Appendix C

BSD UNIX

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

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

C.1 UNIX History

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

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

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

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

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

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

C.1.1 UNIX Support Group

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

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

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

C.1.2 Berkeley Begins Development

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

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

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

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

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

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

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

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

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

C.1.3 The Spread of UNIX

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

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

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

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

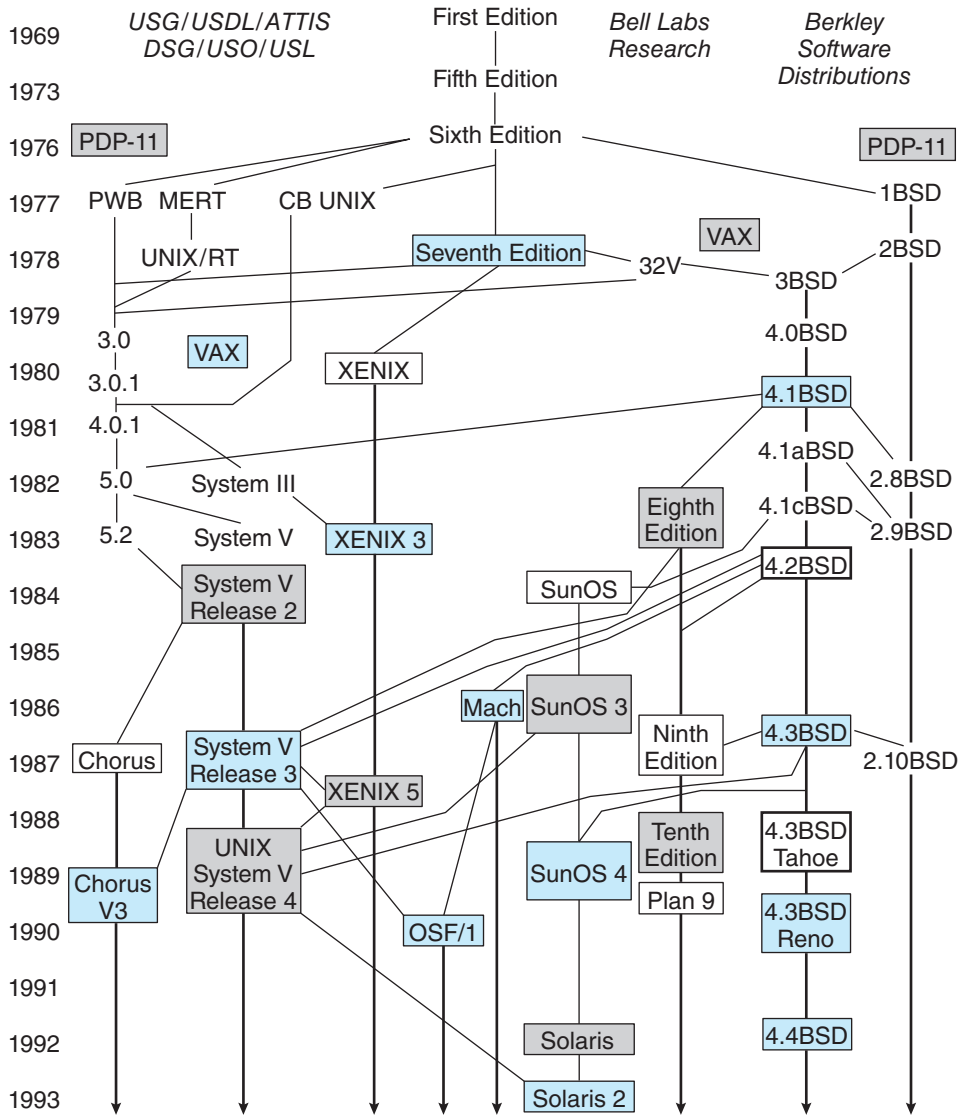

Figure C.1 History of UNIX versions up to 1993.

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

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

C.1.4 History of FreeBSD

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

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

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

C.2 Design Principles

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

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

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

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

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

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

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

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

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

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

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

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

C.3 Programmer Interface

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

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

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

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

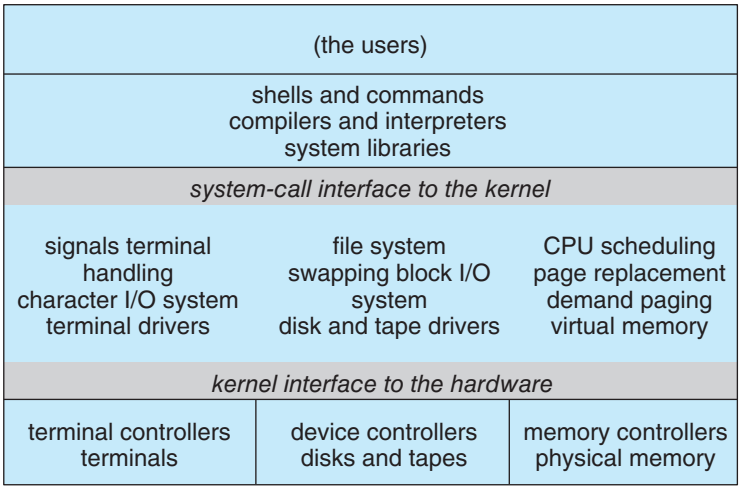

Figure C.2 4.4BSD layer structure.

C.3.1 File Manipulation

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

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

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

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

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

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

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

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

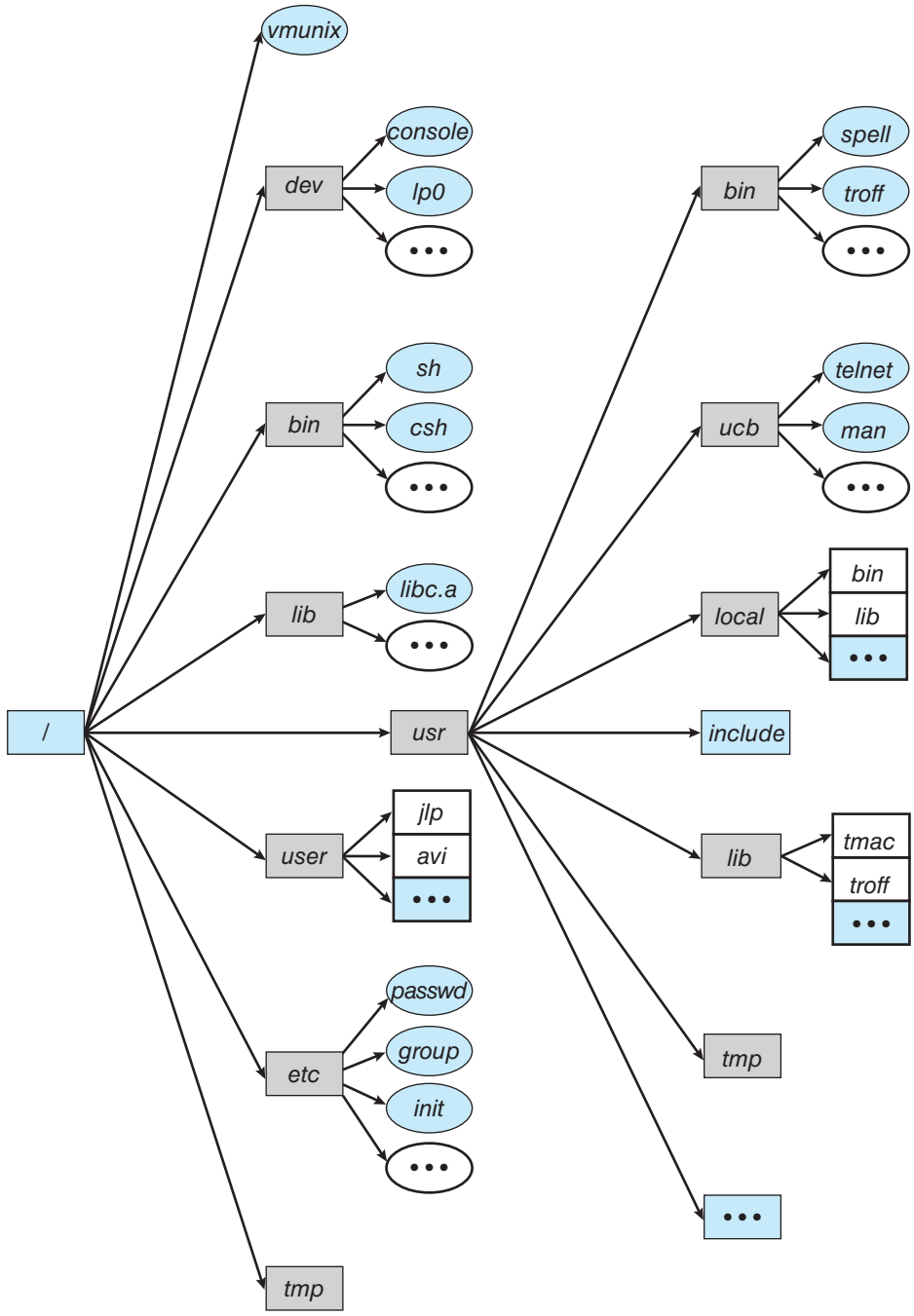

Figure C.3 Typical UNIX directory structure.

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

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

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

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

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

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

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

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

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

C.3.2 Process Control

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

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

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

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

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

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

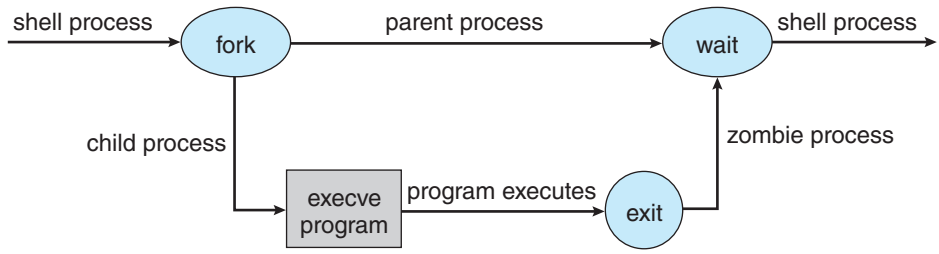

Figure C.4 A shell forks a subprocess to execute a program.

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

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

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

C.3.3 Signals

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

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

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

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

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

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

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

C.3.4 Process Groups

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

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

C.3.5 Information Manipulation

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

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

C.3.6 Library Routines

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

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

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

C.4 User Interface

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

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

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

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

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

C.4.1 Shells and Commands

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

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

% ***ls -l***

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

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

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

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

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

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

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

C.4.2 Standard I/O

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

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

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

Figure C.5 Standard /io/ redirection.

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

C.4.3 Pipelines, Filters, and Shell Scripts

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

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

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

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

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

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

C.5 Process Management

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

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

C.5.1 Process Control Blocks

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

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

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

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

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

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

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

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

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

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

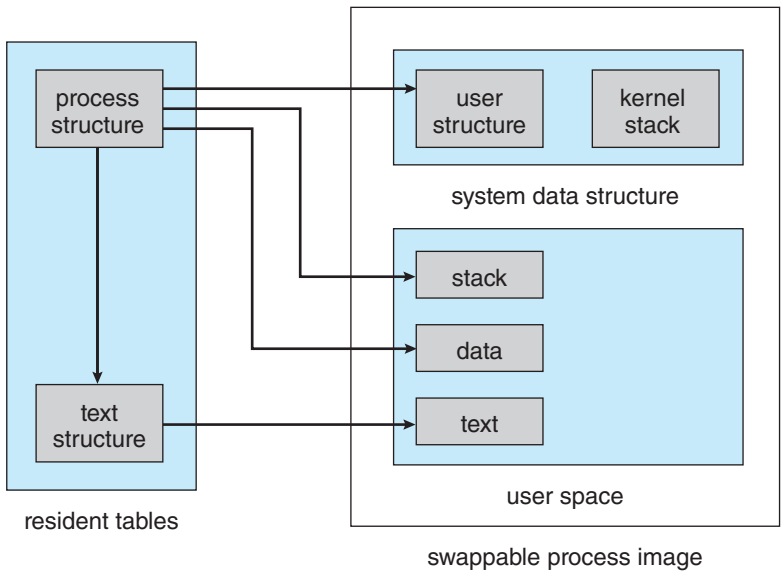

Figure C.6 Finding parts of a process using the process structure.

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

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

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

C.5.2 CPU Scheduling

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

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

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

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

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

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

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

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

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

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

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

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

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

C.6 Memory Management

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

C.7 File System

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

C.7.1 Blocks and Fragments

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

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

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

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

C.7.2 Inodes

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

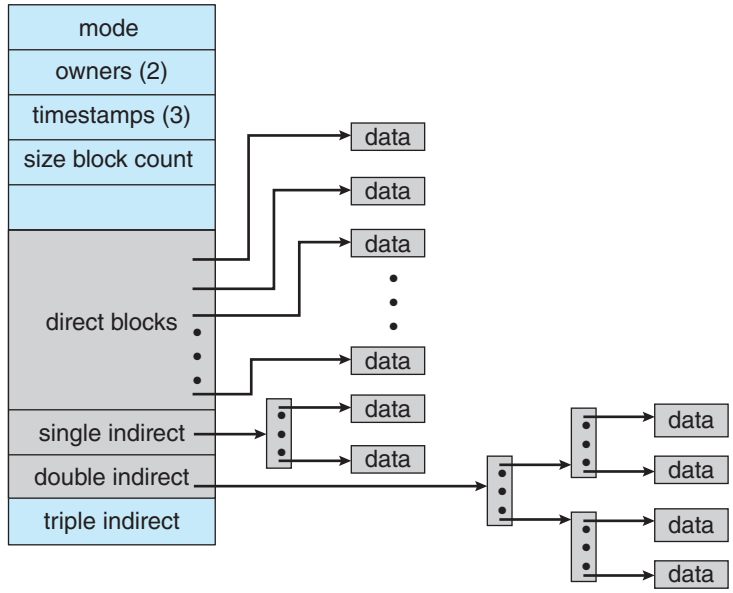

Figure C.7 The UNIX inode.

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

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

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

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

C.7.3 Directories

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

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

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

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

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

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

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

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

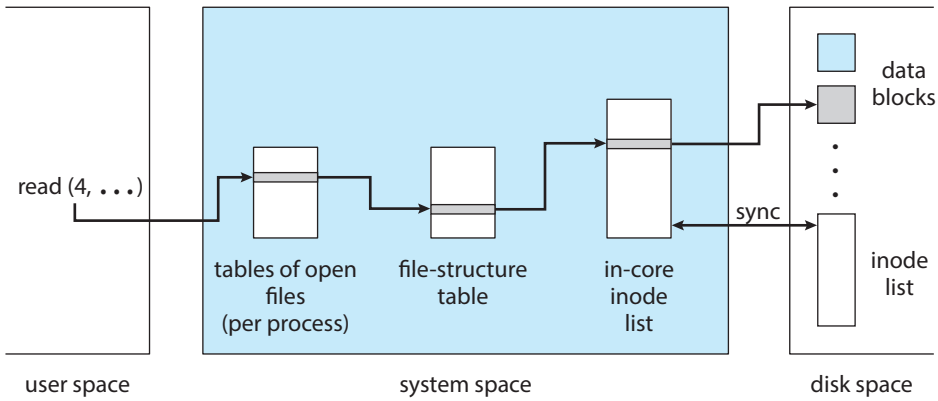

Figure C.8 File-system control blocks.

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

C.7.4 Mapping a File Descriptor to an Inode

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

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

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

C.7.5 Disk Structures

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

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

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

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

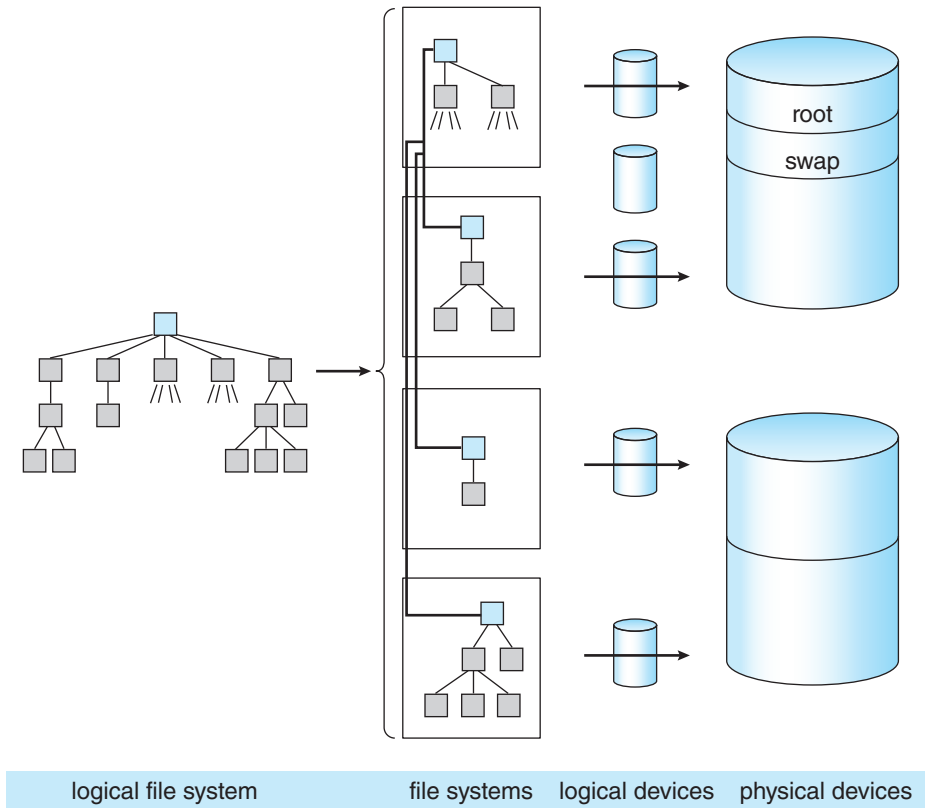

Figure C.9 Mapping of a logical file system to physical devices.

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

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

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

C.7.6 Implementations

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

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

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

C.7.7 Layout and Allocation Policies

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

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

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

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

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

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

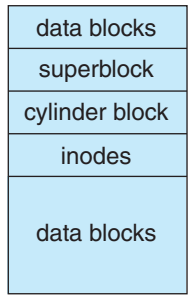

Figure C.10 4.3 BSD cylinder group.

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

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

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

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

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

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

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

C.8 I/O System

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

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

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

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

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

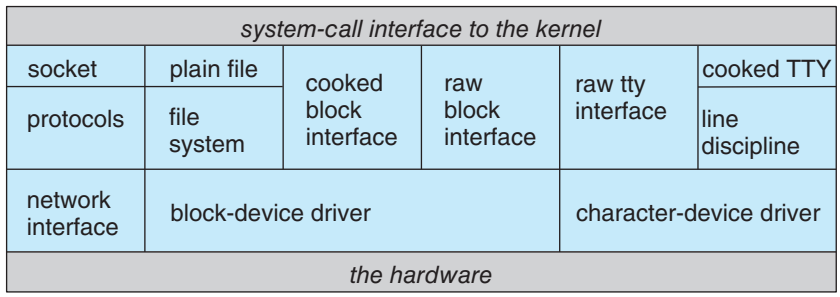

Figure C.11 4.3 BSD kernel I/O structure.

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

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

C.8.1 Block Buffer Cache

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

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

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

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

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

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

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

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

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

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

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

C.8.2 Raw Device Interfaces

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

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

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

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

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

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

C.8.3 C-Lists

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

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

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

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

C.9 Interprocess Communication

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

C.9.1 Sockets

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

C.9.2 Network Support

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Figure C.12 Network reference models and layering.

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

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

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

C.10 Summary

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

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

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

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

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

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

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

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

Further Reading

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

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

Bibliography

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

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

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