock a mutex
pthread_mutex_unlock	Unlock a mutex
pthread_cond_init	Create a condition variable
pthread_cond_destroy	Destroy a condition variable
pthread_cond_wait	Wait on a condition variable
pthread_cond_signal	Release one thread waiting on a condition variable

Figure 10-8. The principal POSIX thread calls.

Let us briefly examine the thread calls shown in Fig. 10-8. The first call, pthread_create, creates a new thread. It is called by

```
err = pthread_create(&tid, attr, function, arg);
```

This call creates a new thread in the current process running the code *function* with *arg* passed to it as a parameter. The new thread's ID is stored in memory at the location pointed to by the first parameters. The *attr* parameter can be used to specify certain attributes for the new thread, such as its scheduling priority. After successful completion, one more thread is running in the caller's address space than was before the call.

A thread that has done its job and wants to terminate calls pthread_exit. A thread can wait for another thread to exit by calling pthread_join. If the thread waited for has already exited, the pthread_join finishes immediately. Otherwise it blocks.

Threads can synchronize using locks called **mutexes**. Typically a mutex guards some resource, such as a buffer shared by two threads. To make sure that only one thread at a time accesses the shared resource, threads are expected to lock the mutex before touching the resource and unlock it when they are done. As long as all threads obey this protocol, race conditions can be avoided. Mutexes are like binary semaphores, that is, semaphores that can take on only the values of 0 and 1. The name "mutex" comes from the fact that mutexes are used to ensure mutual exclusion on some resource.

Mutexes can be created and destroyed by the calls pthread_mutex_init and pthread_mutex_destroy, respectively. A mutex can be in one of two states: locked or unlocked. When a thread needs to set a lock on an unlocked mutex (using pthread_mutex_lock), the lock is set and the thread continues. However, when a thread tries to lock a mutex that is already locked, it blocks. When the locking thread is finished with the shared resource, it is expected to unlock the corresponding mutex by calling pthread_mutex_unlock.

Mutexes are intended for short-term locking, such as protecting a shared variable. They are not intended for long-term synchronization, such as waiting for a tape drive to become free. For long-term synchronization, **condition variables** are provided. These are created and destroyed by calls to pthread_cond_init and pthread_cond_destroy, respectively.

A condition variable is used by having one thread wait on it, and another thread signal it. For example, having discovered that the tape drive it needs is busy, a thread would do pthread_cond_wait on a condition variable that all the threads have agreed to associate with the tape drive. When the thread using the tape drive is finally done with it (possibly even hours later), it uses pthread_cond_signal to release exactly one thread waiting on that condition variable (if any). If no thread is waiting, the signal is lost. In other words, condition variables do not count like semaphores. A few other operations are also defined on threads, mutexes, and condition variables.

10.3.3 Implementation of Processes in UNIX

A process in UNIX 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 kernel maintains two key data structures related to processes, the **process table** and the **user structure**. The process table is resident all the time and contains information needed for all processes, even those that are not currently present in memory. The user structure is swapped or paged out when its associated process is not in memory, in order not to waste memory on information that is not needed.

The information in the process table 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, if paging is used, to their 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. **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.

The user structure contains information that is not needed when the process is not physically in memory and runnable. 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 the process table, so they are 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 contained in the user structure includes the following items:

1. **Machine registers**. When a trap to the kernel occurs, the machine registers (including the floating-point ones, if used) are saved here.

2. **System call state**. Information about the current system call, including the parameters, and results.

3. **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.

4. **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.

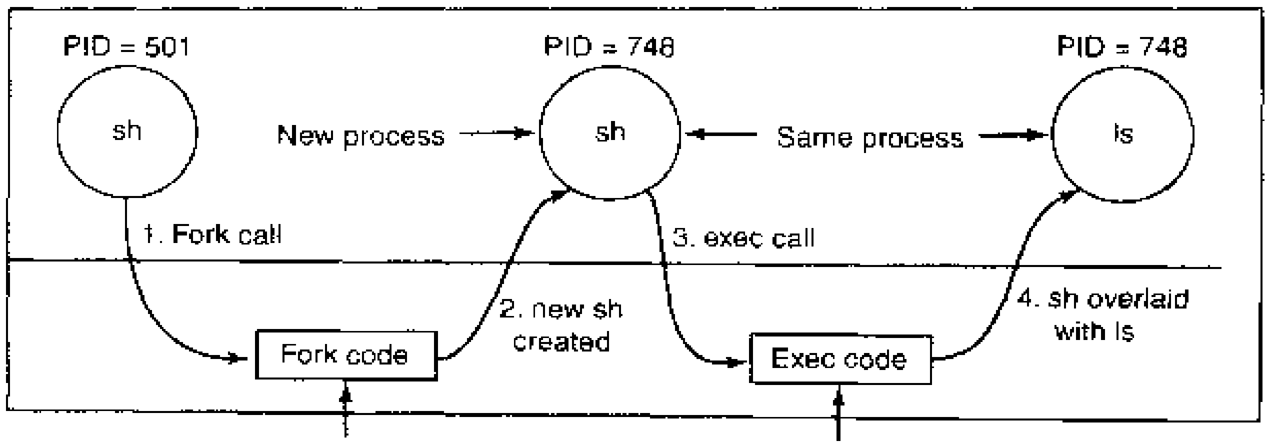
5. **Kernel stack**. A fixed stack for use by the kernel part of the process.

Bearing the use of these tables in mind, it is now easy to explain how processes are created in UNIX. When a fork system call is executed, the calling process traps to the kernel and looks for a free slot in the process table for use by the child. If it finds one, it copies all the information from the parent's process table entry to the child's entry. It then allocates memory for the child's data and stack segments, and makes exact copies of the parent's data and stack segments there. The user structure (which is often kept adjacent to the stack segment), is copied along with the stack. The text segment may either be copied or shared since it is read only. At this point, the child is ready to run.

When a user types a command, say, 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*. The steps involved are shown in Fig. 10-9.

The mechanism for creating a new process is actually fairly straightforward. A new process table slot 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.

In principle, a complete copy of the address space should be made, since the semantics of fork say that no memory is shared between parent and child. However, copying memory is expensive, so all modern UNIX 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



Allocate child's process table entry	Find the executable program
Fill child's entry from parent	Verify the execute permission
Allocate child's stack and user area	Read and verify the header
Fill child's user area from parent	Copy arguments, environ to kernel
Allocate PID for child	Free the old address space
Set up child to share parent's text	Allocate new address space
Copy page tables for data and stack	Copy arguments, environ to stack
Set up sharing of open files	Reset signals
Copy parent's registers to child	Initialize registers

Figure 10-9. The steps in executing the command *ls* typed to the shell.

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 System V, BSD, and most other UNIX 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. 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.

Threads in UNIX

The implementation of threads depends on whether they are supported in the kernel or not. If they are not, such as in 4BSD, the implementation is entirely in a user-space library. If they are, as in System V and Solaris, the kernel has some work to do. We discussed threads in a general way in Chap. 2. Here we will just make a few remarks about kernel threads in UNIX.

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.

Threads in Linux

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.

The heart of the Linux implementation of threads is a new system call, clone, that is not present in any other version of UNIX. 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 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 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. Five bits are defined, as listed in Fig. 10-10. Each bit controls some aspect of sharing, and each of the bits can be set independently of the other ones. 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 address space. Having its own address space means that the effect of its STORE instructions are 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.

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

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

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 UNIX.

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. Finally, *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.

This fine-grained sharing is possible because Linux maintains separate data structures for the various items listed at the start of Sec. 10.3.3 (scheduling parameters, memory image, etc.). The process table and user structure just point to these data structures, so it is easy to make a new process table entry for each cloned thread and have it either point 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 UNIX does not offer this functionality. A Linux program that takes advantage of it is then no longer portable to UNIX.

Scheduling in UNIX

Let us now examine the UNIX scheduling algorithm. Because UNIX has always been a multiprogramming system, its scheduling algorithm was designed from the beginning to provide good response to interactive processes. It is a two-level algorithm. The low-level algorithm picks the process to run next from the set of processes in memory and ready to run. The high-level algorithm moves processes between memory and disk so that all processes get a chance to be in memory and run.

Each version of UNIX has a slightly different low-level scheduling algorithm, but most of them are close to the generic one we will describe now. The low-level algorithm uses multiple queues. Each queue is associated with a range of nonoverlapping priority values. Processes executing in user mode (the top of the iceberg) have positive values. Processes executing in kernel mode (doing system calls) have negative values. Negative values have the highest priority and large positive values have the lowest, as illustrated in Fig. 10-11. Only processes that are in memory and ready to run are located on the queues, since the choice must be made from this set.

When the (low-level) scheduler runs, it searches the queues starting at the highest priority (i.e., most negative value) until it finds a queue that is occupied. The first process on that queue is then chosen and started. It is allowed to run for a maximum of one quantum, typically 100 msec, or until it blocks. If a process uses up its quantum, it is put back on the end of its queue, and the scheduling algorithm is run again. Thus processes within the same priority range share the CPU using a round-robin algorithm.

Once a second, each process' priority is recalculated according to a formula involving three components:

$$priority = CPU\_usage + nice + base$$

Based on its new priority, each process is attached to the appropriate queue of Fig. 10-11, usually by dividing the priority by a constant to get the queue number. Let us now briefly examine each of the three components of the priority formula.

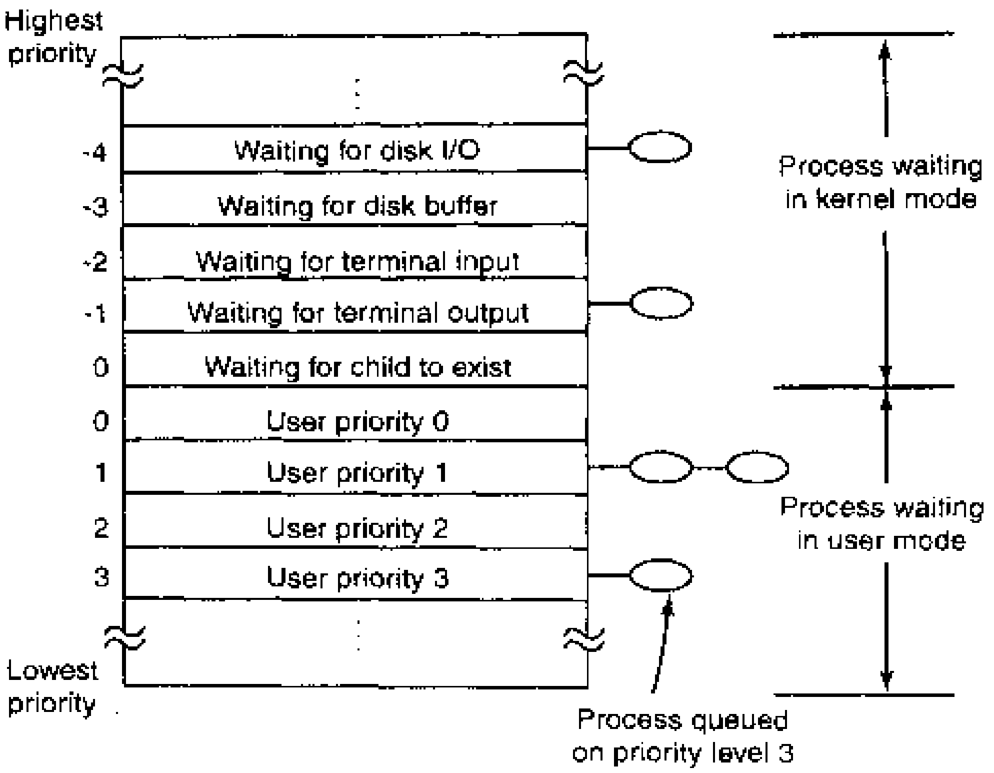

Highest priority
-4
Waiting for disk I/O
-3
Waiting for disk buffer
-2
Waiting for terminal input
-1
Waiting for terminal output
0
Waiting for child to exist
0
User priority 0
1
User priority 1
2
User priority 2
3
User priority 3
Lowest priority
Process waiting in kernel mode
Process waiting in user mode
Process queued on priority level 3

Figure 10-11. The UNIX scheduler is based on a multilevel queue structure.

CPU_usage, represents the average number of clock ticks per second that the process has had during the past few seconds. Every time the clock ticks, the CPU usage counter in the running process' process table entry is incremented by 1. This counter will ultimately be added to the process' priority giving it a higher numerical value and thus putting it on a lower-priority queue.

However, UNIX does not punish a process forever for using the CPU, so *CPU_usage* decays with time. Different versions of UNIX do the decay slightly differently. One way that has been used is to add the current value of *CPU_usage* to the number of ticks acquired in the past ΔT and divide the result by 2. This algorithm weights the most recent ΔT by ½, the one before that by ¼, and so on. This weighting algorithm is very fast because it just has one addition and one shift, but other weighting schemes have also been used.

Every process has a *nice* value associated with it. The default value is 0, but the allowed range is generally −20 to +20. A process can set *nice* to a value in the range 0 to 20 by the nice system call. 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.

When a process traps to the kernel to make a system call, it is entirely possible that the process has to block before completing the system call and returning to user mode. For example, it may have just done a waitpid system call and have to wait for one of its children to exit. It may also have to wait for terminal input or for disk I/O to complete, to mention only a few of the many possibilities. When it blocks, it is removed from the queue structure, since it is unable to run.

However, when the event it was waiting for occurs, it is put onto a queue with a negative value. The choice of queue is determined by the event it was waiting for. In Fig. 10-11, disk I/O is shown as having the highest priority, so a process that has just read or written a block from the disk will probably get the CPU within 100 msec. The relative priority of disk I/O, terminal I/O, etc. is hardwired into the operating system, and can only be modified by changing some constants in the source code and recompiling the system. These (negative) values are represented by *base* in the formula given above and are spaced far enough apart that processes being restarted for different reasons are clearly separated into different queues.

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 it can make the next one quickly. Similarly, if a process was blocked waiting for terminal 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 (i.e., those on the positive queues) basically get any service that is left over when all the I/O bound and interactive processes are blocked.

Scheduling in Linux

Scheduling is one of the few areas in which Linux uses a different algorithm from UNIX. We have just examined the UNIX scheduling algorithm, so we will now look at the Linux 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. Real-time round-robin threads are the same as real-time FIFO threads except that they 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.

Each thread has a scheduling priority. The default value is 20, but that can be altered using the nice(value) system call to a value of $20 - value$. Since *value* must be in the range -20 to $+19$, priorities always fall in the range: $1 \leq priority \leq 40$. The intention is that the quality of service is roughly proportional to the priority, with higher priority threads getting faster response time and a larger fraction of the CPU time than lower priority threads.

In addition to a priority, each thread has a quantum associated with it. The quantum is the number of clock ticks the thread may continue to run for. The clock runs at 100 Hz by default, so each tick is 10 msec, which is called a **jiffy**. The scheduler uses the priority and quantum as follows. It first computes the **goodness** of each ready thread by applying the following rules:

```
if (class == real_time) goodness = 1000 + priority;
if (class == timesharing && quantum > 0) goodness = quantum + priority;
if (class == timesharing && quantum == 0) goodness = 0;
```

Both real-time classes count for the first rule. All that marking a thread as real time does is make sure it gets a higher goodness than all timesharing threads. The algorithm has one little extra feature: if the process that ran last still has some quantum left, it gets a bonus point, so that it wins any ties. The idea here is that all things being equal, it is more efficient to run the previous process since its pages and cache blocks are likely to be loaded.

Given this background, the scheduling algorithm is very simple: when a scheduling decision is made, the thread with the highest goodness is selected. As the selected thread runs, at every clock tick, its quantum is decremented by 1. The CPU is taken away from a thread if any of these conditions occur:

1. Its quantum hits 0.
2. The thread blocks on I/O, a semaphore, or something else.
3. A previously blocked thread with a higher goodness becomes ready.

Since the quanta keep counting down, sooner or later every ready thread will grind its quantum into the ground and they will all be 0. However, I/O bound threads that are currently blocked may have some quantum left. At this point the scheduler resets the quantum of *all* threads, ready and blocked, using the rule:

```
quantum = (quantum/2) + priority
```

where the new quantum is in jiffies. A thread that is highly compute bound will usually exhaust its quantum quickly and have it 0 when quanta are reset, giving it a quantum equal to its priority. An I/O-bound thread may have considerable quantum left and thus get a larger quantum next time. If nice is not used, the priority will be 20, so the quantum becomes 20 jiffies or 200 msec. On the other hand, for a highly I/O bound thread, it may still have a quantum of 20 left when quanta are reset, so if its priority is 20, its new quantum becomes 20/2 + 20 = 30 jiffies. If another reset happens before it has spent 1 tick, next time it gets a quantum of 30/2 + 20 = 35 jiffies. The asymptotic value in jiffies is twice the priority. As a consequence of this algorithm, I/O-bound threads get larger quanta and thus higher goodness than compute-bound threads. This gives I/O-bound threads preference in scheduling.

Another property of this algorithm is that when compute-bound threads are competing for the CPU, ones with higher priority get a larger fraction of it. To see this, consider two compute-bound threads, *A*, with priority 20 and *B*, with priority 5. *A* goes first and 20 ticks later has used up its quantum. Then *B* gets to run for 5 quanta. At this point the quanta are reset. *A* gets 20 and *B* gets 5. This goes on forever, so *A* is getting 80% of the CPU and *B* is getting 20% of the CPU.

10.3.4 Booting UNIX

The exact details of how UNIX is booted vary from system to system. Below we will look briefly at how 4.4BSD is booted, but the ideas are somewhat similar for all versions. When the computer starts, the first sector of the boot disk (the master boot record) is read into memory 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 it does. Then it reads in the operating system kernel and jumps to it. At this point, *boot* 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 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 buffer 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.

Once the device list has been determined, the device drivers must be located. This is one area in which UNIX systems differ somewhat. In particular, 4.4BSD cannot load device drivers dynamically, so any I/O device whose driver was not statically linked with the kernel cannot be used. In contrast, some other versions of UNIX, such as Linux, can load drivers 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 execs 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 execs 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 illustrated in Fig. 10-12 for a system with three terminals.

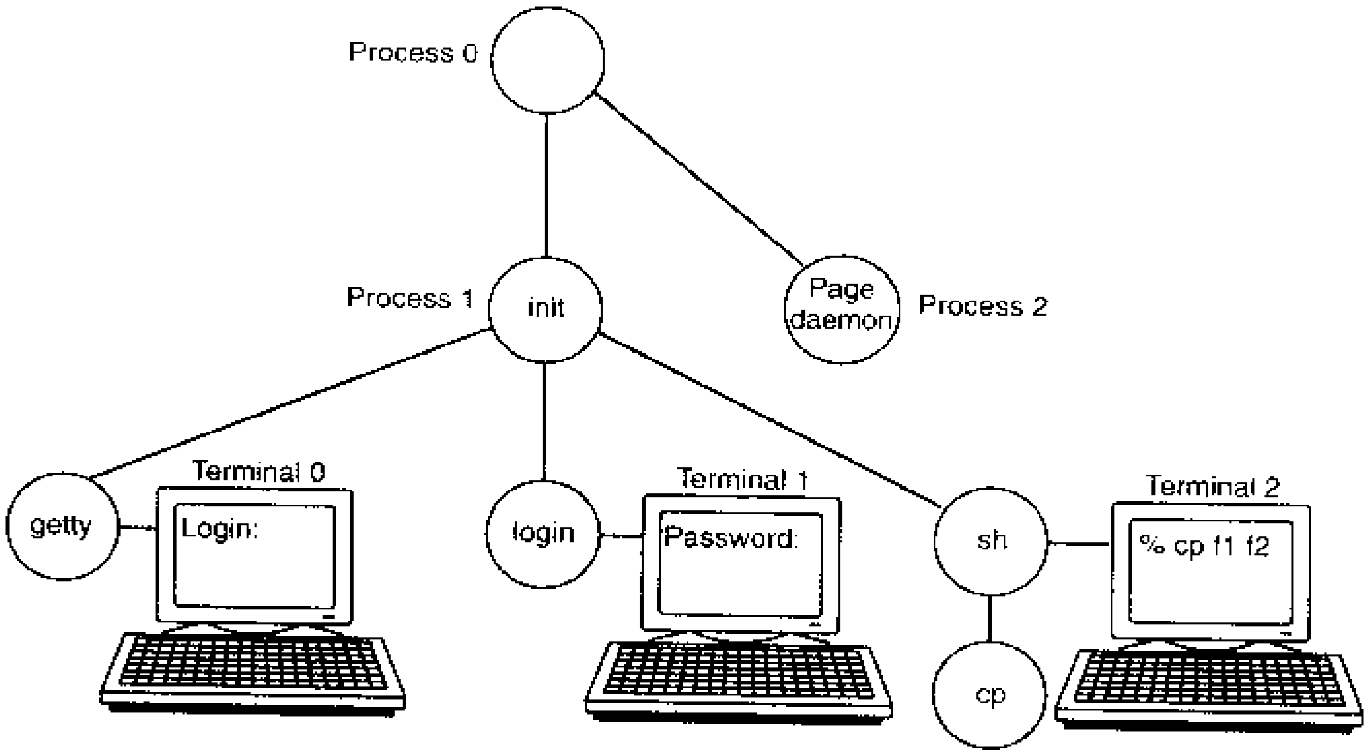


Figure 10-12. The sequence of processes used to boot some UNIX systems.

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 exec 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 UNIX

The UNIX memory model is straightforward, to make programs portable and to make it possible to implement UNIX 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 UNIX process has an address space consisting of three segments: text, data, and stack. An example process' address space is depicted in Fig. 10-13(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.

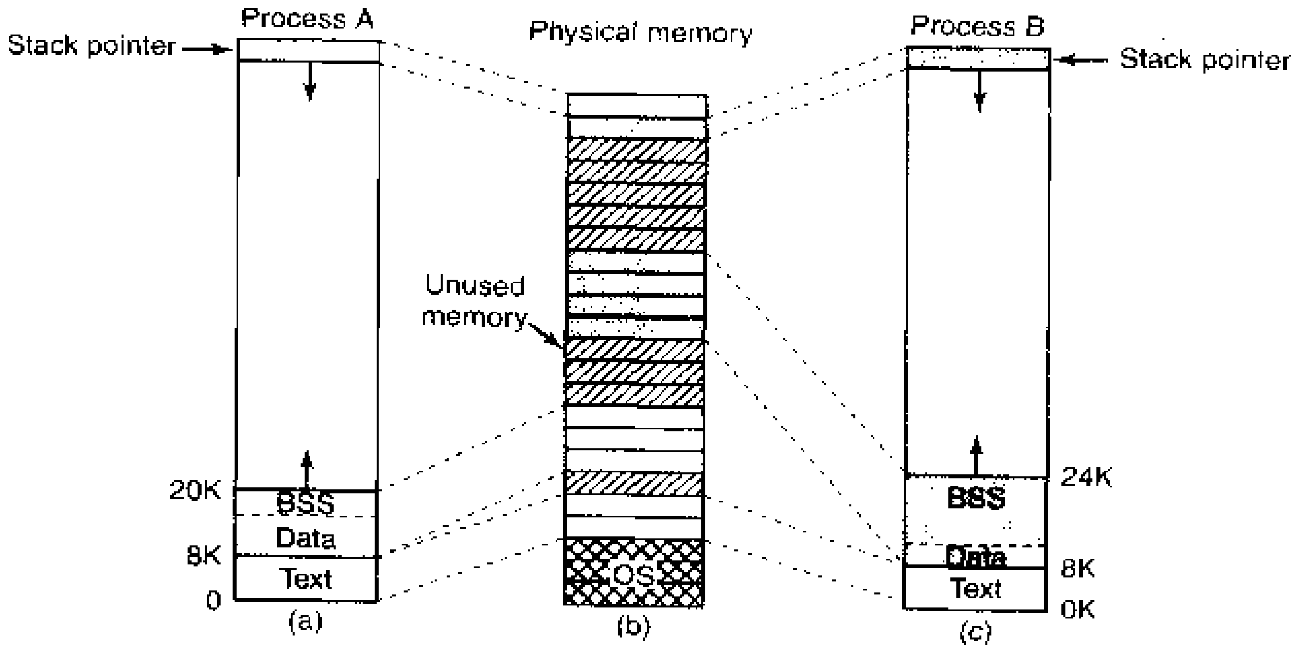


Figure 10-13. (a) Process *A*'s virtual address space. (b) Physical memory. (c) Process *B*'s virtual address space.

The **data segment** contains storage for 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**. The initialized part of the data segment contains variables and compiler constants that need an initial value when the program is started.

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, 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. 10-13(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.

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. UNIX 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 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. If the stack grows below the bottom of the stack segment, a hardware fault normally 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 UNIX systems support **shared text segments**. In Fig. 10-13(a) and Fig. 10-13(c) we see two processes, *A* and *B*, that have the same text segment. In Fig. 10-13(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, UNIX 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.

Many versions of UNIX support **memory-mapped files**. This feature makes it possible to map a file onto a portion of a process' address space so 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. 10-14 we see a file that is mapped into two processes at the same time, at different virtual addresses.

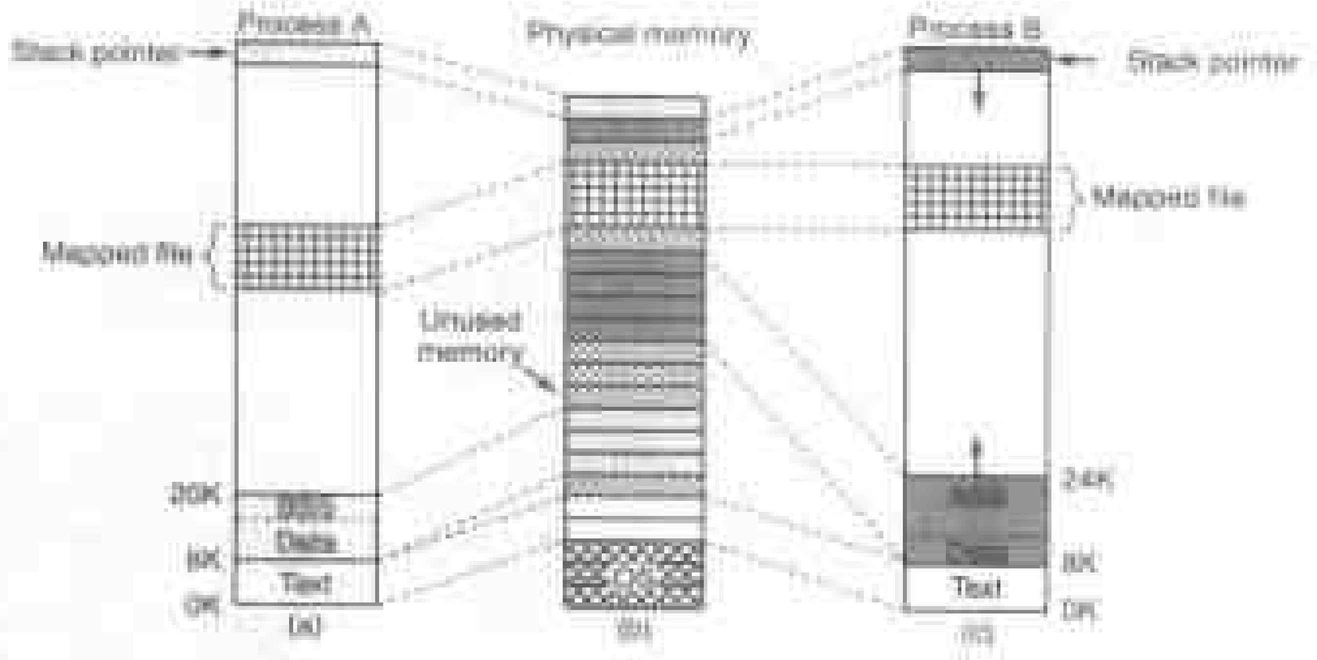


Figure 10-14. Two processes can share a mapped file.

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 them 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 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 UNIX

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 UNIX systems have system calls for managing memory. The most common ones are listed in Fig. 10-15. 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.

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 10-15. 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.

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 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 a mapped file. If only a portion of the file is unmapped, the rest remains mapped.

10.4.3 Implementation of Memory Management in UNIX

Prior to 3BSD, most UNIX systems were based on swapping, which worked as follows. When more processes existed than could be kept in memory, some of them were swapped out to disk. A swapped out process was always swapped out in its entirety (except possibly for shared text). A process was thus either in memory or on disk.

Swapping

Movement between memory and disk was handled by the upper level of the two-level scheduler, known as the **swapper**. Swapping from memory to disk was initiated when the kernel ran out of free memory on account of one of the following events:

1. A fork system call needed memory for a child process.
2. A brk system call needed to expand a data segment.
3. A stack became larger and ran out of the space allocated to it.

In addition, when it was time to bring in a process that had been on disk too long, it was frequently necessary to remove another process to make room for it.

To choose a victim to evict, the swapper first looked at the processes that were blocked waiting for something (e.g., terminal input). Better to remove a process that could not run than one that could. If one or more were found, the one whose priority plus residence time was the highest was chosen. Thus a process that had consumed a large amount of CPU time recently was a good candidate, as was one that had been in memory for a long time, even if it was mostly doing I/O. If no blocked process was available, then a ready process was chosen based on the same criteria.

Every few seconds, the swapper examined the list of processes currently swapped out to see if any of them were ready to run. If any were, the one that had been on disk longest was selected. Next, the swapper checked to see if this was going to be an easy swap or a hard one. An easy swap was one for which enough free memory currently existed, so that no process had to be removed to make room for the new one. A hard swap required removing one or more processes. An easy swap was implemented by just bringing in the process. A hard one was implemented by first freeing up enough memory by swapping out one or more processes, then bringing in the desired process.

This algorithm was then repeated until one of two conditions was met: (1) no processes on disk were ready to run, or (2) memory was so full of processes that had just been brought in that there was no room left for any more. To prevent thrashing, no process was ever swapped out until it had been in memory for 2 sec.

Free storage in memory and on the swap device was kept track of by linked lists of holes. When storage was needed on either one, the appropriate hole list was read using the first fit algorithm, which returned the first sufficiently large hole it could find. The hole was then reduced in size, leaving only the residual part not needed. list.

Paging in UNIX

All versions of UNIX for the PDP-11 and Interdata machines, as well as the initial VAX implementation, were based on swapping, as just described. Starting with 3BSD, however, Berkeley added paging in order to handle the ever-larger programs that were being written. Virtually all UNIX systems now have demand paging, tracing their origins to 3BSD. Below we will describe the 4BSD design, but the System V one is closely based on 4BSD and is almost the same.

The basic idea behind paging in 4BSD 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.

Berkeley UNIX does not use the working set model or any other form of prepaging because doing so requires knowing which pages are in use and which are not. Because the VAX did not have page reference bits, this information was not easily available (although it can be obtained in software at the cost of substantial additional overhead).

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 swapper, and process 1 is init, as shown in Fig. 10-12). Like all daemons, the page daemon is started up periodically so it can look around to see if there is any work for it to do. If it discovers that the number of pages on the list of free memory pages is too low, it initiates action to free up more pages.

Main memory in 4BSD is organized as shown in Fig. 10-16. It consists of three parts. The first two parts, the kernel and core 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 **core map** contains information about the contents of the page frames. Core map entry 0 describes page frame 0, core map entry 1 describes page frame 1, and so forth. With 1-KB page frames and 16-byte core map entries, less than 2 percent of the memory is taken up by the core map. The first two items in the

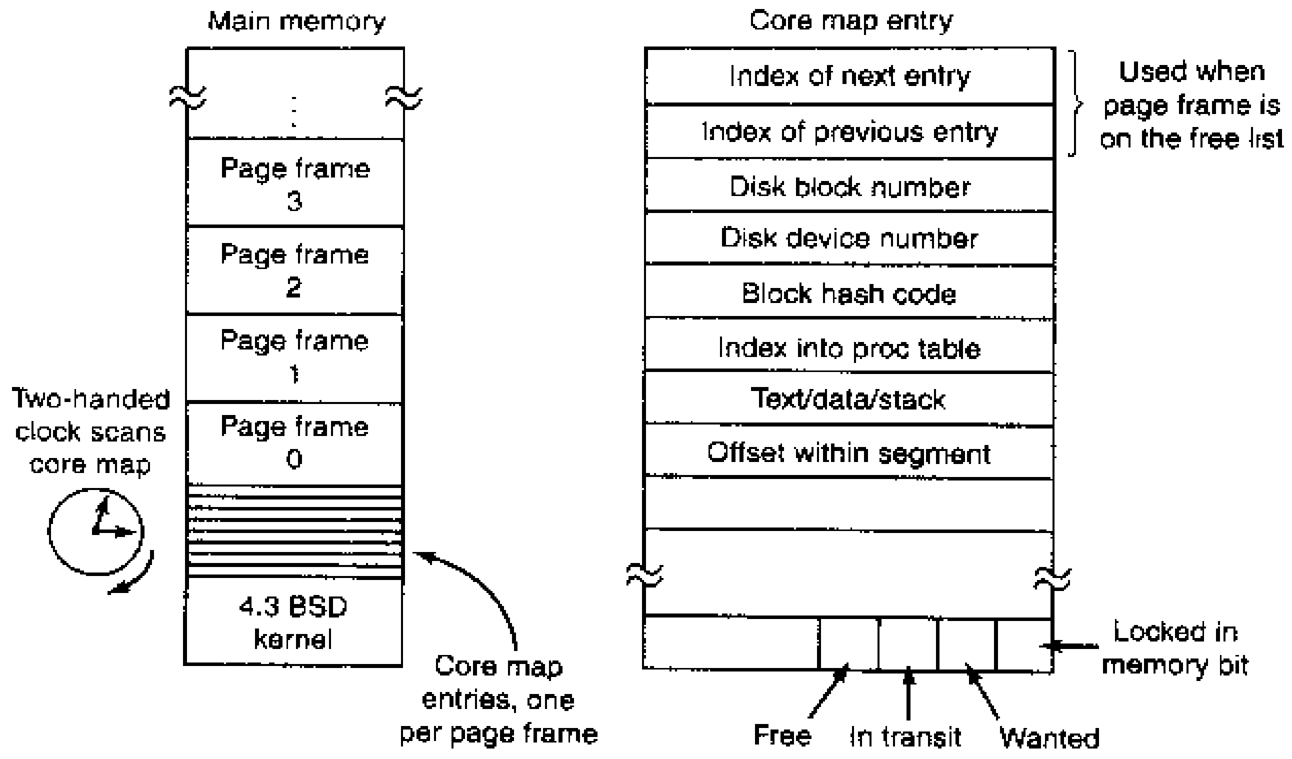


Figure 10-16. The core map in 4BSD.

core map entry shown in Fig. 10-16 are used only when the corresponding page frame is on the free list. Then they are used to hold a doubly linked list stringing together all the free page frames. The next three entries are used when the page frame contains information. Each page in memory also has a location on some disk where it is put when it is paged out. These entries are used to find the disk location where the page is stored. The next three entries give the process table entry for the page's process, which segment it is in, and where it is located in that segment. The last one shows some of the flags needed by the paging algorithm.

When a process is started, it may get a page fault because one or more of its pages are not present in memory. If a page fault occurs, the operating system takes the first page frame on the free list, removes it from the list, and reads the needed page into it. If the free list is empty, the process is suspended until the page daemon has freed a page frame.

The Page Replacement Algorithm

The page replacement algorithm is executed by the page daemon. Every 250 msec it is awakened to see if the number of free page frames is at least equal to a system parameter called *lotsfree* (typically set to 1/4 of memory). If insufficient page frames are free, the page daemon starts transferring pages from memory to disk until *lotsfree* page frames are available. If the page daemon discovers that more than *lotsfree* page frames are on the free list, it knows that it has nothing to do so it just goes back to sleep. If the machine has plenty of memory and few active processes, the page daemon sleep nearly all the time.

The page daemon uses a modified version of the clock algorithm. It is a global algorithm, meaning that when removing a page, it does not take into account whose page is being removed. Thus the number of pages each process has assigned to it varies in time.

The basic clock algorithm works by scanning the page frames circularly (as though they lay around the circumference of a clock). On the first pass, when the hand points to a page frame, the usage bit is cleared. On the second pass, any page frame that has not been accessed since the first pass will have its usage bit still cleared, and will be put on the free list (after writing it to disk, if it is dirty). A page frame on the free list retains its contents, which can be recovered if that page is needed before it is overwritten.

On a machine like the VAX that had no usage bits, when the clock hand pointed to a page frame on the first pass, the software usage bit was cleared and the page marked as invalid in the page table. When the page was next accessed, a page fault occurred, which allowed the operating system to set the software usage bit. The effect was the same as having a hardware usage bit, but the implementation was much more complicated and slower. Thus the software paid the price for poor hardware design.

Originally, Berkeley UNIX used the basic clock algorithm, but it was discovered that with large memories, the passes took too long. The algorithm was then modified to the **two-handed clock algorithm**, symbolized at the left of Fig. 10-16. With this algorithm, the page daemon maintains two pointers into the core map. When it runs, it first clears the usage bit at the front end, and then checks the usage bit at the back hand, after which it advances both hands. If the two hands are kept close together, then only very heavily used pages have much of a chance of being accessed between the time the front hand passes by and the time the back one does. If the two hands are 359 degrees apart (meaning the back hand is just ahead of the front hand), we are back to the original clock algorithm. Each time the page daemon runs, the hands rotate less than a full revolution, the amount depending on how far they have to go to get the number of pages on the free list up to *lotsfree*.

If the system notices that the paging rate is too high and the number of free pages is always way below *lotsfree*, the swapper is started to remove one or more processes from memory so that they will no longer compete for page frames. The 4BSD swap out algorithm is as follows. First the swapper looks to see if any processes have been idle for 20 sec or more. If any exist, the one that has been idle the longest is swapped out. If none exist, the four largest processes are examined and the one that has been in memory the longest is swapped out. If need be, the algorithm is repeated until enough memory has been recovered.

Every few seconds the swapper checks to see if any ready processes on the disk should be brought in. Each process on the disk is assigned a value that is a function of how long it has been swapped out, its size, the value it set using nice (if any), and how long it was sleeping before being swapped out. The function is weighted to usually bring in the process that has been out the longest, unless it is extremely large. The theory is that bringing in large processes is expensive, so they should not be moved too often. Swap-in only occurs if there are enough free pages, so that when the inevitable page faults start occurring, there will be page frames for them. Only the user structure and the page tables are actually brought in by the swapper. The text, data, and stack pages are paged in as they are used.

Each segment of each active process has a place on disk where it resides when it is paged or swapped out. Data and stack segments go to a scratch device, but program text is paged in from the executable binary file itself. No scratch copy is used for program text.

Paging in System V is fundamentally similar to that in 4BSD, which is not entirely surprising since the Berkeley version had been stable and running for years before paging was added to System V. Nevertheless, there are two interesting differences.

First, System V uses the original one-handed clock algorithm, instead of the two-handed one. Furthermore, instead of putting an unused page on the free list on the second pass, a page is only put there if it is unused for *n* consecutive passes. While this decision does not free pages as quickly as the Berkeley algorithm, it greatly increases the chance that a page once freed will not be needed again quickly.

Second, instead of a single variable *lotsfree*, System V has two variables, *min* and *max*. Whenever the number of free page frames falls below *min*, the page daemon is started to free up more pages. The daemon continues to run until there are *max* free page frames. This approach eliminates a potential instability in 4BSD. Consider a situation in which the number of free page frames is one less than *lotsfree*, so the page daemon runs to free one page and bring it up to *lotsfree*. Then another page fault occurs, using up one page frame, and reducing the number of available frames below *lotsfree* again, so the daemon has to run again. By setting *max* substantially above *min*, whenever the page daemon runs, it builds up a sufficient inventory so it does not have to run again for a substantial time.

Memory Management in Linux

Each Linux process on a 32-bit machine 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 address space is created when the process is created and is overwritten on an exec system call.

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. 10-14). 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.

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 all the pages can be found. When the list gets too long (more than 32 entries), a tree is created to speed up searching. The *vm_area_struct* entry lists the area's properties. These 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.

Linux uses a three-level paging scheme. Although this scheme was put into the system for the Alpha, it is also used (in degenerate form) for all architectures. Each virtual address is broken up into four fields, as shown in Fig. 10-17. The directory field is used as an index into the global directory, of which there is a private one for each process. The value found is a pointer to one of the page middle tables, which is again indexed by a field from the virtual address. The selected entry 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 middle directory has only one entry, so the global directory entry effectively chooses the page table to use.

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 buffer cache used by the file system, the paging cache, and other purposes. The buffer cache holds file blocks that have recently been read or have been read in advance in expectation of being used in the near future. It is dynamic in size and competes for the same pool of pages as the user pages. 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.

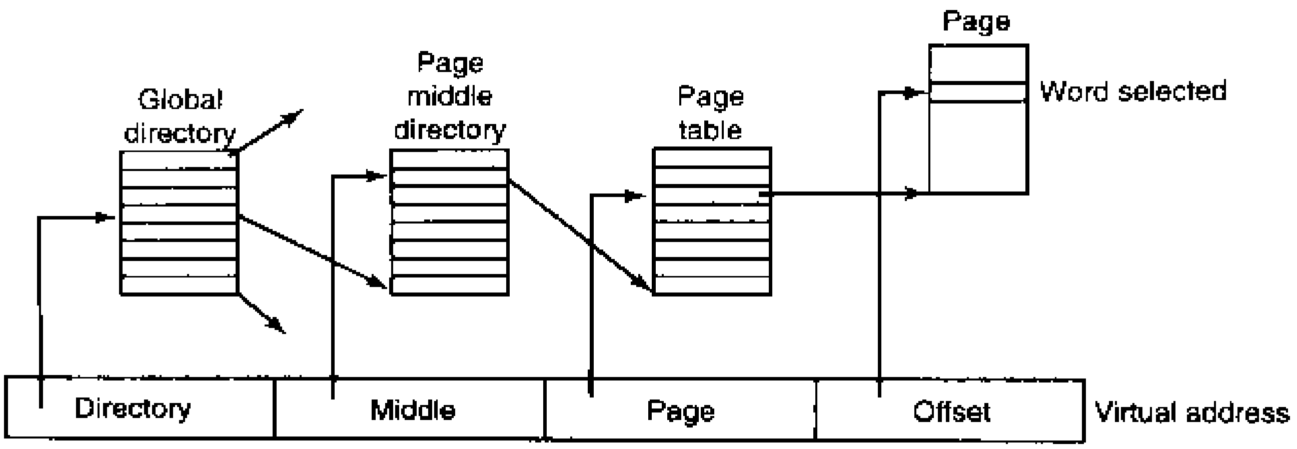


Figure 10-17. Linux uses three-level page tables.

In addition, Linux supports dynamically loaded modules, generally device drivers. These can be of arbitrary size and each one must be allocated a contiguous piece of kernel memory. As a 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.

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. 10-18(a). When a request for memory comes in, it is first rounded up to a power of two, say 8 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).

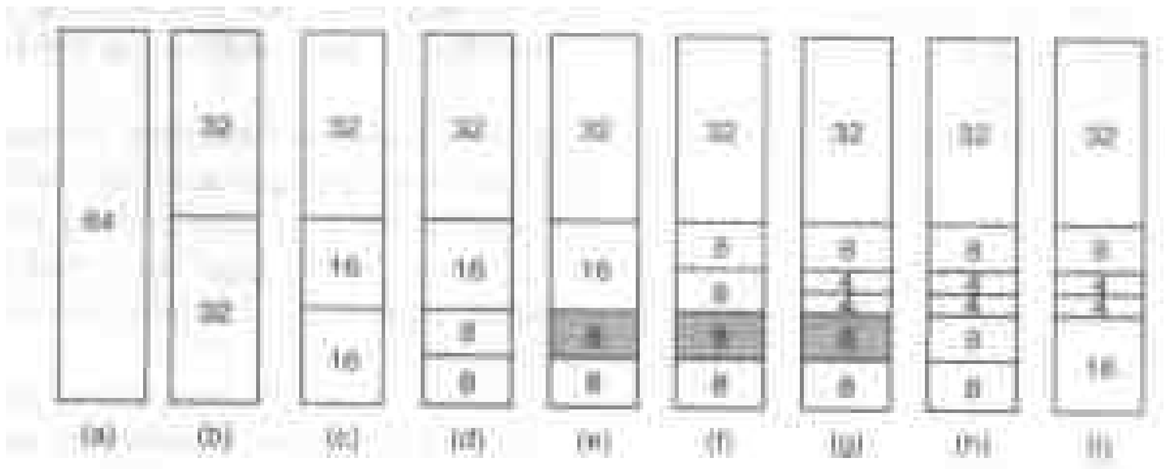


Figure 10-18. Operation of the buddy algorithm.

Now suppose that a second request comes in for 8 pages. This can be satisfied directly now (e). At this point a third request comes in for 4 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 8-page chunk is released.

Since the two adjacent 8-page chunks that were just freed are buddies, that is, originated 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, etc. 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 that takes chunks using the buddy algorithm but then carves slabs (smaller units) from them and manages the smaller units separately. A third memory allocator is also available when the requested memory need only be contiguous in virtual space, but not in physical memory. All these memory allocators are derived from those in System V.

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 hopes 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 up to 8 fixed-length paging files. 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 disk location is written into the page table.

Page replacement works as follows. Linux tries to keep some pages free so they can be claimed as needed. Of course, this pool must be continually replenished, so the real paging algorithm is how this happens. At boot time, *init* starts up a page daemon, *kswapd*, that runs once a second. It checks to see if there are enough free pages available. If so, it goes back to sleep for another second, although it can be awakened early if more pages are suddenly needed. The page daemon's code consists of a loop that it executes up to six times, with increasingly high urgency. Why six? Probably the author of the code thought four was not enough and eight was too many. Linux is sometimes like that.

The body of the loop makes calls to three procedures, each of which tries to reclaim different kinds of pages. 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. The effect of this algorithm is to first take the easy pages from each category before going after the hard ones. When enough pages have been reclaimed, the page daemon goes back to sleep.

The first procedure tries to reclaim pages from the paging cache and the file system buffer cache that have not been referenced recently, using a clock-like algorithm. The second procedure looks for shared pages that none of the users seems to be using much. The third procedure, which tries to reclaim ordinary user pages, is the most interesting, so let us take a quick look at it.

First, a loop is made over all the processes to see which one has the most pages currently in memory. Once that process is located, all of its *vm_area_structs* are scanned and all the pages are inspected in virtual address order starting from where we left off last time. If a page is invalid, absent from memory, shared, locked in memory, or being used for DMA, it is skipped. If the page has its reference bit on, the bit is turned off and the page is spared. If the reference bit is turned off, out goes the page, so this part of the algorithm is similar to clock (except that the pages are not scanned in FIFO order).

If the page is chosen for eviction and it is clean, it is abandoned immediately. If it is dirty and has a backing store page on disk, it is scheduled for a disk write. Finally, if it is dirty and has no backing store page, it goes into the paging cache, from which it might conceivably get a reprieve later if it is reused before it is actually paged out. The idea of scanning the pages in virtual address order is based on the hope that pages near each other in virtual address space will tend to be used or not used as a group, so they should be written to disk as a group and later brought in together as a group.

One other aspect of the memory management system that we have not yet mentioned is a second daemon, *bdflush*. It wakes up periodically (and in some cases is explicitly awakened) to check if too large a fraction of the pages are dirty. If so, it starts writing them back to disk.

10.5 INPUT/OUTPUT IN UNIX

The I/O system in UNIX is fairly straightforward. 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 UNIX 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 UNIX 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 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 a 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 email (which uses @ within email 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.

Rather than making a choice and forcing everyone to use it, UNIX 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.

Networking

Another example of I/O is networking, as pioneered by Berkeley UNIX, and summarized below. 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.

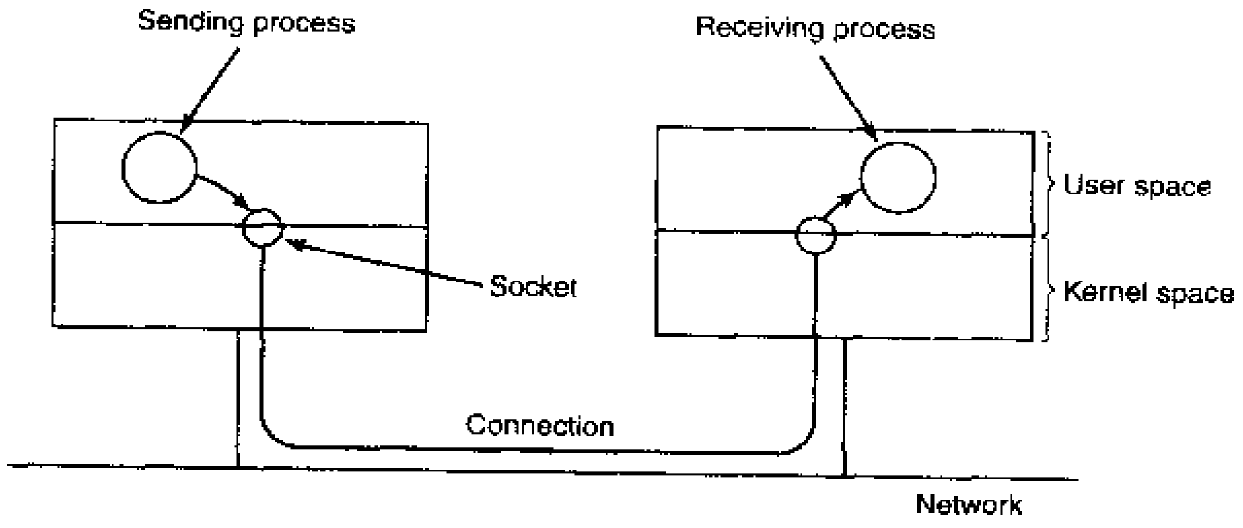


Figure 10-19. The uses of sockets for networking.

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 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 one 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.

10.5.2 Input/Output System Calls in UNIX

Each I/O device in a UNIX 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 the terminal. Whether each one is a separate system call or they share a single system call or something else is implementation dependent.

The first four 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. This asymmetry still persists, with some telephone companies offering inbound service at 1.5 Mbps and outbound service at 384 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 = cfgtetispeed(&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 still exists on most UNIX systems.

10.5.3 Implementation of Input/Output in UNIX

I/O in UNIX 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 structure arrays, *bdevsw* for block special files or *cdevsw* for character special files. 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 UNIX means adding a new entry to one of these tables and supplying the corresponding procedures to handle the various operations on the device.

The most important fields in the *cdevsw* array for a typical system might look as 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 *cdevsw* array to select the proper row (structure), then calls the function in the corresponding structure member (column) to have the work performed. Thus each row contains pointers to functions contained in one 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 fields of a typical *cdevsw* table.

Each driver is split into two parts. The top half runs in the context of the caller and interfaces to the rest of UNIX. 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 UNIX is covered in detail in (Egan and Teixeira, 1992).

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 that part of the system that does I/O on block special files (e.g., disks) is to minimize the number of actual transfers that must be done. To accomplish this goal, UNIX systems have a **buffer cache** between the disk drivers and the file system, as illustrated in Fig. 10-22. The buffer 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 buffer cache. If so, it is taken from there and a disk access is avoided. The buffer cache greatly improves system performance.

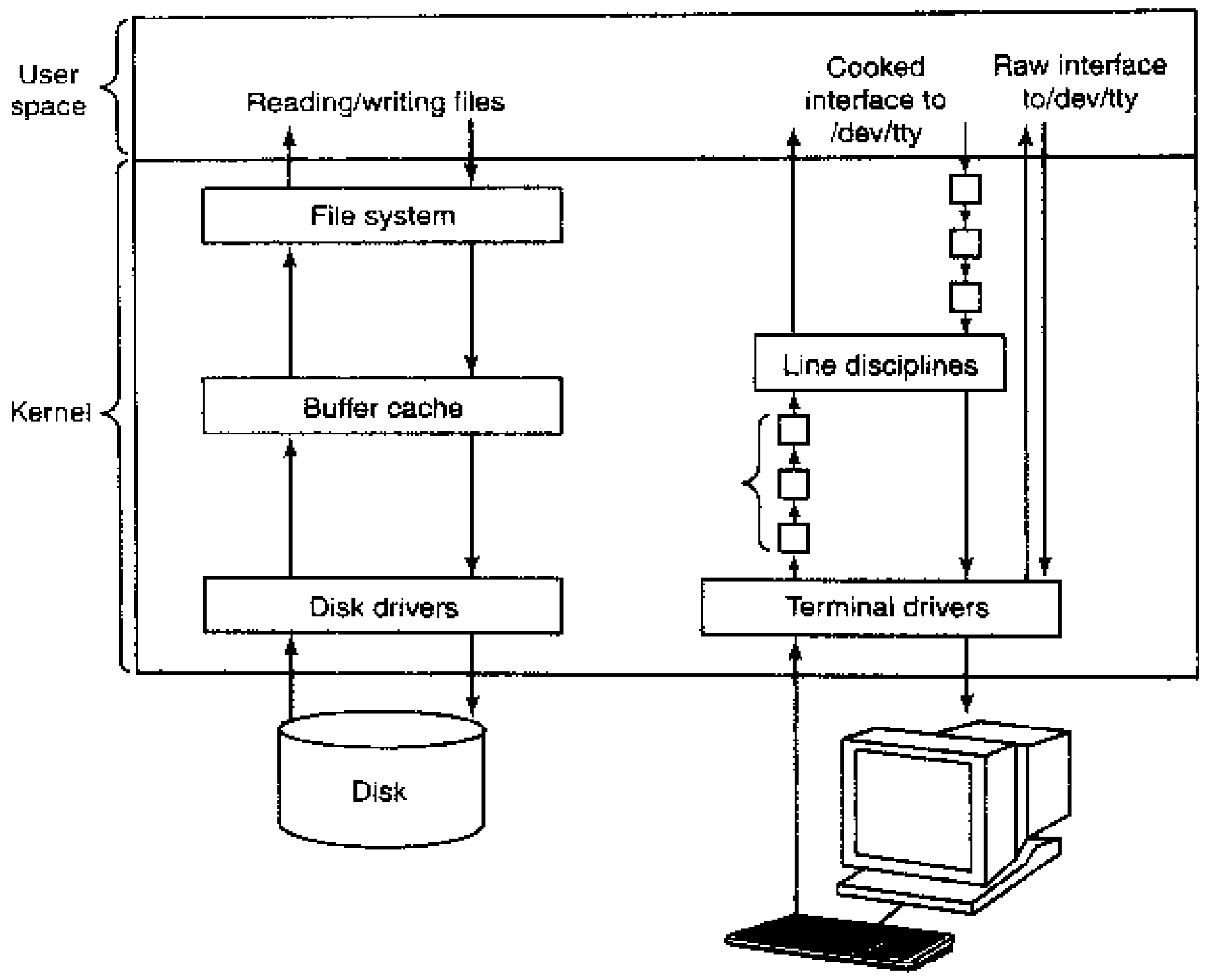


Figure 10-22. The UNIX I/O system in BSD.

If the block is not in the buffer cache, it is read from the disk into the buffer cache and from there, copied to where it is needed. Since the buffer cache has room for only a fixed number of blocks, some algorithm is needed to manage it. Usually, the blocks in the cache are linked together in a linked list. Whenever a block is accessed, it is moved to the head of the list. When a block must be removed from the cache to make room for a new block, the one at the rear of the chain is selected, since it is the least recently used block.

The buffer cache works for writes as well as for reads. When a program writes a block, it goes to the cache, not to the disk. Only when the cache fills up and the buffer must be reclaimed is the block forced out onto the disk. 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.

For decades, UNIX device drivers have been statically linked into the kernel so they were all present in memory when the system was booted every time. 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 it 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 *cdevsw* or *bdevsw*, 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. Some modern UNIX systems also support loadable modules now, too.

10.5.4 Streams

Since character special files deal with character streams and do not move blocks of information between memory and disk, they do not use the buffer cache. Instead, in the first versions of UNIX, each character device driver did all the work it needed for its device. However, in the course of time it became clear that many drivers were duplicating code present in other drivers such as code for buffering, flow control, and network protocols. Two different solutions were developed for structuring and modularizing the character drivers. We will now briefly examine each of them in turn.

The BSD solutions builds on data structures present in classical UNIX systems called **C-lists**, shown as small boxes in Fig. 10-22. Each one is a block of up to 64 characters, plus a count and a pointer to the next block. As characters arrive from terminals and other character devices, they are buffered in a chain of these blocks.

When a user process reads from */dev/tty* (e.g., standard input), the characters are not passed directly from the C-list to the process. Instead, they pass through a piece of kernel code called a **line discipline**. The line discipline acts like a filter, accepting the raw character stream from the terminal driver, processing it, and producing what is called a **cooked character stream**. In the cooked stream, local line editing has been done (i.e., erased characters and lines have been removed), carriage returns have been mapped onto line feeds, and other special processing has been completed. The cooked stream is passed to the process. However, if the process wants to interact on every character, it can put the line in raw mode, in which case the line discipline will be bypassed.

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 System V solution, **streams**, devised by Dennis Ritchie, is more general and is shown in Fig. 10-23. (System V also has a buffer cache for block special files, but since that is essentially the same as in BSD it is not shown here.) The basic idea of a stream is to be able to connect a user process to a driver and be able to insert processing modules into the stream dynamically, during execution. To some extent, a stream is the kernel analog of pipelines in user space.

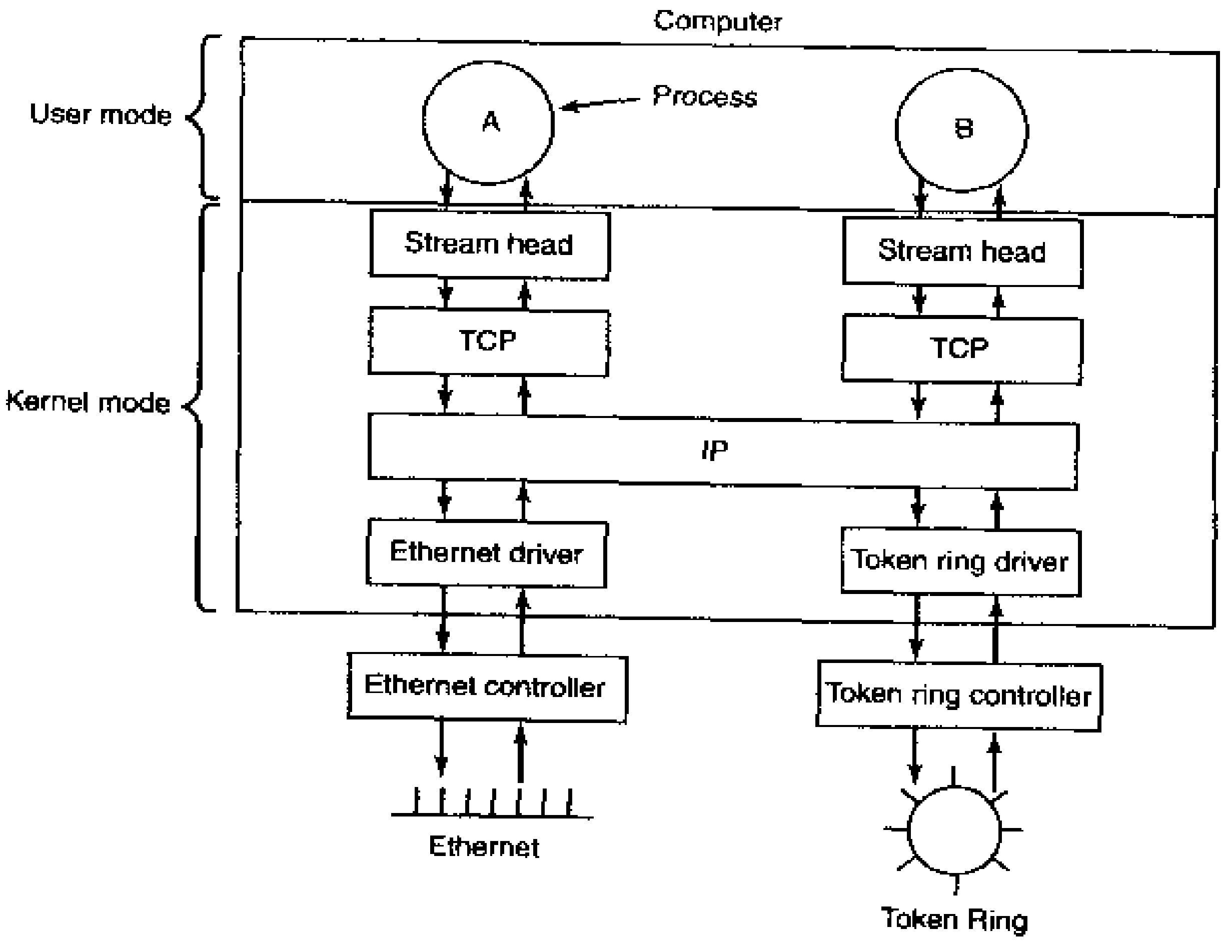


Figure 10-23. An example of streams in System V.

A stream always has a stream head on the top and connects to a driver on the bottom. As many modules as needed can be inserted into the stream. Processing can be bidirectional, so each module may need a section for reading (from the driver) and one for writing (to the driver). When a user process writes to a stream, code in the stream head interprets the system call and packs the data into stream buffers that are passed from module to module downward, with each module performing whatever transformations it is supposed to. Each module maintains a read queue and a write queue, so buffers are processed in the correct order. Modules have well-defined interfaces, defined by the streams infrastructure, so unrelated modules can be plugged together.

The example of Fig. 10-23 shows how streams are used when using the Internet TCP protocol over two different kinds of local area networks, Ethernet and Token ring. In particular, it illustrates another important feature of streams—multiplexing. A multiplexing module can take one stream and split it into multiple streams or can take multiple streams and merge them into one stream. The IP module in this configuration does both.

10.6 THE UNIX FILE SYSTEM

The most visible part of any operating system, including UNIX, is the file system. In the following sections we will examine the basic ideas behind the UNIX 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. The UNIX 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, UNIX nevertheless provides a powerful and elegant file system.

10.6.1 Fundamental Concepts

A UNIX 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 were originally restricted to 14 arbitrary characters, but Berkeley UNIX increased the limit to 255 characters, and this has been adopted by System V and most other versions as well. 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.Z*, which is probably a 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. 10-24.

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

Figure 10-24. Some important directories found in most UNIX systems.

There are two ways to specify file names in UNIX, 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/mos2/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 *mos2* which contains the file *chap-10*.

Absolute path names are often long and inconvenient. For this reason, UNIX 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/mos2* is the working directory, then the shell command

```
cp chap-10 backup1
```

has exactly the same effect as the longer command

```
cp /usr/ast/books/mos2/chap-10 /usr/ast/books/mos2/backup1
```

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. UNIX 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. 10-25(a). Fred and Lisa are working together on a project, and each one needs frequent 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. 10-25(b), after which he can use *x* to mean */usr/lisa/x*.

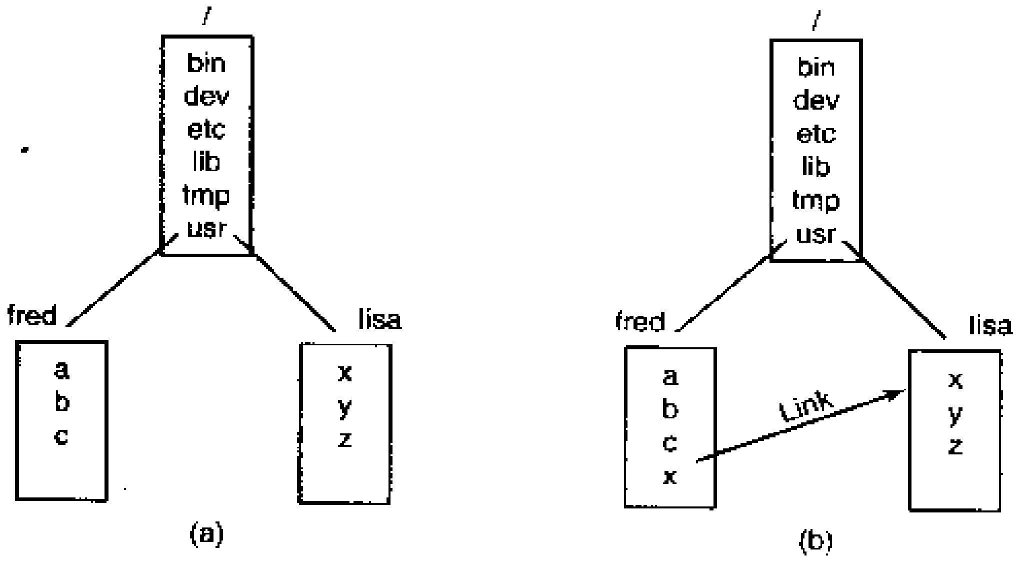


Figure 10-25. (a) Before linking. (b) After linking.

In the example just discussed, we suggested that before linking, the only way for Fred to refer to Lisa's file *x* was using its absolute path. Actually, this is not really true. When a directory is created, two entries, . and .., are automatically 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, UNIX 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 a diskette 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. 10-26(a). Here we have a hard disk, which we will call *C*:, and a diskette, which we will call *A*:. 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  A:/x  /a/d/x
```

This is the approach taken by systems like MS-DOS, Windows 98, and VMS.

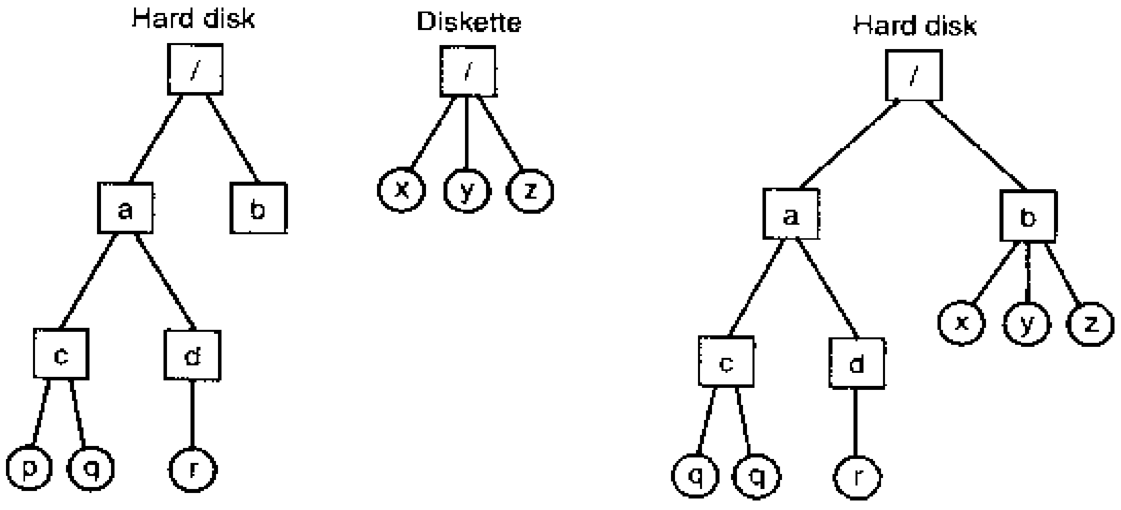


Figure 10-26. (a) Separate file systems. (b) After mounting.

The UNIX solution is to allow one disk to be mounted in another disk's file tree. In our example, we could mount the diskette on the directory */b*, yielding the file system of Fig. 10-26(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 UNIX 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.

Locked regions may overlap. In Fig. 10-27(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. 10-27(b). Finally, *C* locks bytes 2 through 11. As long as all these locks are shared, they can co-exist. Now consider what happens if a process tries to acquire an exclusive lock to byte 9 of the file of Fig. 10-27(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 UNIX

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.

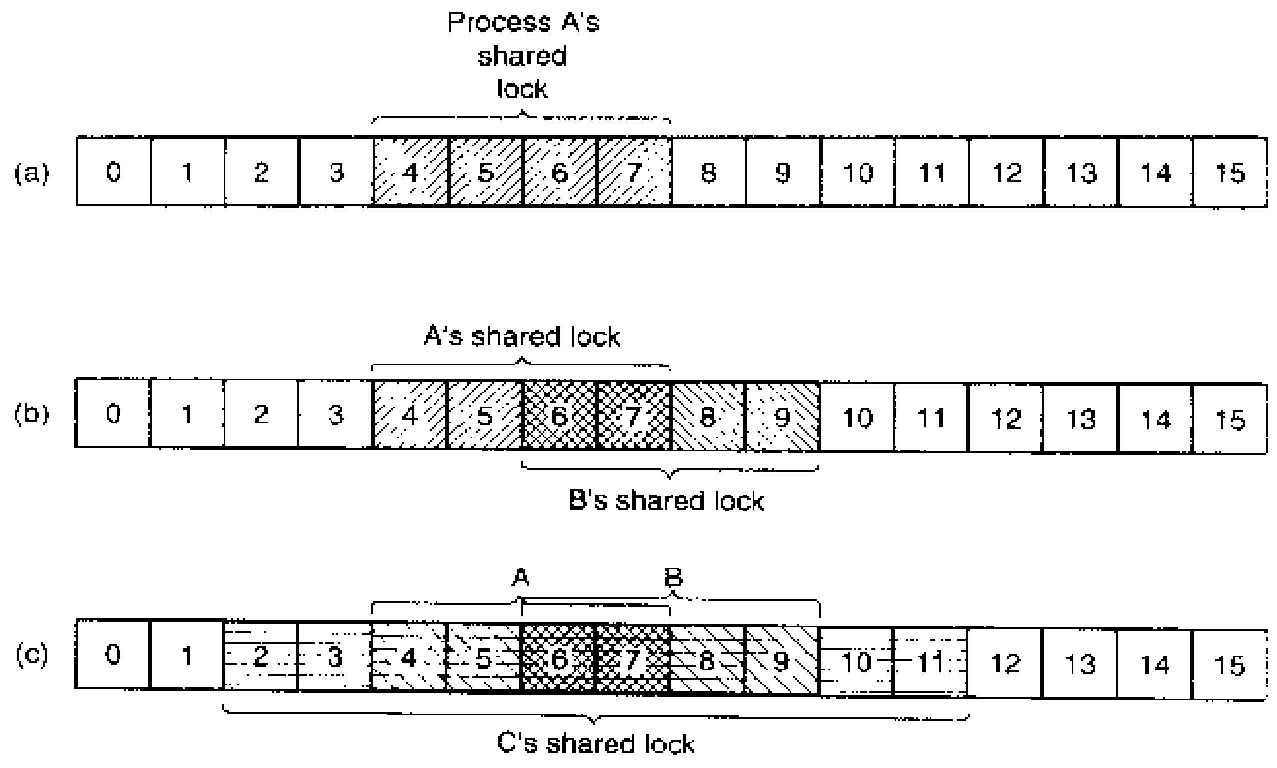


Figure 10-27. (a) A file with one lock. (b) Addition of a second lock. (c) A third lock.

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.

Now let us continue looking at the principal file system calls, which are listed in Fig. 10-28. 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

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 10-28. 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.

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 most 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 (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 one is the file descriptor for the file; the second one is a file position; the third one 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 was 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, UNIX 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 one is a pointer to a structure where the information requested is to be put. The fields in the structure are shown in Fig. 10-29. 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.

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 10-29. The fields returned by the stat system call.

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
```

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 where a simple concept (redirection) with a simple implementation (file descriptors 0 and 1) leads to a powerful tool (connecting programs in arbitrary ways without having to modify them at all).

The last system call in Fig. 10-28 is fcntl. It is used to lock and unlock files 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. 10-30. 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. 10-25, 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

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 10-30. 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.

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. 10-30 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 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 UNIX File System

In this section we will describe the implementation of the traditional UNIX file system. Afterward, we will discuss the Berkeley improvements. Other file systems are also in use. All UNIX systems can handle multiple disk partitions, each with a different file system on it.

A classical UNIX, disk partition contains a file system with the layout illustrated in Fig. 10-31. Block 0 is not used by UNIX and often contains code to boot the computer. Block 1 is the **superblock**. It contains critical 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). Destruction of the superblock will render the file system unreadable.

Following the superblock are the **i-nodes** (short for index-nodes, but never called that, although some lazy people drop the hyphen and call them **inodes**).

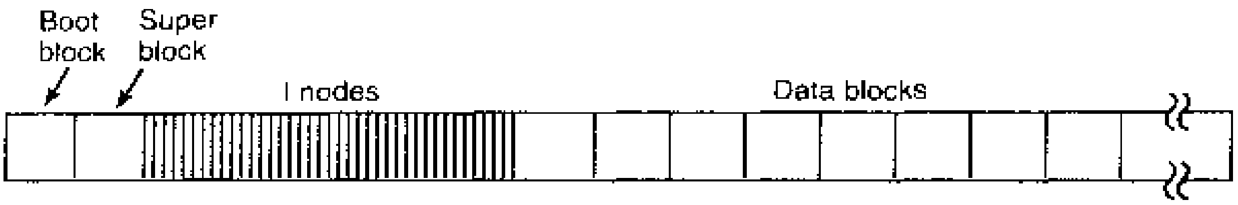


Figure 10-31. Disk layout in classical UNIX systems.

They are numbered from 1 up to some maximum. Each i-node is 64 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. It is this scatter that the Berkeley improvements were designed to reduce.

A directory in the traditional (i.e., V7) file system consists of an unsorted collection of 16-byte entries. Each entry contains a file name (up to 14 arbitrary characters), and the number of the file's i-node, as shown in Fig. 6-37. To open a file in the working directory, the system just reads the directory, comparing the name to be looked up to each entry until it either finds the name or concludes that it is not present.

If the file is present, the system extracts the i-node number, and uses this 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 varies somewhat from UNIX version to UNIX version. As a bare minimum, all the fields returned by the stat system call must be present to make stat work (see Fig. 10-29). In Fig. 10-32 we show the format used by all AT&T versions from Version 7 through System V.

Looking up an absolute path name such as */usr/ast/file* is slightly more complicated. First, the system locates the root directory, which generally uses i-node 2, especially when i-node 1 is reserved for bad block handling. Then it looks up the string "usr" in the root directory, to get the i-node number of the */usr* directory. 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.

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	39	Address of first 10 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 10-32. Structure of the i-node in System V.

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);
```

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 about 20).

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 UNIX. 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. 10-33. 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.

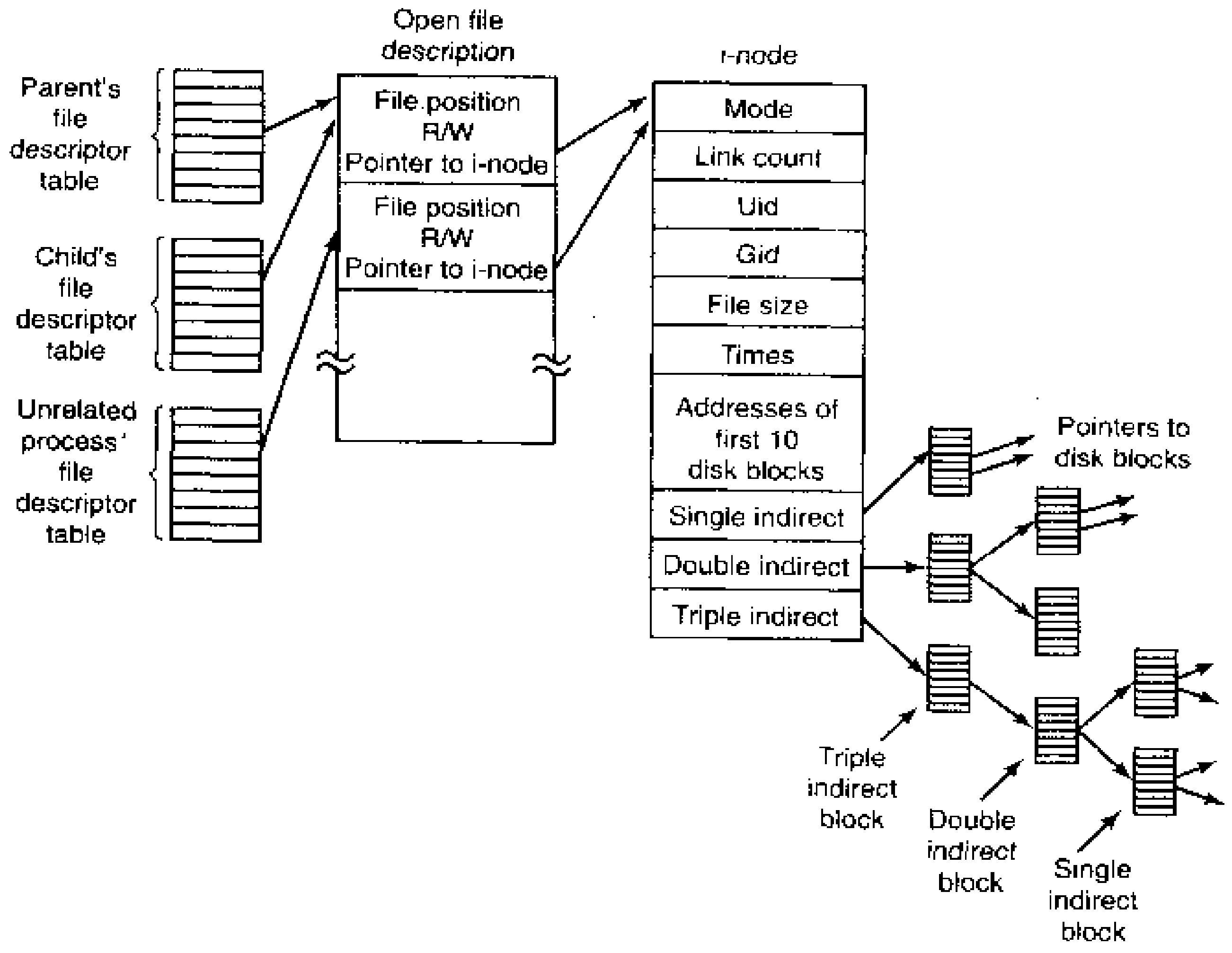


Figure 10-33. The relation between the file descriptor table, the open file description table, and the i-node table.

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 10 blocks of the file. If the file position falls in the first 10 blocks, the block is read and the data are copied to the user. For files longer than 10 blocks, a field in the i-node contains the disk address of a **single indirect block**, as shown in Fig. 10-33. 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 266 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.

The Berkeley Fast File System

The description above explains how the classical UNIX file system works. Let us now take a look at the improvements Berkeley made to it. First, directories were reorganized. Instead of limiting file names to 14 characters, the limit was set to 255 characters. Of course, changing the structure of all the directories meant that programs that naively read directories (which was and is still legal) and which expected a sequence of 16-byte entries, no longer worked. To provide portability across the two kinds of directories, Berkeley provided the system calls opendir, closedir, readdir, and rewinddir to allow programs to read directories without having to know their internal structure. Long file names and these system calls for reading directories were later added to all other versions of UNIX and to POSIX.

The BSD directory structure allows file names up to 255 characters and is illustrated in Fig. 10-34. 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.

Each directory entry in Fig. 10-34 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 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

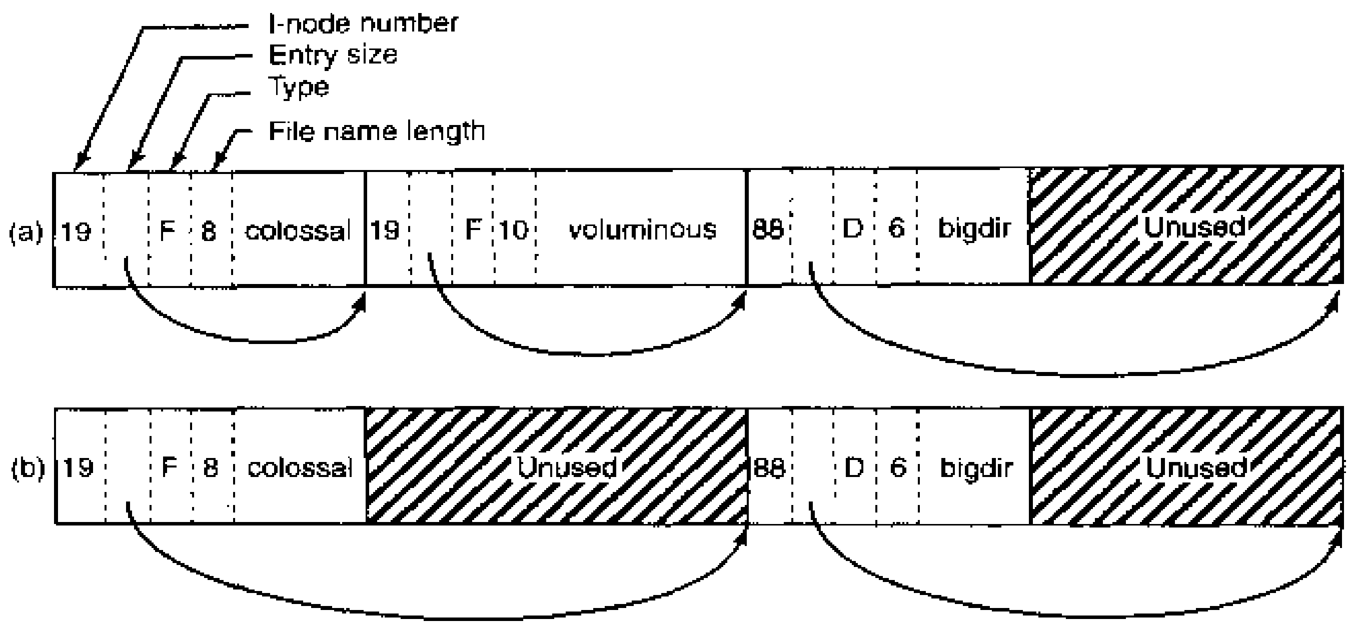


Figure 10-34. (a) A BSD directory with three files. (b) The same directory after the file *voluminous* has been removed.

padded by an unknown length. That is the meaning of the arrow in Fig. 10-34. Then comes the type field: file, directory, etc. 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. 10-34(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. To improve search performance, name caching was added to BSD. Before looking up a name, the system first checks the cache. If a cache hit occurs, the linear search can be bypassed.

The second major change Berkeley made to the file system was the division of the disk up into **cylinder groups**, each with its own superblock, i-nodes, and data blocks. The idea behind this change is to keep the i-node and data blocks of a file close together, to avoid long seeks. Whenever possible, blocks are allocated in the cylinder group containing the i-node.

The third change was the introduction of two block sizes instead of just one. For storing large files, it is more efficient to have a small number of large blocks rather than many small ones. On the other hand, many UNIX files are small, so having only large blocks would be wasteful of disk space. Having two sizes allows efficient transfers for large files and space efficiency for small ones. The price paid is considerable extra complexity in the code.

The Linux File System

The initial Linux file system was the MINIX 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, there was interest in better file systems almost from the beginning. 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 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 that has become the main file system. However, Linux still supports over a dozen file systems using the NFS file system (described in the next section). When Linux is linked, a choice is offered of which file systems should be built into the kernel. Other ones can be dynamically loaded as modules during execution, if need be.

Ext2 is very similar to the Berkeley Fast File system with some small differences. The Ext2 disk layout is shown in Fig. 10-35. Instead of using cylinder groups, which mean almost nothing with today's virtual disk geometries, it divides the disk into groups of blocks without regard to where the cylinder boundaries fall. Each block group starts with a superblock, which tells how many blocks and i-nodes there are, gives the block size, etc. 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.

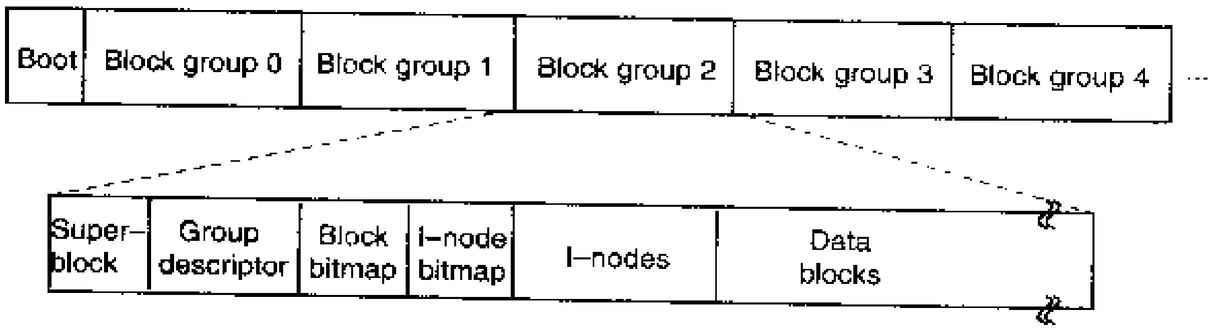


Figure 10-35. Layout of the Linux Ext2 file system.

Two bitmaps keep track of the free blocks and free i-nodes, respectively. 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 the latter is not in practice. Next come the i-nodes themselves. Each i-node is 128 bytes, twice the size of the standard UNIX i-nodes of Fig. 10-32. The extra space is used as follows. Instead of 10 direct block addresses and 3 indirect block addresses, Linux allows 12 direct and 3 indirect addresses. Furthermore, the addresses have been extended from 3 bytes to 4 bytes, to handle disk partitions larger than 2^{24} blocks (16 GB), which was already a problem in UNIX. In addition, fields are reserved for pointers to access control lists for a finer degree of protection, but these are still on the drawing board. The rest of the i-node is reserved for future use. History shows that unused bits do not remain that way for long.

Operation of the file system is similar to the Berkeley fast file system. One difference with Berkeley, however, is the use of standard 1 KB throughout. The Berkeley fast file systems uses 8-KB blocks and then fragments them into 1-KB pieces if need be. Ext2 just does it the easy way. Like the Berkeley system, when a file grows, Ext2 tries to put the block in the same block group as the rest of the file, preferably right after the previous last block. Also, when a new file is added to a directory, Ext2 tries to put it into the same block group as its directory. New directories are scattered uniformly across the disk.

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 UNIX 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 later). In this section we will examine Sun Microsystem's **NFS (Network File System)**, which is used on all modern UNIX systems (and some non-UNIX systems) to join the file systems on separate computers into one logical whole. Three aspects of NFS are of interest: the architecture, the protocol, and the implementation. We will now examine these in turn.

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. 10-36.

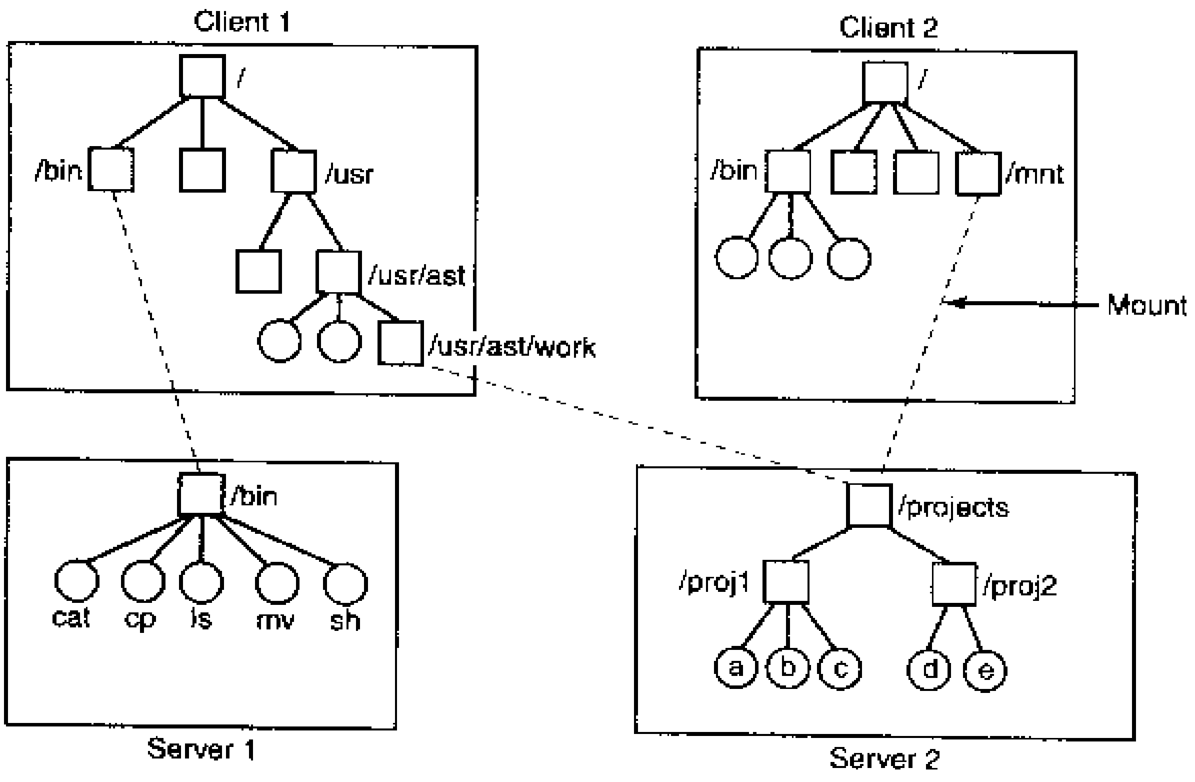


Figure 10-36. Examples of remote mounted file systems. Directories are shown as squares and files are shown as circles.

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.

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 UNIX 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 UNIX 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. Also, they can also access file attributes, such as file mode, size, and time of last modification. Most UNIX system calls are supported by NFS, with the perhaps surprising exception 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 UNIX file semantics. For example, in UNIX 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?—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 enabled, 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 UNIX systems use a three-layer implementation similar to that of Fig. 10-37. 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.

The task of the VFS layer is to maintain a table with one entry for each open file, analogous to the table of i-nodes for open files in UNIX. In ordinary UNIX, an i-node is indicated uniquely by a (device, i-node number) pair. Instead, the VFS layer has an entry, called a **v-node (virtual i-node)**, for every open file. V-

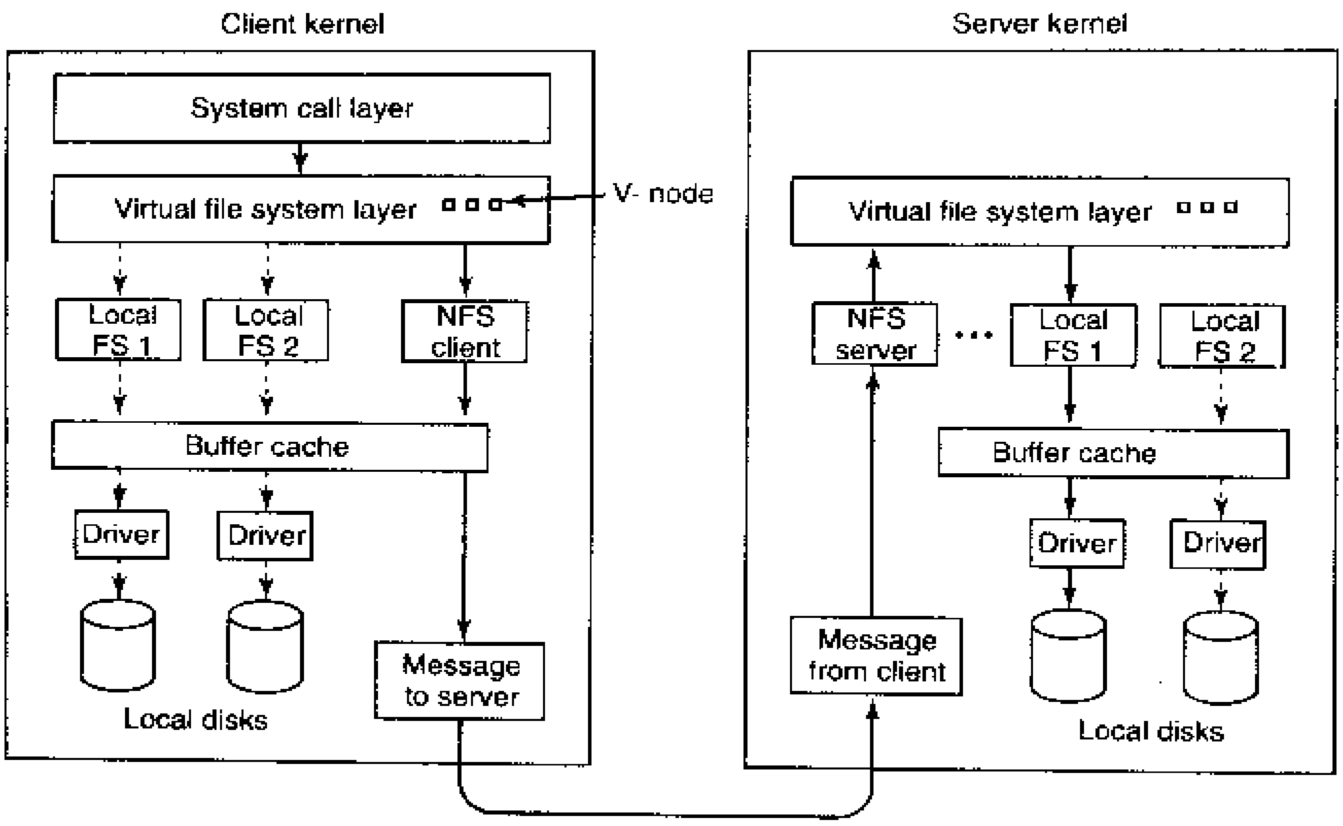


Figure 10-37. The NFS layer structure.

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 UNIX systems can support multiple file systems (e.g., V7, Berkeley FFS, ext2fs, /proc, FAT, etc.). Although VFS was invented to support NFS, most modern UNIX 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. 10-37 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. 10-37). 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.

10.7 SECURITY IN UNIX

Despite the name, 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 UNIX.

10.7.1 Fundamental Concepts

The user community for a UNIX 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. Originally, a user could be in only one group, but some versions of UNIX now allow a user to be in more than one group at the same time. For simplicity, we will not discuss this feature further.

The basic security mechanism in UNIX 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. 10-38.

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 10-38. Some example file protection modes.

The first two entries in Fig. 10-38 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 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 UNIX 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 UNIX

There are only a small number of system calls relating to security. The most important ones are listed in Fig. 10-39. 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 10-39. 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 UNIX

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 *sh*, 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.

Security in Linux is essentially the same as in UNIX. All the UNIX security features are implemented and there is little else in this area.

10.8 SUMMARY

UNIX began life as a minicomputer timesharing system, but is now used on machines ranging from notebook computers to supercomputers. Three interfaces to it exist: the shell, the C library, and the system calls themselves. 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 lean and mean, roughly 100 calls, each of which doing what is needed and no more.

The key concepts in UNIX 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 UNIX uses two key data structures, the process table and the user structure. The former is always in memory, but the latter can be swapped or paged out. Process creation is done by duplicating the process table entry, and then the memory image. 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 used to be done by swapping, but is now done by paging in most UNIX systems. The core map keeps track of the state of each page, and the page daemon uses a 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 buffer cache to reduce the number of disk accesses. An LRU algorithm is used to manage the cache. Character I/O can be done in raw or cooked mode. Line disciplines or streams are used to add features to character I/O.

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.

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

PROBLEMS

1. When the kernel catches system call, how does it know which system call it is supposed to carry out?

2. 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*?
   ```

3. What does the following UNIX shell pipeline do?

   ```
   grep nd xyz | wc –l
   ```

4. Write a UNIX pipeline that prints the eighth line of file *z* on standard output.

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

6. 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?

7. When the UNIX 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?

8. About how long does it take to fork off a child process under the following conditions: text size = 100 KB, data size = 20 KB, stack size = 10 KB, process table size = 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.

9. As megabyte programs became more common, the time spent executing the fork system call grew proportionally. Worse yet, nearly all of this time was wasted, since most programs call exec shortly after forking. To improve performance, Berkeley invented a new system call, vfork, in which the child shares the parent's address space, instead of getting its own copy of it. Describe a situation in which a poorly-behaved child can do something that makes the semantics of vfork fundamentally different from those of fork.

10. If a UNIX process has a *CPU_usage* value of 20, how many ΔT intervals does it take to decay to 0 if it does not get scheduled for a while? Use the simple decay algorithm given in the text.

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

12. To what hardware concept is a signal closely related? Give two examples of how signals are used.

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

14. 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?

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

16. 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.

17. In every process' entry in the process table, the PID of the process' parent is stored. Why?

18. 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?

19. The Linux scheduler computes the goodness for real-time processes by adding the priority to 1000. Could a different constant have been chosen and still have the algorithm make the same choices as the regular one?

20. When booting UNIX (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 the extra step needed? Surely it would be simpler to have the bootstrap loader in sector 0 just load the operating system directly.

21. 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?

22. In 4BSD, each core map entry contains an index for the next entry on the free list, which is used when the current entry is on the free list. This field is 16 bits. Pages are 1 KB. Do these sizes have any implications for the total amount of memory BSD can support? Explain your answer.

23. In BSD, 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?

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

25. 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?

26. 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?

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

28. 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, give an example of how this can happen. If not, show that it is impossible.

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

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

31. 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.

32. Consider the locked file of Fig. 10-27(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 kind of problems are introduced into the semantics by this situation. Propose and defend two solutions.

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

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

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

36. In Fig. 10-25, both Fred and Lisa have access to the file *x* in their respective directories after linking. Is this access completely symmetrical in the sense anything one of them can do with it the other one can too?

37. 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.

38. 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.

39. A UNIX i-node has 10 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?

40. 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?

41. Why does LRU work for managing the buffer cache, whereas it rarely works for keeping track of pages in a virtual memory system?

42. UNIX has a system call, sync, that flushes the buffer cache back to disk. At boot time, a program called *update* is started. Once every 30 sec it calls sync, then goes to sleep for 30 sec. Why does this program exist?

43. 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?

44. Make an educated guess as to which UNIX system call is the fastest.

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

46. 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.

47. 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?

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

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

50. 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.

51. Write a dumb terminal program to connect two UNIX or Linux workstations via the serial ports. Use the POSIX terminal management calls to configure the ports.