Contents

PART 1

Overview

CHAPTER 1

History and Goals

1.1 History of the UNIX System

The UNIX system has been in wide use for over 20 years, and has helped to define many areas of computing. Although numerous organizations have contributed (and still contribute) to the development of the UNIX system, this book will primarily concentrate on the BSD thread of development:

- Bell Laboratories, which invented UNIX
- The Computer Systems Research Group (CSRG) at the University of California at Berkeley, which gave UNIX virtual memory and the reference implementation of TCP/IP
- Berkeley Software Design, Incorporated (BSDI), The FreeBSD Project, and The NetBSD Project, which continue the work started by the CSRG

Origins

The first version of the UNIX system was developed at Bell Laboratories in 1969 by Ken Thompson as a private research project to use an otherwise idle PDP-7. Thompson was joined shortly thereafter by Dennis Ritchie, who not only contributed to the design and implementation of the system, but also invented the C programming language. The system was completely rewritten into C, leaving almost no assembly language. The original elegant design of the system [Ritchie, 1978] and developments of the past 15 years [Ritchie, 1984a; Compton, 1985] have made the UNIX system an important and powerful operating system [Ritchie, 1987].

Ritchie, Thompson, and other early UNIX developers at Bell Laboratories had worked previously on the Multics project [Peirce, 1985; Organick, 1975], which had a strong influence on the newer operating system. Even the name *UNIX* is merely a pun on *Multics;* in areas where Multics attempted to do many tasks, UNIX tried to do one task well. The basic organization of the UNIX filesystem, the idea of using a user process for the command interpreter, the general organization of the filesystem interface, and many other system characteristics, come directly from Multics.

Ideas from various other operating systems, such as the Massachusetts Institute of Technology's (MIT's) CTSS, also have been incorporated. The fork operation to create new processes comes from Berkeley's GENIE (SDS-940, later XDS-940) operating system. Allowing a user to create processes inexpensively led to using one process per command, rather than to commands being run as procedure calls, as is done in Multics.

There are at least three major streams of development of the UNIX system. Figure 1.1 sketches their early evolution; Figure 1.2 (shown on page 6) sketches their more recent developments, especially for those branches leading to 4.4BSD and to System V [Chambers & Quarterman, 1983; Uniejewski, 1985]. The dates given are approximate, and we have made no attempt to show all influences. Some of the systems named in the figure are not mentioned in the text, but are included to show more clearly the relations among the ones that we shall examine.

Research UNIX

The first major editions of UNIX were the Research systems from Bell Laboratories. In addition to the earliest versions of the system, these systems include the **UNIX Time-Sharing System, Sixth Edition,** commonly known as V6, which, in 1976, was the first version widely available outside of Bell Laboratories. Systems are identified by the edition numbers of the *UNIX Programmer's Manual* that were current when the distributions were made.

The UNIX system was distinguished from other operating systems in three important ways:

1. The UNIX system was written in a high-level language.
2. The UNIX system was distributed in source form.
3. The UNIX system provided powerful primitives normally found in only those operating systems that ran on much more expensive hardware.

Most of the system source code was written in C, rather than in assembly language. The prevailing belief at the time was that an operating system had to be written in assembly language to provide reasonable efficiency and to get access to the hardware. The C language itself was at a sufficiently high level to allow it to be compiled easily for a wide range of computer hardware, without its being so complex or restrictive that systems programmers had to revert to assembly language to get reasonable efficiency or functionality. Access to the hardware was provided through assembly-language stubs for the 3 percent of the operating-system functions—such as context switching—that needed them. Although the success of UNIX does not stem solely from its being written in a high-level

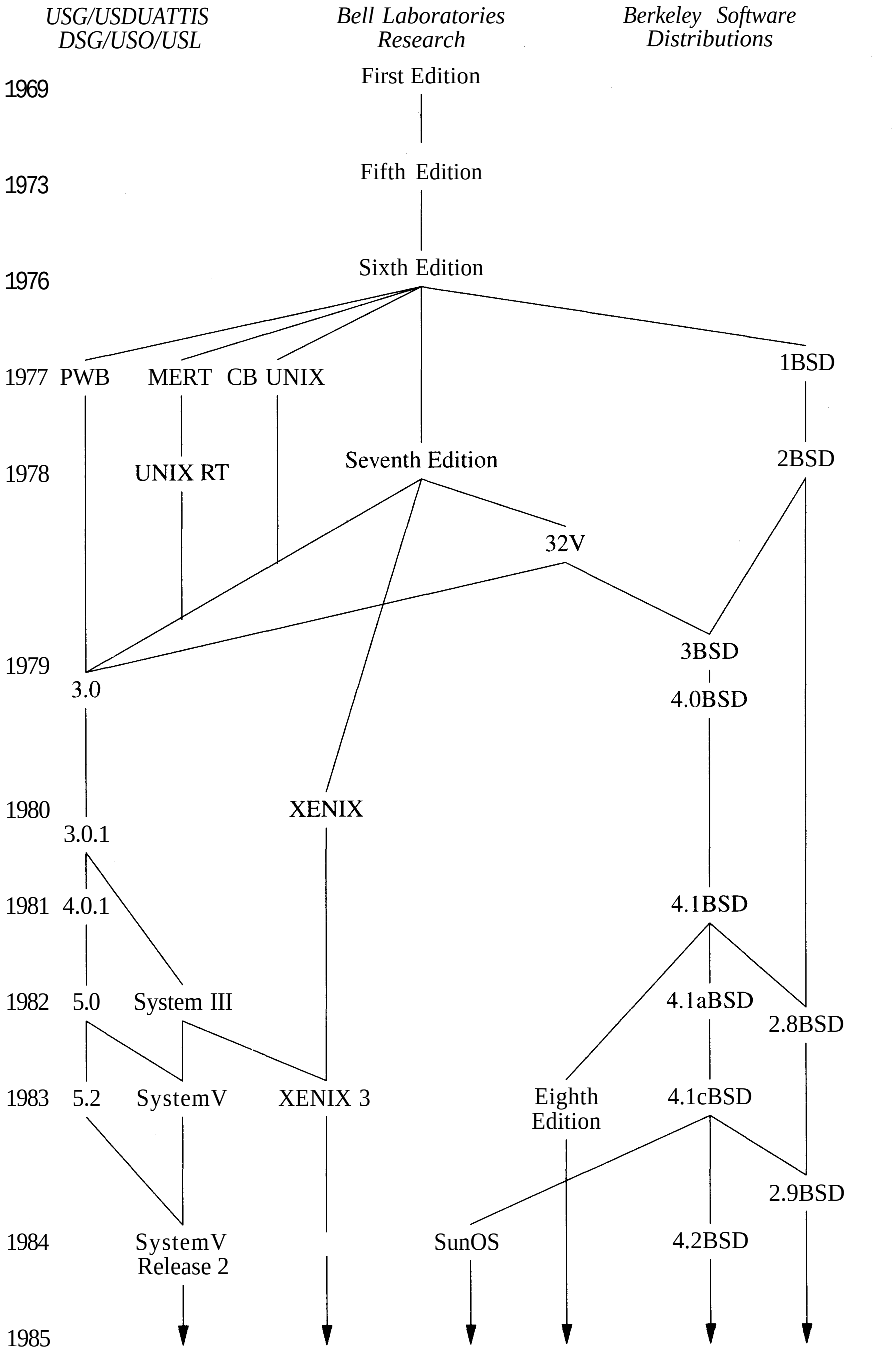


Figure 1.1 The UNIX system family tree, 1969-1985.

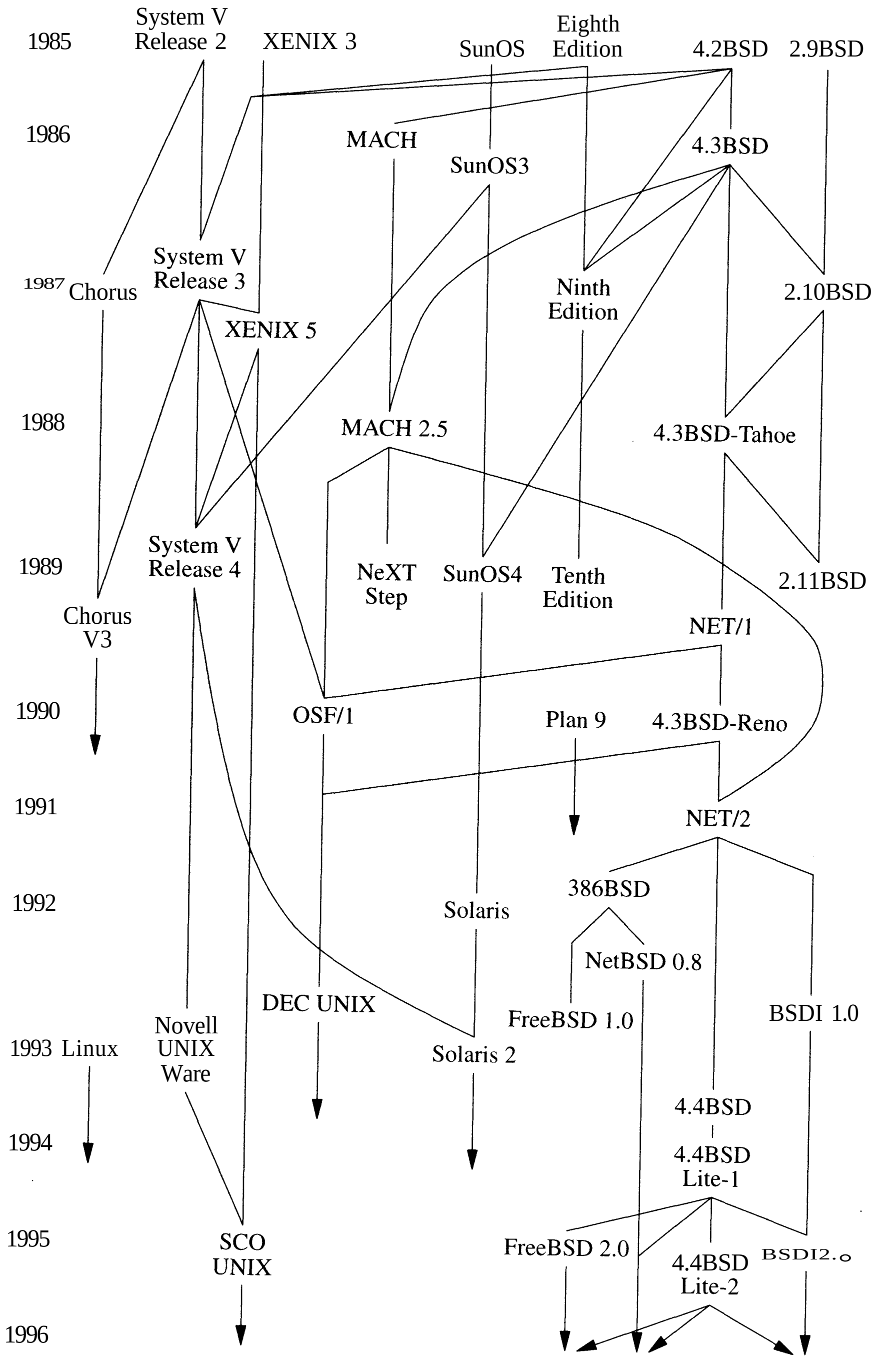


Figure 1.2 The UNIX system family tree, 1986-1996.

language, the use of C was a critical first step [Ritchie et al, 1978; Kernighan & Ritchie, 1978; Kernighan & Ritchie, 1988]. Ritchie's C language is descended [Rosier, 1984] from Thompson's B language, which was itself descended from BCPL [Richards & Whitby-Strevens, 1980]. C continues to evolve [Tuthill, 1985; X3J11, 1988], and there is a variant—C++—that more readily permits data abstraction [Stroustrup, 1984; USENIX, 1987].

The second important distinction of UNIX was its early release from Bell Laboratories to other research environments in source form. By providing source, the system's founders ensured that other organizations would be able not only to use the system, but also to tinker with its inner workings. The ease with which new ideas could be adopted into the system always has been key to the changes that have been made to it. Whenever a new system that tried to upstage UNIX came along, somebody would dissect the newcomer and clone its central ideas into UNIX. The unique ability to use a small, comprehensible system, written in a high-level language, in an environment swimming in new ideas led to a UNIX system that evolved far beyond its humble beginnings.

The third important distinction of UNIX was that it provided individual users with the ability to run multiple processes concurrently and to connect these processes into pipelines of commands. At the time, only operating systems running on large and expensive machines had the ability to run multiple processes, and the number of concurrent processes usually was controlled tightly by a system administrator.

Most early UNIX systems ran on the PDP-11, which was inexpensive and powerful for its time. Nonetheless, there was at least one early port of Sixth Edition UNIX to a machine with a different architecture, the Interdata 7/32 [Miller, 1978]. The PDP-11 also had an inconveniently small address space. The introduction of machines with 32-bit address spaces, especially the VAX-11/780, provided an opportunity for UNIX to expand its services to include virtual memory and networking. Earlier experiments by the Research group in providing UNIX-like facilities on different hardware had led to the conclusion that it was as easy to move the entire operating system as it was to duplicate UNIX's services under another operating system. The first UNIX system with portability as a specific goal was **UNIX Time-Sharing System, Seventh Edition** (V7), which ran on the PDP-11 and the Interdata 8/32, and had a VAX variety called **UNIX/32V Time-Sharing, System Version 1.0** (32V). The Research group at Bell Laboratories has also developed **UNIX Time-Sharing System, Eighth Edition** (V8), **UNIX Time-Sharing System, Ninth Edition** (V9), and **UNIX Time-Sharing System, Tenth Edition** (V10). Their 1996 system is Plan 9.

AT&T UNIX System III and System V

After the distribution of Seventh Edition in 1978, the Research group turned over external distributions to the UNIX Support Group (USG). USG had previously distributed internally such systems as the **UNIX Programmer's Work Bench (PWB),** and had sometimes distributed them externally as well [Mohr, 1985].

USG's first external distribution after Seventh Edition was **UNIX System III (System III),** in 1982, which incorporated features of Seventh Edition, of 32V, and also of several UNIX systems developed by groups other than the Research group. Features of UNIX /RT (a real-time UNIX system) were included, as were many features from PWB. USG released **UNIX System V (System V)** in 1983; that system is largely derived from System III. The court-ordered divestiture of the Bell Operating Companies from AT&T permitted AT&T to market System V aggressively [Wilson, 1985; Bach, 1986].

USG metamorphosed into the UNIX System Development Laboratory (USDL), which released **UNIX System V, Release 2** in 1984. System V, Release 2, Version 4 introduced paging [Miller, 1984; Jung, 1985], including copy-on-write and shared memory, to System V The System V implementation was not based on the Berkeley paging system. USDL was succeeded by AT&T Information Systems (ATTIS), which distributed UNIX **System V, Release 3** in 1987. That system included STREAMS, an IPC mechanism adopted from V8 [Presotto & Ritchie, 1985]. ATTIS was succeeded by UNIX System Laboratories (USL), which was sold to Novell in 1993. Novell passed the UNIX trademark to the X/OPEN consortium, giving the latter sole rights to set up certification standards for using the UNIX name on products. Two years later, Novell sold UNIX to The Santa Cruz Operation (SCO).

Other Organizations

The ease with which the UNIX system can be modified has led to development work at numerous organizations, including the Rand Corporation, which is responsible for the Rand ports mentioned in Chapter 11; Bolt Beranek and Newman (BBN), who produced the direct ancestor of the 4.2BSD networking implementation discussed in Chapter 13; the University of Illinois, which did earlier networking work; Harvard; Purdue; and Digital Equipment Corporation (DEC).

Probably the most widespread version of the UNIX operating system, according to the number of machines on which it runs, is XENIX by Microsoft Corporation and The Santa Cruz Operation. XENIX was originally based on Seventh Edition, but later on System V More recently, SCO purchased UNIX from Novell and announced plans to merge the two systems.

Systems prominently *not* based on UNIX include IBM's OS/2 and Microsoft's Windows 95 and Windows/NT. All these systems have been touted as UNIX killers, but none have done the deed.

Berkeley Software Distributions

The most influential of the non-Bell Laboratories and non-AT&T UNIX development groups was the University of California at Berkeley [McKusick, 1985]. Software from Berkeley is released in **Berkeley Software Distributions** (BSD)—for example, as 4.3BSD. The first Berkeley VAX UNIX work was the addition to 32V of virtual memory, demand paging, and page replacement in 1979 by William Joy and Ozalp Babaoglu, to produce 3BSD [Babaoglu & Joy, 1981]. The reason for the large virtual-memory space of 3BSD was the development of what at the time were large programs, such as Berkeley's *Franz* LISP. This memory-management work convinced the Defense Advanced Research Projects Agency (DARPA) to fund the Berkeley team for the later development of a standard system (4BSD) for DARPA's contractors to use.

A goal of the 4BSD project was to provide support for the DARPA Internet networking protocols, TCP/IP [Cerf & Cain, 1983]. The networking implementation was general enough to communicate among diverse network facilities, ranging from local networks, such as Ethernets and token rings, to long-haul networks, such as DARPA's ARPANET.

We refer to all the Berkeley VAX UNIX systems following 3BSD as 4BSD, although there were really several releases—4.0BSD, 4.1BSD, 4.2BSD, 4.3BSD, 4.3BSD Tahoe, and 4.3BSD Reno. 4BSD was the UNIX operating system of choice for VAXes from the time that the VAX first became available in 1977 until the release of System V in 1983. Most organizations would purchase a 32V license, but would order 4BSD from Berkeley. Many installations inside the Bell System ran 4.1BSD (and replaced it with 4.3BSD when the latter became available). A new virtual-memory system was released with 4.4BSD. The VAX was reaching the end of its useful lifetime, so 4.4BSD was not ported to that machine. Instead, 4.4BSD ran on the newer 68000, SPARC, MIPS, and Intel PC architectures.

The 4BSD work for DARPA was guided by a steering committee that included many notable people from both commercial and academic institutions. The culmination of the original Berkeley DARPA UNIX project was the release of **4.2BSD** in 1983; further research at Berkeley produced **4.3BSD** in mid-1986. The next releases included the **4.3BSD Tahoe** release of June 1988 and the **4.3BSD Reno** release of June 1990. These releases were primarily ports to the Computer Consoles Incorporated hardware platform. Interleaved with these releases were two unencumbered networking releases: the **4.3BSD Netl** release of March 1989 and the **4.3BSD** Net2 release of June 1991. These releases extracted nonproprietary code from 4.3BSD; they could be redistributed freely in source and binary form to companies that and individuals who were not covered by a UNIX source license. The final CSRG release was to have been two versions of 4.4BSD, to be released in June 1993. One was to have been a traditional full source and binary distribution, called 4.4BSD-Encumbered, that required the recipient to have a UNIX source license. The other was to have been a subset of the source, called 4.4BSD-Lite, that contained no licensed code and did not require the recipient to have a UNIX source license. Following these distributions, the CSRG would be dissolved. The 4.4BSD-Encumbered was released as scheduled, but legal action by USL prevented the distribution of 4.4BSD-Lite. The legal action was resolved about 1 year later, and 4.4BSD-Lite was released in April 1994. The last of the money in the CSRG coffers was used to produce a bug-fixed version 4.4BSD-Lite, release 2, that was distributed in June 1995. This release was the true final distribution from the CSRG.

Nonetheless, 4BSD still lives on in all modern implementations of UNIX, and in many other operating systems.

UNIX in the World

Dozens of computer manufacturers, including almost all the ones usually considered major by market share, have introduced computers that run the UNIX system or close derivatives, and numerous other companies sell related peripherals, software packages, support, training, and documentation. The hardware packages involved range from micros through minis, multis, and mainframes to supercomputers. Most of these manufacturers use ports of System V, 4.2BSD, 4.3BSD, 4.4BSD, or mixtures. We expect that, by now, there are probably no more machines running software based on System III, 4.1BSD, or Seventh Edition, although there may well still be PDP-1 1s running 2BSD and other UNIX variants. If there are any Sixth Edition systems still in regular operation, we would be amused to hear about them (our contact information is given at the end of the Preface).

The UNIX system is also a fertile field for academic endeavor. Thompson and Ritchie were given the Association for Computing Machinery Turing award for the design of the system [Ritchie, 1984b]. The UNIX system and related, specially designed teaching systems—such as Tunis [Ewens et al, 1985; Holt, 1983], XINU [Comer, 1984], and MINIX [Tanenbaum, 1987]—are widely used in courses on operating systems. Linus Torvalds reimplemented the UNIX interface in his freely redistributable LINUX operating system. The UNIX system is ubiquitous in universities and research facilities throughout the world, and is ever more widely used in industry and commerce.

Even with the demise of the CSRG, the 4.4BSD system continues to flourish. In the free software world, the FreeBSD and NetBSD groups continue to develop and distribute systems based on 4.4BSD. The FreeBSD project concentrates on developing distributions primarily for the personal-computer (PC) platform. The NetBSD project concentrates on providing ports of 4.4BSD to as many platforms as possible. Both groups based their first releases on the Net2 release, but switched over to the 4.4BSD-Lite release when the latter became available.

The commercial variant most closely related to 4.4BSD is BSD/OS, produced by Berkeley Software Design, Inc. (BSDI). Early BSDI software releases were based on the Net2 release; the current BSDI release is based on 4.4BSD-Lite.

1.2 BSD and Other Systems

The CSRG incorporated features not only from UNIX systems, but also from other operating systems. Many of the features of the 4BSD terminal drivers are from TENEX/TOPS-20. Job control (in concept—not in implementation) is derived from that of TOPS-20 and from that of the MIT Incompatible Timesharing System (ITS). The virtual-memory interface first proposed for 4.2BSD, and since implemented by the CSRG and by several commercial vendors, was based on the file-mapping and page-level interfaces that first appeared in TENEX/TOPS-20. The current 4.4BSD virtual-memory system (see Chapter 5) was adapted from MACH, which was itself an offshoot of 4.3BSD. Multics has often been a reference point in the design of new facilities.

The quest for efficiency has been a major factor in much of the CSRG's work. Some efficiency improvements have been made because of comparisons with the proprietary operating system for the VAX, VMS [Kashtan, 1980; Joy, 1980].

Other UNIX variants have adopted many 4BSD features. AT&T UNIX System V [AT&T, 1987], the IEEE POSIX.l standard [P1003.1, 1988], and the related National Bureau of Standards (NBS) Federal Information Processing Standard (FIPS) have adopted

- Job control (Chapter 2)
- Reliable signals (Chapter 4)
- Multiple file-access permission groups (Chapter 6)
- Filesystem interfaces (Chapter 7)

The X/OPEN Group, originally comprising solely European vendors, but now including most U.S. UNIX vendors, produced the *X/OPEN Portability Guide* [X/OPEN, 1987] and, more recently, the *Spec 1170 Guide.* These documents specify both the kernel interface and many of the utility programs available to UNIX system users. When Novell purchased UNIX from AT&T in 1993, it transferred exclusive ownership of the UNIX name to X/OPEN. Thus, all systems that want to brand themselves as UNIX must meet the X/OPEN interface specifications. The X/OPEN guides have adopted many of the POSIX facilities. The POSIX.l standard is also an ISO International Standard, named SC22 WG15. Thus, the POSIX facilities have been accepted in most UNIX-like systems worldwide.

The 4BSD *socket* interprocess-communication mechanism (see Chapter 11) was designed for portability, and was immediately ported to AT&T System III, although it was never distributed with that system. The 4BSD implementation of the TCP/IP networking protocol suite (see Chapter 13) is widely used as the basis for further implementations on systems ranging from AT&T 3B machines running System V to VMS to IBM PCs.

The CSRG cooperated closely with vendors whose systems are based on 4.2BSD and 4.3BSD. This simultaneous development contributed to the ease of further ports of 4.3BSD, and to ongoing development of the system.

The Influence of the User Community

Much of the Berkeley development work was done in response to the user community. Ideas and expectations came not only from DARPA, the principal direct-funding organization, but also from users of the system at companies and universities worldwide.

The Berkeley researchers accepted not only ideas from the user community, but also actual software. Contributions to 4BSD came from universities and other organizations in Australia, Canada, Europe, and the United States. These contributions included major features, such as autoconfiguration and disk quotas. A few ideas, such as *thefcntl* system call, were taken from System V, although licensing and pricing considerations prevented the use of any actual code from System III or System V in 4BSD. In addition to contributions that were included in the distributions proper, the CSRG also distributed a set of user-contributed software.

An example of a community-developed facility is the public-domain time-zone-handling package that was adopted with the 4.3BSD Tahoe release. It was designed and implemented by an international group, including Arthur Olson, Robert Elz, and Guy Harris, partly because of discussions in the USENET newsgroup **comp.std.unix.** This package takes time-zone-conversion rules completely out of the C library, putting them in files that require no system-code changes to change time-zone rules; this change is especially useful with binary-only distributions of UNIX. The method also allows individual processes to choose rules, rather than keeping one ruleset specification systemwide. The distribution includes a large database of rules used in many areas throughout the world, from China to Australia to Europe. Distributions of the 4.4BSD system are thus simplified because it is not necessary to have the software set up differently for different destinations, as long as the whole database is included. The adoption of the time-zone package into BSD brought the technology to the attention of commercial vendors, such as Sun Microsystems, causing them to incorporate it into their systems.

Berkeley solicited electronic mail about bugs and the proposed fixes. The UNIX software house MT XINU distributed a bug list compiled from such submissions. Many of the bug fixes were incorporated in later distributions. There is constant discussion of UNIX in general (including 4.4BSD) in the USENET **comp.unix** newsgroups, which are distributed on the Internet; both the Internet and USENET are international in scope. There was another USENET newsgroup dedicated to 4BSD bugs: **comp.bugs.4bsd.** Few ideas were accepted by Berkeley directly from these newsgroups' associated mailing lists because of the difficulty of sifting through the voluminous submissions. Later, a moderated newsgroup dedicated to the CSRG-sanctioned fixes to such bugs, called **comp.bugs.4bsd.bugfixes,** was created. Discussions in these newsgroups sometimes led to new facilities being written that were later incorporated into the system.

1.3 Design Goals of 4BSD

4BSD is a research system developed for and partly by a research community, and, more recently, a commercial community. The developers considered many design issues as they wrote the system. There were nontraditional considerations and inputs into the design, which nevertheless yielded results with commercial importance.

The early systems were technology driven. They took advantage of current hardware that was unavailable in other UNIX systems. This new technology included

- Virtual-memory support
- Device drivers for third-party (non-DEC) peripherals
- Terminal-independent support libraries for screen-based applications; numerous applications were developed that used these libraries, including the screen-based editor vi

4BSD's support of numerous popular third-party peripherals, compared to the AT&T distribution's meager offerings in 32V, was an important factor in 4BSD popularity. Until other vendors began providing their own support of 4.2BSD-based systems, there was no alternative for universities that had to minimize hardware costs.

Terminal-independent screen support, although it may now seem rather pedestrian, was at the time important to the Berkeley software's popularity.

4.2BSD Design Goals

DARPA wanted Berkeley to develop 4.2BSD as a standard research operating system for the VAX. Many new facilities were designed for inclusion in 4.2BSD. These facilities included a completely revised virtual-memory system to support processes with large sparse address space, a much higher-speed filesystem, interprocess-communication facilities, and networking support. The high-speed filesystem and revised virtual-memory system were needed by researchers doing computer-aided design and manufacturing (CAD/CAM), image processing, and artificial intelligence (AI). The interprocess-communication facilities were needed by sites doing research in distributed systems. The motivation for providing networking support was primarily DARPA's interest in connecting their researchers through the 56-Kbit-per-second ARPA Internet (although Berkeley was also interested in getting good performance over higher-speed local-area networks).

No attempt was made to provide a true distributed operating system [Popek, 1981]. Instead, the traditional ARPANET goal of resource sharing was used. There were three reasons that a resource-sharing design was chosen:

1. The systems were widely distributed and demanded administrative autonomy. At the time, a true distributed operating system required a central administrative authority.
2. The known algorithms for tightly coupled systems did not scale well.
3. Berkeley's charter was to incorporate current, proven software technology, rather than to develop new, unproven technology.

Therefore, easy means were provided for remote login *(rlogin, telnef),* file transfer *(rcp, ftp),* and remote command execution *(rsh),* but all host machines retained separate identities that were not hidden from the users.

Because of time constraints, the system that was released as 4.2BSD did not include all the facilities that were originally intended to be included. In particular, the revised virtual-memory system was not part of the 4.2BSD release. The CSRG did, however, continue its ongoing work to track fast-developing hardware technology in several areas. The networking system supported a wide range of hardware devices, including multiple interfaces to 10-Mbit-per-second Ethernet, token ring networks, and to NSC's Hyperchannel. The kernel sources were modularized and rearranged to ease portability to new architectures, including to microprocessors and to larger machines.

4.3BSD Design Goals

Problems with 4.2BSD were among the reasons for the development of 4.3BSD. Because 4.2BSD included many new facilities, it suffered a loss of performance compared to 4.1BSD, partly because of the introduction of symbolic links. Some pernicious bugs had been introduced, particularly in the TCP protocol implementation. Some facilities had not been included due to lack of time. Others, such as TCP/IP subnet and routing support, had not been specified soon enough by outside parties for them to be incorporated in the 4.2BSD release.

Commercial systems usually maintain backward compatibility for many releases, so as not to make existing applications obsolete. Maintaining compatibility is increasingly difficult, however, so most research systems maintain little or no backward compatibility. As a compromise for other researchers, the BSD releases were usually backward compatible for one release, but had the deprecated facilities clearly marked. This approach allowed for an orderly transition to the new interfaces without constraining the system from evolving smoothly. In particular, backward compatibility of 4.3BSD with 4.2BSD was considered highly desirable for application portability.

The C language interface to 4.3BSD differs from that of 4.2BSD in only a few commands to the terminal interface and in the use of one argument to one IPC system call (*select;* see Section 6.4). A flag was added in 4.3BSD to the system call that establishes a signal handler to allow a process to request the 4.1BSD semantics for signals, rather than the 4.2BSD semantics (see Section 4.7). The sole purpose of the flag was to allow existing applications that depended on the old semantics to continue working without being rewritten.

The implementation changes between 4.2BSD and 4.3BSD generally were not visible to users, but they were numerous. For example, the developers made changes to improve support for multiple network-protocol families, such as XEROX NS, in addition to TCP/IP.

The second release of 4.3BSD, hereafter referred to as 4.3BSD Tahoe, added support for the Computer Consoles, Inc. (CCI) Power 6 (Tahoe) series of minicomputers in addition to the VAX. Although generally similar to the original release of 4.3BSD for the VAX, it included many modifications and new features.

The third release of 4.3BSD, hereafter referred to as **4.3BSD-Reno,** added ISO/OSI networking support, a freely redistributable implementation of NFS, and the conversion to and addition of the POSIX.l facilities.

4.4BSD Design Goals

4.4BSD broadened the 4.3BSD hardware base, and now supports numerous architectures, including Motorola 68K, Sun SPARC, MIPS, and Intel PCs.

The 4.4BSD release remedies several deficiencies in 4.3BSD. In particular, the virtual-memory system needed to be and was completely replaced. The new virtual-memory system provides algorithms that are better suited to the large memories currently available, and is much less dependent on the VAX architecture. The 4.4BSD release also added an implementation of networking protocols in the International Organization for Standardization (ISO) suite, and further TCP/IP performance improvements and enhancements.

The terminal driver had been carefully kept compatible not only with Seventh Edition, but even with Sixth Edition. This feature had been useful, but is increasingly less so now, especially considering the lack of orthogonality of its commands and options. In 4.4BSD, the CSRG replaced it with a POSIX-compatible terminal driver; since System V is compliant with POSIX, the terminal driver is compatible with System V. POSIX compatibility in general was a goal. POSIX support is not limited to kernel facilities such as termios and sessions, but rather also includes most POSIX utilities.

The most critical shortcoming of 4.3BSD was the lack of support for multiple filesystems. As is true of the networking protocols, there is no single filesystem that provides enough speed and functionality for all situations. It is frequently necessary to support several different filesystem protocols, just as it is necessary to run several different network protocols. Thus, 4.4BSD includes an object-oriented interface to filesystems similar to Sun Microsystems' vnode framework. This framework supports multiple local and remote filesystems, much as multiple networking protocols are supported by 4.3BSD [Sandberg et al, 1985]. The vnode interface has been generalized to make the operation set dynamically extensible and to allow filesystems to be stacked. With this structure, 4.4BSD supports numerous filesystem types, including loopback, union, and uid/gid mapping layers, plus an ISO9660 filesystem, which is particularly useful for CD-ROMs. It also supports Sun's Network filesystem (NFS) Versions 2 and 3 and a new local disk-based log-structured filesystem.

Original work on the flexible configuration of IPC processing modules was done at Bell Laboratories in UNIX Eighth Edition [Presotto & Ritchie, 1985]. This *stream I/O system* was based on the UNIX character I/O system. It allowed a user process to open a raw terminal port and then to insert appropriate kernel-processing modules, such as one to do normal terminal line editing. Modules to process network protocols also could be inserted. Stacking a terminal-processing module on top of a network-processing module allowed flexible and efficient implementation of *network virtual terminals* within the kernel. A problem with stream modules, however, is that they are inherently linear in nature, and thus they do not adequately handle the fan-in and fan-out associated with multiplexing in datagram-based networks; such multiplexing is done in device drivers, below the modules proper. The Eighth Edition stream I/O system was adopted in System V, Release 3 as the STREAMS system.

The design of the networking facilities for 4.2BSD took *a* different approach, based on the *socket* interface and a flexible multilayer network architecture. This design allows a single system to support multiple sets of networking protocols with stream, datagram, and other types of access. Protocol modules may deal with multiplexing of data from different connections onto a single transport medium, as well as with demultiplexing of data for different protocols and connections received from each network device. The 4.4BSD release made small extensions to the socket interface to allow the implementation of the ISO networking protocols.

1.4 Release Engineering

The CSRG was always a small group of software developers. This resource limitation required careful software-engineering management. Careful coordination was needed not only of the CSRG personnel, but also of members of the general community who contributed to the development of the system. Even though the CSRG is no more, the community still exists; it continues the BSD traditions with FreeBSD, NetBSD, and BSDI.

Major CSRG distributions usually alternated between

- Major new facilities: 3BSD, 4.0BSD, 4.2BSD, 4.4BSD
- Bug fixes and efficiency improvements: 4.1BSD, 4.3BSD

This alternation allowed timely release, while providing for refinement and correction of the new facilities and for elimination of performance problems produced by the new facilities. The timely follow-up of releases that included new facilities reflected the importance that the CSRG placed on providing a reliable and robust system on which its user community could depend.

Developments from the CSRG were released in three steps: alpha, beta, and final, as shown in Table 1.1. Alpha and beta releases were not true distributions—they were test systems. Alpha releases were normally available to only a few sites, most of those within the University. More sites got beta releases, but they did not get these releases directly; a tree structure was imposed to allow bug reports, fixes, and new software to be collected, evaluated, and checked for redundancies by first-level sites before forwarding to the CSRG. For example, 4.1aBSD ran at more than 100 sites, but there were only about 15 primary beta sites. The beta-test tree allowed the developers at the CSRG to concentrate on actual development, rather than sifting through details from every beta-test site. This book was reviewed for technical accuracy by a similar process.

Table 1.1 Test steps for the release of 4.2BSD.

	Release steps			
Description	**alpha**	**internal**	**beta**	**final**
name:	4.1aBSD	4.1bBSD	4.1cBSD	4.2BSD
major new facility:	networking	fast filesystem	IPC	revised signals

Many of the primary beta-test personnel not only had copies of the release running on their own machines, but also had login accounts on the development machine at Berkeley. Such users were commonly found logged in at Berkeley over the Internet, or sometimes via telephone dialup, from places far away, such as Australia, England, Massachusetts, Utah, Maryland, Texas, and Illinois, and from closer places, such as Stanford. For the 4.3BSD and 4.4BSD releases, certain accounts and users had permission to modify the master copy of the system source directly. Several facilities, such as the Fortran and C compilers, as well as important system programs, such as *telnet andftp*, include significant contributions from people who did not work for the CSRG. One important exception to this approach was that changes to the kernel were made by only the CSRG personnel, although the changes often were suggested by the larger community.

People given access to the master sources were carefully screened beforehand, but were not closely supervised. Their work was checked at the end of the beta-test period by the CSRG personnel, who did a complete comparison of the source of the previous release with the current master sources—for example, of 4.3BSD with 4.2BSD. Facilities deemed inappropriate, such as new options to the directory-listing command or a changed return value for the *fseek()* library routine, were removed from the source before final distribution.

This process illustrates an *advantage* of having only a few principal developers: The developers all knew the whole system thoroughly enough to be able to coordinate their own work with that of other people to produce a coherent final system. Companies with large development organizations find this result difficult to duplicate.

There was no CSRG marketing division. Thus, technical decisions were made largely for technical reasons, and were not driven by marketing promises. The Berkeley developers were fanatical about this position, and were well known for never promising delivery on a specific date.

References

AT&T, 1987.
AT&T, *The System V Interface Definition (SVID),* Issue 2, American Telephone and Telegraph, Murray Hill, NJ, January 1987.

Babaoglu & Joy, 1981.
O. Babaoglu & W. N. Joy, "Converting a Swap-Based System to Do Paging in an Architecture Lacking Page-Referenced Bits," *Proceedings of the Eighth Symposium on Operating Systems Principles,* pp. 78-86, December 1981.

Bach, 1986.
M. J. Bach, *The Design of the UNIX Operating System,* Prentice-Hall, Englewood Cliffs, NJ, 1986.

Cerf& Cain, 1983.
V. Cerf & E. Cain, *The DoD Internet Architecture Model,* pp. 307-318, Elsevier Science, Amsterdam, Netherlands, 1983.

Chambers & Quarterman, 1983.
J. B. Chambers & J. S. Quarterman, "UNIX System V and 4.1C BSD," *USENIX Association Conference Proceedings,* pp. 267-291, June 1983.

Comer, 1984.
D. Comer, *Operating System Design: The Xinu Approach,* Prentice-Hall, Englewood Cliffs, NJ, 1984.

Compton, 1985.
M. Compton, editor, "The Evolution of UNIX," *UNIX Review,* vol. 3, no. 1, January 1985.

Ewensetal, 1985.
P. Ewens, D. R. Blythe, M. Funkenhauser, & R. C. Holt, "Tunis: A Distributed Multiprocessor Operating System," *USENIX Association Conference Proceedings,* pp. 247-254, June 1985.

Holt, 1983.
R. C. Holt, *Concurrent Euclid, the UNIX System, and Tunis,* Addison-Wesley, Reading, MA, 1983.

Joy, 1980,
W. N. Joy, "Comments on the Performance of UNIX on the VAX," Technical Report, University of California Computer System Research Group, Berkeley, CA, April 1980.

Jung, 1985.
R. S. Jung, "Porting the AT&T Demand Paged UNIX Implementation to Microcomputers," *USENIX Association Conference Proceedings,* pp. 361-370, June 1985.

Kashtan, 1980.
D. L. Kashtan, "UNIX and VMS: Some Performance Comparisons," Technical Report, SRI International, Menlo Park, CA, February 1980.

Kernighan & Ritchie, 1978.
B. W. Kernighan & D. M. Ritchie, *The C Programming Language,* Prentice-Hall, Englewood Cliffs, NJ, 1978.

Kernighan & Ritchie, 1988.
B. W. Kernighan & D. M. Ritchie, *The C Programming Language,* 2nd ed, Prentice-Hall, Englewood Cliffs, NJ, 1988.

McKusick, 1985.
M. K. McKusick, "A Berkeley Odyssey," *UNIX Review,* vol. 3, no. 1, p. 30, January 1985.

Miller, 1978.
R. Miller, "UNIX—A Portable Operating System," *ACM Operating System Review,* vol. 12, no. 3, pp. 32-37, July 1978.

Miller, 1984.
R. Miller, "A Demand Paging Virtual Memory Manager for System V," *USENIX Association Conference Proceedings,* p. 178-182, June 1984.

Mohr, 1985.
A. Mohr, "The Genesis Story," *UNIX Review,* vol. 3, no. 1, p. 18, January 1985.

Organick, 1975.
E. I. Organick, *The Multics System: An Examination of Its Structure,* MIT Press, Cambridge, MA, 1975.

P1003.1, 1988.
P1003.1, *IEEE P1003.1 Portable Operating System Interface for Computer Environments (POSIX),* Institute of Electrical and Electronic Engineers, Piscataway, NJ, 1988.

Peirce, 1985.
N. Peirce, "Putting UNIX In Perspective: An Interview with Victor Vyssotsky," *UNIX Review,* vol. 3, no. 1, p. 58, January 1985.

Popek, 1981.
B. Popek, "Locus: A Network Transparent, High Reliability Distributed System," *Proceedings of the Eighth Symposium on Operating Systems Principles,* p. 169-177, December 1981.

Presotto & Ritchie, 1985.
D. L. Presotto & D. M. Ritchie, "Interprocess Communication in the Eighth Edition UNIX System," *USENIX Association Conference Proceedings,* p. 309-316, June 1985.

Richards & Whitby-Strevens, 1980.
M. Richards & C. Whitby-Strevens, *BCPL: The Language and Its Compiler,* Cambridge University Press, Cambridge, U.K., 1980, 1982.

Ritchie, 1978.
D. M. Ritchie, "A Retrospective," *Bell System Technical Journal,* vol. 57, no. 6, p. 1947-1969, July-August 1978.

Ritchie, 1984a.
D. M. Ritchie, "The Evolution of the UNIX Time-Sharing System," *AT&T Bell Laboratories Technical Journal,* vol. 63, no. 8, p. 1577-1593, October 1984.

Ritchie, 1984b.
D. M. Ritchie, "Reflections on Software Research," *Comm ACM,* vol. 27, no. 8, p. 758-760, 1984.

Ritchie, 1987.
D. M. Ritchie, "Unix: A Dialectic," *USENIX Association Conference Proceedings,* p. 29-34, January 1987.

Ritchie et al, 1978.
D. M. Ritchie, S. C. Johnson, M. E. Lesk, & B. W. Kernighan, "The C Programming Language," *Bell System Technical Journal,* vol. 57, no. 6, p. 1991-2019, July-August 1978.

Rosier, 1984.
L. Rosier, "The Evolution of C—Past and Future," *AT&T Bell Laboratories Technical Journal,* vol. 63, no. 8, pp. 1685-1699, October 1984.

Sandberg et al, 1985.
R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, & B. Lyon, "Design and Implementation of the Sun Network Filesystem," *USENIX Association Conference Proceedings,* pp. 119-130, June 1985.

Stroustrup, 1984.
B. Stroustrup, "Data Abstraction in C," *AT&T Bell Laboratories Technical Journal,* vol. 63, no. 8, pp. 1701-1732, October 1984.

Tanenbaum, 1987.
A. S. Tanenbaum, *Operating Systems: Design and Implementation,* Prentice-Hall, Englewood Cliffs, NJ, 1987.

Tuthill, 1985.
B. Tuthill, "The Evolution of C: Heresy and Prophecy," *UNIX Review,* vol. 3, no. 1, p. 80, January 1985.

Uniejewski, 1985.
J. Uniejewski, *UNIX System V and BSD4.2 Compatibility Study,* Apollo Computer, Chelmsford, MA, March 1985.

USENIX, 1987.
USENIX, *Proceedings of the* C++ *Workshop,* USENIX Association, Berkeley, CA, November 1987.

Wilson, 1985.
O. Wilson, "The Business Evolution of the UNIX System," *UNIX Review,* vol. 3, no. 1, p. 46, January 1985.

X3J11, 1988.
X3J11, *X3.159 Programming Language C Standard,* Global Press, Santa Ana, CA, 1988.

X/OPEN, 1987.
X/OPEN, *The X/OPEN Portability Guide (XPG),* Issue 2, Elsevier Science, Amsterdam, Netherlands, 1987.

CHAPTER 2

Design Overview of 4.4BSD

2.1 4.4BSD Facilities and the Kernel

The 4.4BSD kernel provides four basic facilities: processes, a filesystem, communications, and system startup. This section outlines where each of these four basic services is described in this book.

1. Processes constitute a thread of control in an address space. Mechanisms for creating, terminating, and otherwise controlling processes are described in Chapter 4. The system multiplexes separate virtual-address spaces for each process; this memory management is discussed in Chapter 5.

2. The user interface to the filesystem and devices is similar; common aspects are discussed in Chapter 6. The filesystem is a set of named files, organized in a tree-structured hierarchy of directories, and of operations to manipulate them, as presented in Chapter 7. Files reside on physical media such as disks. 4.4BSD supports several organizations of data on the disk, as set forth in Chapter 8. Access to files on remote machines is the subject of Chapter 9. Terminals are used to access the system; their operation is the subject of Chapter 10.

3. Communication mechanisms provided by traditional UNIX systems include simplex reliable byte streams between related processes (see pipes, Section 11.1), and notification of exceptional events (see signals, Section 4.7). 4.4BSD also has a general interprocess-communication facility. This facility, described in Chapter 11, uses access mechanisms distinct from those of the filesystem, but, once a connection is set up, a process can access it as though it were a pipe. There is a general networking framework, discussed in Chapter 12, that is normally used as a layer underlying the IPC facility. Chapter 13 describes a particular networking implementation in detail.

4. Any real operating system has operational issues, such as how to start it running. Startup and operational issues are described in Chapter 14.

Sections 2.3 through 2.14 present introductory material related to Chapters 3 through 14. We shall define terms, mention basic system calls, and explore historical developments. Finally, we shall give the reasons for many major design decisions.

The Kernel

The *kernel* is the part of the system that runs in protected mode and mediates access by all user programs to the underlying hardware (e.g., CPU, disks, terminals, network links) and software constructs (e.g., filesystem, network protocols). The kernel provides the basic system facilities; it creates and manages processes, and provides functions to access the filesystem and communication facilities. These functions, called *system calls,* appear to user processes as library subroutines. These system calls are the only interface that processes have to these facilities. Details of the system-call mechanism are given in Chapter 3, as are descriptions of several kernel mechanisms that do not execute as the direct result of a process doing a system call.

A *kernel,* in traditional operating-system terminology, is a small nucleus of software that provides only the minimal facilities necessary for implementing additional operating-system services. In contemporary research operating systems—such as Chorus [Rozier et al, 1988], Mach [Accetta et al, 1986], Tunis [Ewens et al, 1985], and the V Kernel [Cheriton, 1988]—this division of functionality is more than just a logical one. Services such as filesystems and networking protocols are implemented as client application processes of the nucleus or kernel.

The 4.4BSD kernel is not partitioned into multiple processes. This basic design decision was made in the earliest versions of UNIX. The first two implementations by Ken Thompson had no memory mapping, and thus made no hardware-enforced distinction between user and kernel space [Ritchie, 1988]. A message-passing system could have been implemented as readily as the actually implemented model of kernel and user processes. The monolithic kernel was chosen for simplicity and performance. And the early kernels were small; the inclusion of facilities such as networking into the kernel has increased its size. The current trend in operating-systems research is to reduce the kernel size by placing such services in user space.

Users ordinarily interact with the system through a command-language interpreter, called a *shell,* and perhaps through additional user application programs. Such programs and the shell are implemented with processes. Details of such programs are beyond the scope of this book, which instead concentrates almost exclusively on the kernel.

Sections 2.3 and 2.4 describe the services provided by the 4.4BSD kernel, and give an overview of the latter's design. Later chapters describe the detailed design and implementation of these services as they appear in 4.4BSD.

.2 Kernel Organization

In this section, we view the organization of the 4.4BSD kernel in two ways:

1. As a static body of software, categorized by the functionality offered by the modules that make up the kernel
2. By its dynamic operation, categorized according to the services provided to users

The largest part of the kernel implements the system services that applications access through system calls. In 4.4BSD, this software has been organized according to the following:

- Basic kernel facilities: timer and system-clock handling, descriptor management, and process management
- Memory-management support: paging and swapping
- Generic system interfaces: the I/O, control, and multiplexing operations performed on descriptors
- The filesystem: files, directories, pathname translation, file locking, and I/O buffer management
- Terminal-handling support: the terminal-interface driver and terminal line disciplines
- Interprocess-communication facilities: sockets
- Support for network communication: communication protocols and generic network facilities, such as routing

Most of the software in these categories is machine independent and is portable across different hardware architectures.

The machine-dependent aspects of the kernel are isolated from the mamstream code. In particular, none of the machine-independent code contains conditional code for specific architectures. When an architecture-dependent action is needed, the machine-independent code calls an architecture-dependent function that is located in the machine-dependent code. The software that is machine dependent includes

- Low-level system-startup actions
- Trap and fault handling
- Low-level manipulation of the run-time context of a process
- Configuration and initialization of hardware devices
- Run-time support for I/O devices

Table 2.1 Machine-independent software in the 4.4BSD kernel.

Category	Lines of code	Percentage of kernel
headers	9,393	4.6
initialization	1,107	0.6
kernel facilities	8,793	4.4
generic interfaces	4,782	2.4
interprocess communication	4,540	2.2
terminal handling	3,911	1.9
virtual memory	11,813	5.8
vnode management	7,954	3.9
filesystem naming	6,550	3.2
fast filestore	4,365	2.2
log-structure filestore	4,337	2.1
memory-based filestore	645	0.3
cd9660 filesystem	4,177	2.1
miscellaneous filesystems (10)	12,695	6.3
network filesystem	17,199	8.5
network communication	8,630	4.3
internet protocols	11,984	5.9
ISO protocols	23,924	11.8
X.25 protocols	10,626	5.3
XNS protocols	5,192	2.6
total machine independent	162,617	80.4

Table 2.1 summarizes the machine-independent software that constitutes the 4.4BSD kernel for the HP300. The numbers in column 2 are for lines of C source code, header files, and assembly language. Virtually all the software in the kernel is written in the C programming language; less than 2 percent is written in assembly language. As the statistics in Table 2.2 show, the machine-dependent software, excluding HP/UX and device support, accounts for a minuscule 6.9 percent of the kernel.

Only a small part of the kernel is devoted to initializing the system. This code is used when the system is *bootstrapped* into operation and is responsible for setting up the kernel hardware and software environment (see Chapter 14). Some operating systems (especially those with limited physical memory) discard or *overlay* the software that performs these functions after that software has been executed. The 4.4BSD kernel does not reclaim the memory used by the startup code because that memory space is barely 0.5 percent of the kernel resources used on a typical machine. Also, the startup code does not appear in one place in the kernel—it is scattered throughout, and it usually appears in places logically associated with what is being initialized.

Table 2.2 Machine-dependent software for the HP300 in the 4.4BSD kernel.

Category	Lines of code	Percentage of kernel
machine dependent headers	1,562	0.8
device driver headers	3,495	1.7
device driver source	17,506	8.7
virtual memory	3,087	1.5
other machine dependent	6,287	3.1
routines in assembly language	3,014	1.5
HP/UX compatibility	4,683	2.3
total machine dependent	39,634	19.6

2.3 Kernel Services

The boundary between the kernel- and user-level code is enforced by hardware-protection facilities provided by the underlying hardware. The kernel operates in a separate address space that is inaccessible to user processes. Privileged operations—such as starting I/O and halting the central processing unit (CPU)—are available to only the kernel. Applications request services from the kernel with *system calls.* System calls are used to cause the kernel to execute complicated operations, such as writing data to secondary storage, and simple operations, such as returning the current time of day. All system calls appear *synchronous* to applications: The application does not run while the kernel does the actions associated with a system call. The kernel may finish some operations associated with a system call after it has returned. For example, a *write* system call will copy the data to be written from the user process to a kernel buffer while the process waits, but will usually return from the system call before the kernel buffer is written to the disk.

A system call usually is implemented as a hardware trap that changes the CPU's execution mode and the current address-space mapping. Parameters supplied by users in system calls are validated by the kernel before being used. Such checking ensures the integrity of the system. All parameters passed into the kernel are copied into the kernel's address space, to ensure that validated parameters are not changed as a side effect of the system call. System-call results are returned by the kernel, either in hardware registers or by their values being copied to user-specified memory addresses. Like parameters passed into the kernel, addresses used for the return of results must be validated to ensure that they are part of an application's address space. If the kernel encounters an error while processing a system call, it returns an error code to the user. For the C programming language, this error code is stored in the global variable *errno,* and the function that executed the system call returns the value -1.

User applications and the kernel operate independently of each other. 4.4BSD does not store I/O control blocks or other operating-system-related data structures in the application's address space. Each user-level application is provided an independent address space in which it executes. The kernel makes most state changes, such as suspending a process while another is running, invisible to the processes involved.

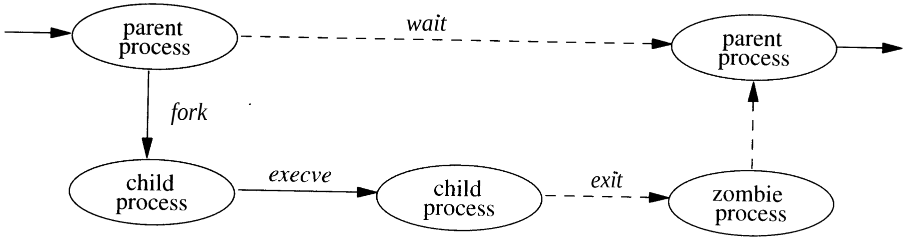
2.4 Process Management

4.4BSD supports a multitasking environment. Each task or thread of execution is termed a *process.* The *context* of a 4.4BSD process consists of user-level state, including the contents of its address space and the run-time environment, and kernel-level state, which includes scheduling parameters, resource controls, and identification information. The context includes everything used by the kernel in providing services for the process. Users can create processes, control the processes' execution, and receive notification when the processes' execution status changes. Every process is assigned a unique value, termed a *process identifier* (PID). This value is used by the kernel to identify a process when reporting status changes to a user, and by a user when referencing a process in a system call.

The kernel creates a process by duplicating the context of another process. The new process is termed a *child process* of the original *parent process.* The context duplicated in process creation includes both the user-level execution state of the process and the process's system state managed by the kernel. Important components of the kernel state are described in Chapter 4.

The process lifecycle is depicted in Fig. 2.1. A process may create a new process that is a copy of the original by using the *fork* system call. The *fork* call returns twice: once in the parent process, where the return value is the process identifier of the child, and once in the child process, where the return value is 0. The parent-child relationship induces a hierarchical structure on the set of processes in the system. The new process shares all its parent's resources, such as file descriptors, signal-handling status, and memory layout.

Figure 2.1 Process-management system calls.



Although there are occasions when the new process is intended to be a copy of the parent, the loading and execution of a different program is a more useful and typical action. A process can overlay itself with the memory image of another program, passing to the newly created image a set of parameters, using the system call *execve.* One parameter is the name of a file whose contents are in a format recognized by the system—either a binary-executable file or a file that causes the execution of a specified interpreter program to process its contents.

A process may terminate by executing an *exit* system call, sending 8 bits of exit status to its parent. If a process wants to communicate more than a single byte of information with its parent, it must either set up an interprocess-communication channel using pipes or sockets, or use an intermediate file. Interprocess communication is discussed extensively in Chapter 11.

A process can suspend execution until any of its child processes terminate using the *wait* system call, which returns the PID and exit status of the terminated child process. A parent process can arrange to be notified by a signal when a child process exits or terminates abnormally. Using the *wait4* system call, the parent can retrieve information about the event that caused termination of the child process and about resources consumed by the process during its lifetime. If a process is orphaned because its parent exits before it is finished, then the kernel arranges for the child's exit status to be passed back to a special system process **(init:** see Sections 3.1 and 14.6).

The details of how the kernel creates and destroys processes are given in Chapter 5.

Processes are scheduled for execution according to a *process-priority* parameter. This priority is managed by a kernel-based scheduling algorithm. Users can influence the scheduling of a process by specifying a parameter *(nice)* that weights the overall scheduling priority, but are still obligated to share the underlying CPU resources according to the kernel's scheduling policy.

Signals

The system defines a set of *signals* that may be delivered to a process. Signals in 4.4BSD are modeled after hardware interrupts. A process may specify a user-level subroutine to be a *handler* to which a signal should be delivered. When a signal is generated, it is blocked from further occurrence while it is being *caught* by the handler. Catching a signal involves saving the current process context and building a new one in which to run the handler. The signal is then delivered to the handler, which can either abort the process or return to the executing process (perhaps after setting a global variable). If the handler returns, the signal is unblocked and can be generated (and caught) again.

Alternatively, a process may specify that a signal is to be *ignored,* or that a default action, as determined by the kernel, is to be taken. The default action of certain signals is to terminate the process. This termination may be accompanied by creation of a *core file* that contains the current memory image of the process for use in postmortem debugging.

Some signals cannot be caught or ignored. These signals include SIGKILL, which kills runaway processes, and the job-control signal SIGSTOP.

A process may choose to have signals delivered on a special stack so that sophisticated software stack manipulations are possible. For example, a language supporting coroutines needs to provide a stack for each coroutine. The language run-time system can allocate these stacks by dividing up the single stack provided by 4.4BSD. If the kernel does not support a separate signal stack, the space allocated for each coroutine must be expanded by the amount of space required to catch a signal.

All signals have the same *priority.* If multiple signals are pending simultaneously, the order in which signals are delivered to a process is implementation specific. Signal handlers execute with the signal that caused their invocation to be blocked, but other signals may yet occur. Mechanisms are provided so that processes can protect critical sections of code against the occurrence of specified signals.

The detailed design and implementation of signals is described in Section 4.7.

Process Groups and Sessions

Processes are organized into *process groups.* Process groups are used to control access to terminals and to provide a means of distributing signals to collections of related processes. A process inherits its process group from its parent process. Mechanisms are provided by the kernel to allow a process to alter its process group or the process group of its descendents. Creating a new process group is easy; the value of a new process group is ordinarily the process identifier of the creating process.

The group of processes in a process group is sometimes referred to as a *job* and is manipulated by high-level system software, such as the shell. A common kind of job created by a shell is a *pipeline* of several processes connected by pipes, such that the output of the first process is the input of the second, the output of the second is the input of the third, and so forth. The shell creates such a job by forking a process for each stage of the pipeline, then putting all those processes into a separate process group.

A user process can send a signal to each process in a process group, as well as to a single process. A process in a specific process group may receive software interrupts affecting the group, causing the group to suspend or resume execution, or to be interrupted or terminated.

A terminal has a process-group identifier assigned to it. This identifier is normally set to the identifier of a process group associated with the terminal. A job-control shell may create a number of process groups associated with the same terminal; the terminal is the *controlling terminal* for each process in these groups. A process may read from a descriptor for its controlling terminal only if the terminal's process-group identifier matches that of the process. If the identifiers do not match, the process will be blocked if it attempts to read from the terminal. By changing the process-group identifier of the terminal, a shell can arbitrate a terminal among several different jobs. This arbitration is called *job control* and is described, with process groups, in Section 4.8.

Just as a set of related processes can be collected into a process group, a set of process groups can be collected into a *session.* The main uses for sessions are to create an isolated environment for a daemon process and its children, and to collect together a user's login shell and the jobs that that shell spawns.

2,5 Memory Management

Each process has its own private address space. The address space is initially divided into three logical segments: *text, data,* and *stack.* The text segment is read-only and contains the machine instructions of a program. The data and stack segments are both readable and writable. The data segment contains the initialized and uninitialized data portions of a program, whereas the stack segment holds the application's run-time stack. On most machines, the stack segment is extended automatically by the kernel as the process executes. A process can expand or contract its data segment by making a system call, whereas a process can change the size of its text segment only when the segment's contents are overlaid with data from the filesystem, or when debugging takes place. The initial contents of the segments of a child process are duplicates of the segments of a parent process.

The entire contents of a process address space do not need to be resident for a process to execute. If a process references a part of its address space that is not resident in main memory, the system *pages* the necessary information into memory. When system resources are scarce, the system uses a two-level approach to maintain available resources. If a modest amount of memory is available, the system will take memory resources away from processes if these resources have not been used recently. Should there be a severe resource shortage, the system will resort to *swapping* the entire context of a process to secondary storage. The *demand paging* and *swapping* done by the system are effectively transparent to processes. A process may, however, advise the system about expected future memory utilization as a performance aid.

BSD Memory-Management Design Decisions

The support of large sparse address spaces, mapped files, and shared memory was a requirement for 4.2BSD. An interface was specified, called *mmap(),* that allowed unrelated processes to request a shared mapping of a file into their address spaces. If multiple processes mapped the same file into their address spaces, changes to the file's portion of an address space by one process would be reflected in the area mapped by the other processes, as well as in the file itself. Ultimately, 4.2BSD was shipped without the *mmap()* interface, because of pressure to make other features, such as networking, available.

Further development of the *mmap()* interface continued during the work on 4.3BSD. Over 40 companies and research groups participated in the discussions leading to the revised architecture that was described in the Berkeley Software Architecture Manual [McKusick, Karels et al, 1994]. Several of the companies have implemented the revised interface [Gingell et al, 1987].

Once again, time pressure prevented 4.3BSD from providing an implementation of the interface. Although the latter could have been built into the existing 4.3BSD virtual-memory system, the developers decided not to put it in because that implementation was nearly 10 years old. Furthermore, the original virtual-memory design was based on the assumption that computer memories were small and expensive, whereas disks were locally connected, fast, large, and inexpensive. Thus, the virtual-memory system was designed to be frugal with its use of memory at the expense of generating extra disk traffic. In addition, the 4.3BSD implementation was riddled with VAX memory-management hardware dependencies that impeded its portability to other computer architectures. Finally, the virtual-memory system was not designed to support the tightly coupled multiprocessors that are becoming increasingly common and important today.

Attempts to improve the old implementation incrementally seemed doomed to failure. A completely new design, on the other hand, could take advantage of large memories, conserve disk transfers, and have the potential to run on multiprocessors. Consequently, the virtual-memory system was completely replaced in 4.4BSD. The 4.4BSD virtual-memory system is based on the Mach 2.0 VM system [Tevanian, 1987], with updates from Mach 2.5 and Mach 3.0. It features efficient support for sharing, a clean separation of machine-independent and machine-dependent features, as well as (currently unused) multiprocessor support. Processes can map files anywhere in their address space. They can share parts of their address space by doing a shared mapping of the same file. Changes made by one process are visible in the address space of the other process, and also are written back to the file itself. Processes can also request private mappings of a file, which prevents any changes that they make from being visible to other processes mapping the file or being written back to the file itself.

Another issue with the virtual-memory system is the way that information is passed into the kernel when a system call is made. 4.4BSD always copies data from the process address space into a buffer in the kernel. For read or write operations that are transferring large quantities of data, doing the copy can be time consuming. An alternative to doing the copying is to remap the process memory into the kernel. The 4.4BSD kernel always copies the data for several reasons:

- Often, the user data are not page aligned and are not a multiple of the hardware page length.
- If the page is taken away from the process, it will no longer be able to reference that page. Some programs depend on the data remaining in the buffer even after those data have been written.
- If the process is allowed to keep a copy of the page (as it is in current 4.4BSD semantics), the page must be made *copy-on-write. A* copy-on-write page is one that is protected against being written by being made read-only. If the process attempts to modify the page, the kernel gets a write fault. The kernel then makes a copy of the page that the process can modify. Unfortunately, the typical process will immediately try to write new data to its output buffer, forcing the data to be copied anyway.
- When pages are remapped to new virtual-memory addresses, most memory-management hardware requires that the hardware address-translation cache be purged selectively. The cache purges are often slow. The net effect is that remapping is slower than copying for blocks of data less than 4 to 8 Kbyte.

The biggest incentives for memory mapping are the needs for accessing big files and for passing large quantities of data between processes. The *mmapO* interface provides a way for both of these tasks to be done without copying.

Memory Management Inside the Kernel

The kernel often does allocations of memory that are needed for only the duration of a single system call. In a user process, such short-term memory would be allocated on the run-time stack. Because the kernel has a limited run-time stack, it is not feasible to allocate even moderate-sized blocks of memory on it. Consequently, such memory must be allocated through a more dynamic mechanism. For example, when the system must translate a pathname, it must allocate a 1-Kbyte buffer to hold the name. Other blocks of memory must be more persistent than a single system call, and thus could not be allocated on the stack even if there was space. An example is protocol-control blocks that remain throughout the duration of a network connection.

Demands for dynamic memory allocation in the kernel have increased as more services have been added. A generalized memory allocator reduces the complexity of writing code inside the kernel. Thus, the 4.4BSD kernel has a single memory allocator that can be used by any part of the system. It has an interface similar to the C library routines *malloc() andfree()* that provide memory allocation to application programs [McKusick & Karels, 1988]. Like the C library interface, the allocation routine takes a parameter specifying the size of memory that is needed. The range of sizes for memory requests is not constrained; however, physical memory is allocated and is not paged. The free routine takes a pointer to the storage being freed, but does not require the size of the piece of memory being freed. *

2.6 I/O System

The basic model of the UNIX I/O system is a sequence of bytes that can be accessed either randomly or sequentially. There are no *access methods* and no *control blocks* in a typical UNIX user process.

Different programs expect various levels of structure, but the kernel does not impose structure on I/O. For instance, the convention for text files is lines of ASCII characters separated by a single newline character (the ASCII line-feed character), but the kernel knows nothing about this convention. For the purposes of most programs, the model is further simplified to being a stream of data bytes, or an *I/O stream.* It is this single common data form that makes the characteristic UNIX tool-based approach work [Kernighan & Pike, 1984]. An I/O stream from one program can be fed as input to almost any other program. (This kind of traditional UNIX I/O stream should not be confused with the Eighth Edition stream I/O system or with the System V, Release 3 STREAMS, both of which can be accessed as traditional I/O streams.)

Descriptors and I/O

UNIX processes use *descriptors* to reference I/O streams. Descriptors are small unsigned integers obtained from the *open* and *socket* system calls. The *open* system call takes as arguments the name of a file and a permission mode to specify whether the file should be open for reading or for writing, or for both. This system call also can be used to create a new, empty file. A *read* or *write* system call can be applied to a descriptor to transfer data. The *close* system call can be used to deallocate any descriptor.

Descriptors represent underlying objects supported by the kernel, and are created by system calls specific to the type of object. In 4.4BSD, three kinds of objects can be represented by descriptors: files, pipes, and sockets.

- A *file* is a linear array of bytes with at least one name. A file exists until all its names are deleted explicitly and no process holds a descriptor for it. A process acquires a descriptor for a file by opening that file's name with the *open* system call. I/O devices are accessed as files.

- A *pipe* is a linear array of bytes, as is a file, but it is used solely as an I/O stream, and it is unidirectional. It also has no name, and thus cannot be opened with *open.* Instead, it is created by the *pipe* system call, which returns two descriptors, one of which accepts input that is sent to the other descriptor reliably, without duplication, and in order. The system also supports a named pipe or FIFO. A FIFO has properties identical to a pipe, except that it appears in the filesystem; thus, it can be opened using the *open* system call. Two processes that wish to communicate each open the FIFO: One opens it for reading, the other for writing.

- A *socket* is a transient object that is used for interprocess communication; it exists only as long as some process holds a descriptor referring to it. A socket is created by the *socket* system call, which returns a descriptor for it. There are different kinds of sockets that support various communication semantics, such as reliable delivery of data, preservation of message ordering, and preservation of message boundaries.

In systems before 4.2BSD, pipes were implemented using the filesystem; when sockets were introduced in 4.2BSD, pipes were reimplemented as sockets.

The kernel keeps for each process a *descriptor table,* which is a table that the kernel uses to translate the external representation of a descriptor into an internal representation. (The descriptor is merely an index into this table.) The descriptor table of a process is inherited from that process's parent, and thus access to the objects to which the descriptors refer also is inherited. The main ways that a process can obtain a descriptor are by opening or creation of an object, and by inheritance from the parent process. In addition, socket IPC allows passing of descriptors in messages between unrelated processes on the same machine.

Every valid descriptor has an associated file *offset* in bytes from the beginning of the object. Read and write operations start at this offset, which is updated after each data transfer. For objects that permit random access, the file offset also may be set with the *lseek* system call. Ordinary files permit random access, and some devices do, as well. Pipes and sockets do not.

When a process terminates, the kernel reclaims all the descriptors that were in use by that process. If the process was holding the final reference to an object, the object's manager is notified so that it can do any necessary cleanup actions, such as final deletion of a file or deallocation of a socket.

Descriptor Management

Most processes expect three descriptors to be open already when they start running. These descriptors are 0, 1, 2, more commonly known as *standard input, standard output,* and *standard error,* respectively. Usually, all three are associated with the user's terminal by the login process (see Section 14.6) and are inherited through *fork* and *exec* by processes run by the user. Thus, a 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 descriptor also is open for writing and is used for error output, whereas standard output is used for ordinary output.

These (and other) descriptors can be mapped to objects other than the terminal; such mapping is called *I/O redirection,* and all the standard shells permit users to do it. The shell can direct the output of a program to a file by closing descriptor 1 (standard output) and opening the desired output file to produce a new descriptor 1. It can similarly redirect standard input to come from a file by closing descriptor 0 and opening the file.

Pipes allow the output of one program to be input to another program without rewriting or even relinking of either program. Instead of descriptor 1 (standard output) of the source program being set up to write to the terminal, it is set up to be the input descriptor of a pipe. Similarly, descriptor 0 (standard input) of the sink program is set up to reference the output of the pipe, instead of the terminal keyboard. The resulting set of two processes and the connecting pipe is known as a *pipeline.* Pipelines can be arbitrarily long series of processes connected by pipes.

The *open, pipe,* and *socket* system calls produce new descriptors with the lowest unused number usable for a descriptor. For pipelines to work, some mechanism must be provided to map such descriptors into 0 and 1. The *dup* system call creates a copy of a descriptor that points to the same file-table entry. The new descriptor is also the lowest unused one, but if the desired descriptor is closed first, *dup* can be used to do the desired mapping. Care is required, however: If descriptor 1 is desired, and descriptor 0 happens also to have been closed, descriptor 0 will be the result. To avoid this problem, the system provides the *dup2* system call; it is like *dup,* but it takes an additional argument specifying the number of the desired descriptor (if the desired descriptor was already open, *dup2* closes it before reusing it).

Devices

Hardware devices have filenames, and may be accessed by the user via the same system calls used for regular files. The kernel can distinguish a *device special file* or *special file,* and can determine to what device it refers, but most processes do not need to make this determination. Terminals, printers, and tape drives are all accessed as though they were streams of bytes, like 4.4BSD disk files. Thus, device dependencies and peculiarities are kept in the kernel as much as possible, and even in the kernel most of them are segregated in the device drivers.

Hardware devices can be categorized as either *structured* or *unstructured;* they are known as *block* or *character* devices, respectively. Processes typically access devices through *special files* in the filesystem. I/O operations to these files are handled by kernel-resident software modules termed *device drivers.* Most network-communication hardware devices are accessible through only the interprocess-communication facilities, and do not have special files in the filesystem name space, because the *raw-socket* interface provides a more natural interface than does a special file.

Structured or block devices are typified by disks and magnetic tapes, and include most random-access devices. The kernel supports read-modify-write-type buffering actions on block-oriented structured devices to allow the latter to be read and written in a totally random byte-addressed fashion, like regular files. Filesystems are created on block devices.

Unstructured devices are those devices that do not support a block structure. Familiar unstructured devices are communication lines, raster plotters, and unbuffered magnetic tapes and disks. Unstructured devices typically support large block I/O transfers.

Unstructured files are called *character devices* because the first of these to be implemented were terminal device drivers. The kernel interface to the driver for these devices proved convenient for other devices that were not block structured.

Device special files are created by the *mknod* system call. There is an additional system call, *ioctl,* for manipulating the underlying device parameters of special files. The operations that can be done differ for each device. This system call allows the special characteristics of devices to be accessed, rather than overloading the semantics of other system calls. For example, there is an *ioctl* on a tape drive to write an end-of-tape mark, instead of there being a special or modified version of *write.*

Socket IPC

The 4.2BSD kernel introduced an IPC mechanism more flexible than pipes, based on *sockets.* A socket is an endpoint of communication referred to by a descriptor, just like a file or a pipe. Two processes can each create a socket, and then connect those two endpoints to produce a reliable byte stream. Once connected, the descriptors for the sockets can be read or written by processes, just as the latter would do with a pipe. The transparency of sockets allows the kernel to redirect the output of one process to the input of another process residing on another machine. A major difference between pipes and sockets is that pipes require a common parent process to set up the communications channel. A connection between sockets can be set up by two unrelated processes, possibly residing on different machines.

System V provides local interprocess communication through FIFOs (also known as *named pipes).* FIFOs appear as an object in the filesystem that unrelated processes can open and send data through in the same way as they would communicate through a pipe. Thus, FIFOs do not require a common parent to set them up; they can be connected after a pair of processes are up and running. Unlike sockets, FIFOs can be used on only a local machine; they cannot be used to communicate between processes on different machines. FIFOs are implemented in 4.4BSD only because they are required by the standard. Their functionality is a subset of the socket interface.

The socket mechanism requires extensions to the traditional UNIX I/O system calls to provide the associated naming and connection semantics. Rather than overloading the existing interface, the developers used the existing interfaces to the extent that the latter worked without being changed, and designed new interfaces to handle the added semantics. The *read* and *write* system calls were used for byte-stream type connections, but six new system calls were added to allow sending and receiving addressed messages such as network datagrams. The system calls for writing messages include *send, sendto,* and *sendmsg.* The system calls for reading messages include *recv, recvfrom,* and *recvmsg.* In retrospect, the first two in each class are special cases of the others; *recvfrom* and *sendto* probably should have been added as library interfaces to *recvmsg* and *sendmsg,* respectively.

Scatter/Gather I/O

In addition to the traditional *read* and *write* system calls, 4.2BSD introduced the ability to do scatter/gather I/O. Scatter input uses the *readv* system call to allow a single read to be placed in several different buffers. Conversely, the *writev* system call allows several different buffers to be written in a single atomic write. Instead of passing a single buffer and length parameter, as is done with *read* and *write,* the process passes in a pointer to an array of buffers and lengths, along with a count describing the size of the array.

This facility allows buffers in different parts of a process address space to be written atomically, without the need to copy them to a single contiguous buffer. Atomic writes are necessary in the case where the underlying abstraction is record based, such as tape drives that output a tape block on each write request. It is also convenient to be able to read a single request into several different buffers (such as a record header into one place and the data into another). Although an application can simulate the ability to scatter data by reading the data into a large buffer and then copying the pieces to their intended destinations, the cost of memory-to-memory copying in such cases often would more than double the running time of the affected application.

Just as *send* and *recv* could have been implemented as library interfaces to *sendto* and *recvfrom,* it also would have been possible to simulate *read* with *readv* and *write* with *writev.* However, *read* and *write* are used so much more frequently that the added cost of simulating them would not have been worthwhile.

Multiple Filesystem Support

With the expansion of network computing, it became desirable to support both local and remote filesystems. To simplify the support of multiple filesystems, the developers added a new virtual node or *vnode* interface to the kernel. The set of operations exported from the vnode interface appear much like the filesystem operations previously supported by the local filesystem. However, they may be supported by a wide range of filesystem types:

- Local disk-based filesystems
- Files imported using a variety of remote filesystem protocols
- Read-only CD-ROM filesystems
- Filesystems providing special-purpose interfaces—for example, the **/proc** filesystem

A few variants of 4.4BSD, such as FreeBSD, allow filesystems to be loaded dynamically when the filesystems are first referenced by the *mount* system call. The vnode interface is described in Section 6.5; its ancillary support routines are described in Section 6.6; several of the special-purpose filesystems are described in Section 6.7.

2.7 Filesystems

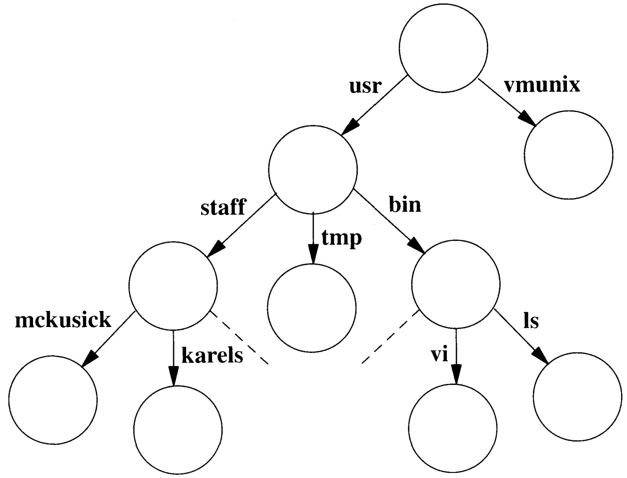
A regular file is a linear array of bytes, and can be read and written starting at any byte in the file. The kernel distinguishes no record boundaries in regular files, although many programs recognize line-feed characters as distinguishing the ends of lines, and other programs may impose other structure. No system-related information about a file is kept in the file itself, but the filesystem stores a small amount of ownership, protection, and usage information with each file.

A *filename* component is a string of up to 255 characters. These filenames are stored in a type of file called a *directory.* The information in a directory about a file is called a *directory entry* and includes, in addition to the filename, a pointer to the file itself. Directory entries may refer to other directories, as well as to plain files. A hierarchy of directories and files is thus formed, and is called a *filesystem;* a small one is shown in Fig. 2.2. Directories may contain subdirectories, and there is no inherent limitation to the depth with which directory nesting may occur. To protect the consistency of the filesystem, the kernel does not permit processes to write directly into directories. A filesystem may include not only plain files and directories, but also references to other objects, such as devices and sockets.

The filesystem forms a tree, the beginning of which is the *root directory,* sometimes referred to by the name **slash,** spelled with a single solidus character (/). The root directory contains files; in our example in Fig. 2.2, it contains **vmunix**, a copy of the kernel-executable object file. It also contains directories; in this example, it contains the **usr** directory. Within the **usr** directory is the bin directory, which mostly contains executable object code of programs, such as the files Is and vi.

A process identifies a file by specifying that file's *pathname,* which is a string composed of zero or more filenames separated by slash (/) characters. The kernel associates two directories with each process for use in interpreting pathnames. A process's *root directory* is the topmost point in the filesystem that the process can access; it is ordinarily set to the root directory of the entire filesystem. A pathname beginning with a slash is called an *absolute pathname,* and is interpreted by the kernel starting with the process's root directory.

Figure 2.2 A small filesystem tree.



A pathname that does not begin with a slash is called a *relative pathname*, and is interpreted relative to the *current working directory* of the process. (This directory also is known by the shorter names *current directory* or *working directory.}* The current directory itself may be referred to directly by the name *dot,* spelled with a single period (.). The filename *dot-dot* (..) refers to a directory's parent directory. The root directory is its own parent.

A process may set its root directory with the *chroot* system call, and its current directory with the *chdir* system call. Any process may do *chdir* at any time, but *chroot* is permitted only a process with superuser privileges. *Chroot* is normally used to set up restricted access to the system.

Using the filesystem shown in Fig. 2.2, if a process has the root of the filesystem as its root directory, and has /usr as its current directory, it can refer to the file vi either from the root with the absolute pathname **/usr/bin/vi,** or from its current directory with the relative pathname **bin/vi.**

System utilities and databases are kept in certain well-known directories. Part of the well-defined hierarchy includes a directory that contains the *home directory* for each user—for example, **/usr/staff/mckusick and /usr/staff/karels** in Fig. 2.2. When users log in, the current working directory of their shell is set to the home directory. Within their home directories, users can create directories as easily as they can regular files. Thus, a user can build arbitrarily complex subhierarchies.

The user usually knows of only one filesystem, but the system may know that this one virtual filesystem is really composed of several physical filesystems, each on a different device. A physical filesystem may not span multiple hardware devices. Since most physical disk devices are divided into several logical devices, there may be more than one filesystem per physical device, but there will be no more than one per logical device. One filesystem—the filesystem that anchors all absolute pathnames—is called the *root filesystem*, and is always available. Others may be mounted; that is, they may be integrated into the directory hierarchy of the root filesystem. References to a directory that has a filesystem mounted on it are converted transparently by the kernel into references to the root directory of the mounted filesystem.

The *link* system call takes the name of an existing file and another name to create for that file. After a successful *link,* the file can be accessed by either filename. A filename can be removed with the *unlink* system call. When the final name for a file is removed (and the final process that has the file open closes it), the file is deleted.

Files are organized hierarchically in *directories.* A directory is a type of file, but, in contrast to regular files, a directory has a structure imposed on it by the system. A process can read a directory as it would an ordinary file, but only the kernel is permitted to modify a directory. Directories are created by the *mkdir* system call and are removed by the *rmdir* system call. Before 4.2BSD, the *mkdir* and *rmdir* system calls were implemented by a series of *link* and *unlink* system calls being done. There were three reasons for adding systems calls explicitly to create and delete directories:

1. The operation could be made atomic. If the system crashed, the directory would not be left half-constructed, as could happen when a series of link operations were used.

2. When a networked filesystem is being run, the creation and deletion of files and directories need to be specified atomically so that they can be serialized.

3. When supporting non-UNIX filesystems, such as an MS-DOS filesystem, on another partition of the disk, the other filesystem may not support link operations. Although other filesystems might support the concept of directories, they probably would not create and delete the directories with links, as the UNIX filesystem does. Consequently, they could create and delete directories only if explicit directory create and delete requests were presented.

The *chown* system call sets the owner and group of a file, and *chmod* changes protection attributes. *Stat* applied to a filename can be used to read back such properties of a file. The *fchown, fchmod,* *a* system calls are applied to a descriptor, instead of to a filename, to do the same set of operations. The *rename* system call can be used to give a file a new name in the filesystem, replacing one of the file's old names. Like the directory-creation and directory-deletion operations, the *rename* system call was added to 4.2BSD to provide atomicity to name changes in the local filesystem. Later, it proved useful explicitly to export renaming operations to foreign filesystems and over the network.

The *truncate* system call was added to 4.2BSD to allow files to be shortened to an arbitrary offset. The call was added primarily in support of the Fortran runtime library, which has the semantics such that the end of a random-access file is set to be wherever the program most recently accessed that file. Without the *truncate* system call, the only way to shorten a file was to copy the part that was desired to a new file, to delete the old file, then to rename the copy to the original name. As well as this algorithm being slow, the library could potentially fail on a full filesystem.

Once the filesystem had the ability to shorten files, the kernel took advantage of that ability to shorten large empty directories. The advantage of shortening empty directories is that it reduces the time spent in the kernel searching them when names are being created or deleted.

Newly created files are assigned the user identifier of the process that created them and the group identifier of the directory in which they were created. A three-level access-control mechanism is provided for the protection of files. These three levels specify the accessibility of a file to

1. The user who owns the file

2. The group that owns the file

3. Everyone else

Each level of access has separate indicators for read permission, write permission, and execute permission.

Files are created with zero length, and may grow'when they are written. While a file is open, the system maintains a pointer into the file indicating the current location in the file associated with the descriptor. This pointer can be moved about in the file in a random-access fashion. Processes sharing a file descriptor through a *fork* or *dup* system call share the current location pointer. Descriptors created by separate *open* system calls have separate current location pointers. Files may have *holes* in them. Holes are void areas in the linear extent of the file where data have never been written. A process can create these holes by positioning the pointer past the current end-of-file and writing. When read, holes are treated by the system as zero-valued bytes.

Earlier UNIX systems had a limit of 14 characters per filename component. This limitation was often a problem. For example, in addition to the natural desire of users to give files long descriptive names, a common way of forming filenames is as **basename.extension,** where the extension (indicating the kind of file, such as .c for C source or .o for intermediate binary object) is one to three characters, leaving 10 to 12 characters for the basename. Source-code-control systems and editors usually take up another two characters, either as a prefix or a suffix, for their purposes, leaving eight to 10 characters. It is easy to use 10 or 12 characters in a single English word as a basename (e.g., "multiplexer").

It is possible to keep within these limits, but it is inconvenient or even dangerous, because other UNIX systems accept strings longer than the limit when creating files, but then *truncate* to the limit. A C language source file named **multiplexer.c** (already 13 characters) might have a source-code-control file with s. prepended, producing a filename **s.multiplexer** that is indistinguishable from the source-code-control file for **multiplexer.ms,** a file containing troff source for documentation for the C program. The contents of the two original files could easily get confused with no warning from the source-code-control system. Careful coding can detect this problem, but the long filenames first introduced in 4.2BSD practically eliminate it.

2.8 Filestores

The operations defined for local filesystems are divided into two parts. Common to all local filesystems are hierarchical naming, locking, quotas, attribute management, and protection. These features are independent of how the data will be stored. 4.4BSD has a single implementation to provide these semantics.

The other part of the local filesystem is the organization and management of the data on the storage media. Laying out the contents of files on the storage media is the responsibility of the filestore. 4.4BSD supports three different filestore layouts:

- The traditional Berkeley Fast Filesystem
- The log-structured filesystem, based on the Sprite operating-system design [Rosenblum & Ousterhout, 1992]
- A memory-based filesystem

Although the organizations of these filestores are completely different, these differences are indistinguishable to the processes using the filestores.

The Fast Filesystem organizes data into cylinder groups. Files that are likely to be accessed together, based on their locations in the filesystem hierarchy, are stored in the same cylinder group. Files that are not expected to accessed together are moved into different cylinder groups. Thus, files written at the same time may be placed far apart on the disk.

The log-structured filesystem organizes data as a log. All data being written at any point in time are gathered together, and are written at the same disk location. Data are never overwritten; instead, a new copy of the file is written that replaces the old one. The old files are reclaimed by a garbage-collection process that runs when the filesystem becomes full and additional free space is needed.

The memory-based filesystem is designed to store data in virtual memory. It is used for filesystems that need to support fast but temporary data, such as /tmp. The goal of the memory-based filesystem is to keep the storage packed as compactly as possible to minimize the usage of virtual-memory resources.

2.9 Network Filesystem

Initially, networking was used to transfer data from one machine to another. Later, it evolved to allowing users to log in remotely to another machine. The next logical step was to bring the data to the user, instead of having the user go to the data—and network filesystems were born. Users working locally do not experience the network delays on each keystroke, so they have a more responsive environment.

Bringing the filesystem to a local machine was among the first of the major client-server applications. The *server* is the remote machine that exports one or more of its filesystems. The *client* is the local machine that imports those filesystems. From the local client's point of view, a remotely mounted filesystem appears in the file-tree name space just like any other locally mounted filesystem. Local clients can change into directories on the remote filesystem, and can read, write, and execute binaries within that remote filesystem identically to the way that they can do these operations on a local filesystem.

When the local client does an operation on a remote filesystem, the request is packaged and is sent to the server. The server does the requested operation and returns either the requested information or an error indicating why the request was stream-type sockets. A new interface was added for more complicated sockets, such as those used to send datagrams, with which a destination address must be presented with each *send* call.

Another benefit is that the new interface is highly portable. Shortly after a test release was available from Berkeley, the socket interface had been ported to System III by a UNIX vendor (although AT&T did not support the socket interface until the release of System V Release 4, deciding instead to use the Eighth Edition stream mechanism). The socket interface was also ported to run in many Ethernet boards by vendors, such as Excelan and Interlan, that were selling into the PC market, where the machines were too small to run networking in the main processor. More recently, the socket interface was used as the basis for Microsoft's Winsock networking interface for Windows.

2.12 Network Communication

Some of the communication domains supported by the *socket* IPC mechanism provide access to network protocols. These protocols are implemented as a separate software layer logically below the socket software in the kernel. The kernel provides many ancillary services, such as buffer management, message routing, standardized interfaces to the protocols, and interfaces to the network interface drivers for the use of the various network protocols.

At the time that 4.2BSD was being implemented, there were many networking protocols in use or under development, each with its own strengths and weaknesses. There was no clearly superior protocol or protocol suite. By supporting multiple protocols, 4.2BSD could provide interoperability and resource sharing among the diverse set of machines that was available in the Berkeley environment. Multiple-protocol support also provides for future changes. Today's protocols designed for 10- to 100-Mbit-per-second Ethernets are likely to be inadequate for tomorrow's 1- to 10-Gbit-per-second fiber-optic networks. Consequently, the network-communication layer is designed to support multiple protocols. New protocols are added to the kernel without the support for older protocols being affected. Older applications can continue to operate using the old protocol over the same physical network as is used by newer applications running with a newer network protocol.

2.13 Network Implementation

The first protocol suite implemented in 4.2BSD was DARPA's Transmission Control Protocol/Internet Protocol (TCP/IP). The CSRG chose TCP/IP as the first network to incorporate into the socket IPC framework, because a 4.1BSD-based implementation was publicly available from a DARPA-sponsored project at Bolt, Beranek, and Newman (BBN). That was an influential choice: The 4.2BSD implementation is the main reason for the extremely widespread use of this protocol suite. Later performance and capability improvements to the TCP/IP implementation have also been widely adopted. The TCP/IP implementation is described in detail in Chapter 13.

The release of 4.3BSD added the Xerox Network Systems (XNS) protocol suite, partly building on work done at the University of Maryland and at Cornell University. This suite was needed to connect isolated machines that could not communicate using TCP/IP.

The release of 4.4BSD added the ISO protocol suite because of the latter's increasing visibility both within and outside the United States. Because of the somewhat different semantics defined for the ISO protocols, some minor changes were required in the socket interface to accommodate these semantics. The changes were made such that they were invisible to clients of other existing protocols. The ISO protocols also required extensive addition to the two-level routing tables provided by the kernel in 4.3BSD. The greatly expanded routing capabilities of 4.4BSD include arbitrary levels of routing with variable-length addresses and network masks.

2.14 System Operation

Bootstrapping mechanisms are used to start the system running. First, the 4.4BSD kernel must be loaded into the main memory of the processor. Once loaded, it must go through an initialization phase to set the hardware into a known state. Next, the kernel must do autoconfiguration, a process that finds and configures the peripherals that are attached to the processor. The system begins running in single-user mode while a start-up script does disk checks and starts the accounting and quota checking. Finally, the start-up script starts the general system services and brings up the system to full multiuser operation.

During multiuser operation, processes wait for login requests on the terminal lines and network ports that have been configured for user access. When a login request is detected, a login process is spawned and user validation is done. When the login validation is successful, a login shell is created from which the user can run additional processes.

Exercises

2.1 How does a user process request a service from the kernel?

2.2 How are data transferred between a process and the kernel? What alternatives are available?

2.3 How does a process access an I/O stream? List three types of I/O streams.

2.4 What are the four steps in the lifecycle of a process?

2.5 Why are process groups provided in 4.3BSD?

2.6 Describe four machine-dependent functions of the kernel?

2.7 Describe the difference between an absolute and a relative pathname.

2.8 Give three reasons why the *mkdir* system call was added to 4.2BSD.

2.9 Define *scatter-gather I/O.* Why is it useful?

2.10 What is the difference between a block and a character device?

2.11 List five functions provided by a terminal driver.

2.12 What is the difference between a pipe and a socket?

2.13 Describe how to create a group of processes in a pipeline.

*2.14 List the three system calls that were required to create a new directory **foo** in the current directory before the addition of the *mkdir* system call.

*2.15 Explain the difference between interprocess communication and networking.

References

Accetta etal, 1986.
M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, & M. Young, "Mach: A New Kernel Foundation for UNIX Development," *USENIX Association Conference Proceedings,* pp. 93-113, June 1986.

Cheriton, 1988.
D. R. Cheriton, "The V Distributed System," *Comm ACM,* vol. 31, no. 3, pp. 314-333, March 1988.

Ewens etal, 1985.
P. Ewens, D. R. Blythe, M. Funkenhauser, & R. C. Holt, "Tunis: A Distributed Multiprocessor Operating System," *USENIX Association Conference Proceedings,* pp. 247-254, June 1985.

Gingelletal, 1987.
R. Gingell, J. Moran, & W. Shannon, "Virtual Memory Architecture in SunOS," *USENIX Association Conference Proceedings,* pp. 81-94, June 1987.

Kernighan & Pike, 1984.
B. W. Kernighan & R. Pike, *The UNIX Programming Environment,* Prentice-Hall, Englewood Cliffs, NJ, 1984.

Macklem, 1994.
R. Macklem, "The 4.4BSD NFS Implementation," in *4.4BSD System Manager's Manual,* pp. 6:1-14, O'Reilly & Associates, Inc., Sebastopol, CA, 1994.

McKusick & Karels, 1988.
M. K. McKusick & M. J. Karels, "Design of a General Purpose Memory

Allocator for the 4.3BSD UNIX Kernel," *USENIX Association Conference Proceedings*, pp. 295-304, June 1988.

McKusick, Karels et al, 1994.
M. K. McKusick, M. J. Karels, S. J. Leffler, W. N. Joy, & R. S. Fabry, "Berkeley Software Architecture Manual, 4.4BSD Edition," in *4.4BSD Programmer's Supplementary Documents*, pp. 5:1-42, O'Reilly & Associates, Inc., Sebastopol, CA, 1994.

Ritchie, 1988
D. M. Ritchie, "Early Kernel Design," private communication, March 1988.

Rosenblum & Ousterhout, 1992.
M Rosenblum & J. Ousterhout, "The Design and Implementation of a Log-Structured File System," *ACM Transactions on Computer Systems, vol.* 10, no. 1, pp. 26-52, Association for Computing Machinery, February 1992.

M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrman, C. Kaiser, S. Langlois, P. Leonard, & W. Neuhauser Chorus Distributed Operating Systems," *USENIX Computing Systems,* vol. 1, no. 4, pp. 305-370, Fall 1988.

Tevanian ,1987. Memory Management for Parallel and Distributed Environments: The Mach Approach, Technical Report CMU-CS-88-106, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, December 1987.

CHAPTER 3

Kernel Services

3.1 Kernel Organization

The 4.4BSD kernel can be viewed as a service provider to user processes. Processes usually access these services through system calls. Some services, such as process scheduling and memory management, are implemented as processes that execute in kernel mode or as routines that execute periodically within the kernel. In this chapter, we describe how kernel services are provided to user processes, and what some of the ancillary processing performed by the kernel is. Then, we describe the basic kernel services provided by 4.4BSD, and provide details of their implementation.

System Processes

All 4.4BSD processes originate from a single process that is crafted by the kernel at startup. Three processes are created immediately and exist always. Two of them are *kernel processes,* and function wholly within the kernel. (Kernel processes execute code that is compiled into the kernel's load image and operate with the kernel's privileged execution mode.) The third is the first process to execute a program in user mode; it serves as the parent process for all subsequent processes.

The two kernel processes are the *swapper* and the *pagedaemon.* The *swapper*—historically, process 0—is responsible for scheduling the transfer of whole processes between main memory and secondary storage when system resources are low. The *pagedaemon*—historically, process 2—is responsible for writing parts of the address space of a process to secondary storage in support of the paging facilities of the virtual-memory system. The third process is the **init** process—historically, process 1. This process performs administrative tasks, such as spawning getty processes for each terminal on a machine and handling the orderly shutdown of a system from multiuser to single-user operation. The **init** process is a user-mode process, running outside the kernel (see Section 14.6).

System Entry

Entrances into the kernel can be categorized according to the event or action that initiates it;

- Hardware interrupt
- Hardware trap
- Software-initiated trap

Hardware interrupts arise from external events, such as an I/O device needing attention or a clock reporting the passage of time. (For example, the kernel depends on the presence of a real-time clock or interval timer to maintain the current time of day, to drive process scheduling, and to initiate the execution of system timeout functions.) Hardware interrupts occur *asynchronously* and may not relate to the context of the currently executing process.

Hardware traps may be either synchronous or asynchronous, but *are* related to the current executing process. Examples of hardware traps are those generated as a result of an illegal arithmetic operation, such as divide by zero.

Software-initiated traps are used by the system to force the scheduling of an event such as process rescheduling or network processing, as soon as is possible. For most uses of software-initiated traps, it is an implementation detail whether they are implemented as a hardware-generated interrupt, or as a flag that is checked whenever the priority level drops (e.g., on every exit from the kernel). An example of hardware support for software-initiated traps is the *asynchronous system trap (AST)* provided by the VAX architecture. An AST is posted by the kernel. Then, when a return-from-interrupt instruction drops the interrupt-priority level below a threshold, an AST interrupt will be delivered. Most architectures today do not have hardware support for ASTs, so they must implement ASTs in software.

System calls are a special case of a software-initiated trap—the machine instruction used to initiate a system call typically causes a hardware trap that is handled specially by the kernel.

Run-Time Organization

The kernel can be logically divided into a *top half* and a *bottom half,* as shown in Fig. 3.1. The top half of the kernel provides services to processes in response to system calls or traps. This software can be thought of as a library of routines shared by all processes. The top half of the kernel executes in a privileged execution mode, in which it has access both to kernel data structures and to the context of user-level processes. The context of each process is contained in two areas of memory reserved for process-specific information. The first of these areas is the *process structure,* which has historically contained the information that is necessary even if the process has been swapped out. In 4.4BSD, this information includes the identifiers associated with the process, the process's rights and privileges, its descriptors, its memory map, pending external events and associated actions, maximum and current resource utilization, and many other things. The second is the *user structure,* which has historically contained the information that is not necessary when the process is swapped out. In 4.4BSD, the user-structure information of each process includes the hardware process control block (PCB), process accounting and statistics, and minor additional information for debugging and creating a core dump. Deciding what was to be stored in the *process structure* and the *user structure* was far more important in previous systems than it was in 4.4BSD. As memory became a less limited resource, most of the user structure was merged into the process structure for convenience; see Section 4.2.

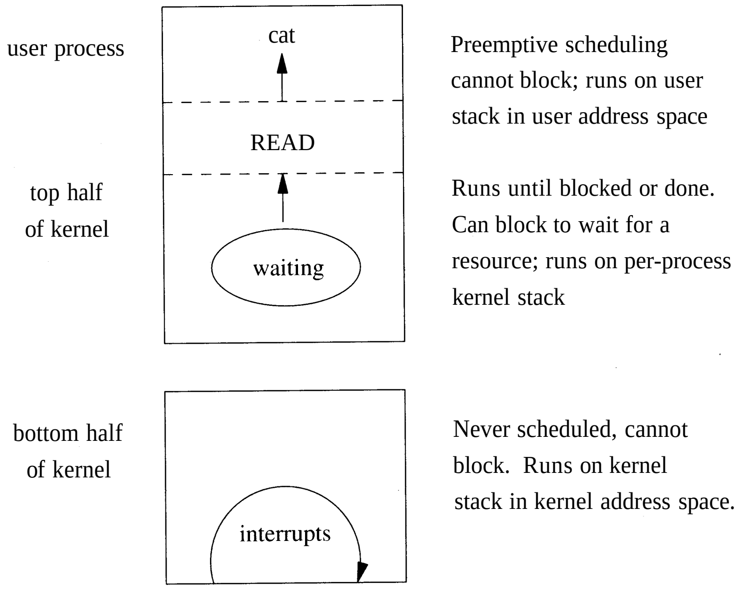

user process
cat
Preemptive scheduling cannot block; runs on user stack in user address space
READ
top half of kernel
waiting
Runs until blocked or done. Can block to wait for a resource; runs on per-process kernel stack
bottom half of kernel
interrupts
Never scheduled, cannot block. Runs on kernel stack in kernel address space.

Figure 3.1 Run-time structure of the kernel.

The bottom half of the kernel comprises routines that are invoked to handle hardware interrupts. The kernel requires that hardware facilities be available to block the delivery of interrupts. Improved performance is available if the hardware facilities allow interrupts to be defined in order of priority. Whereas the HP300 provides distinct hardware priority levels for different kinds of interrupts, UNIX also runs on architectures such as the Perkin Elmer, where interrupts are all at the same priority, or the ELXSI, where there are no interrupts in the traditional sense.

Activities in the bottom half of the kernel are *asynchronous,* with respect to the top half, and the software cannot depend on having a specific (or any) process running when an interrupt occurs. Thus, the state information for the process that initiated the activity is not available. (Activities in the bottom half of the kernel are synchronous with respect to the interrupt source.) The top and bottom halves of the kernel communicate through data structures, generally organized around work queues.

The 4.4BSD kernel is never preempted to run another process while executing in the top half of the kernel—for example, while executing a system call—although it will explicitly give up the processor if it must wait for an event or for a shared resource. Its execution may be interrupted, however, by interrupts for the bottom half of the kernel. The bottom half always begins running at a specific *priority level.* Therefore, the top half can block these interrupts by setting the *processor priority level* to an appropriate value. The value is chosen based on the priority level of the device that shares the data structures that the top half is about to modify. This mechanism ensures the consistency of the work queues and other data structures shared between the top and bottom halves.

Processes cooperate in the sharing of system resources, such as the CPU. The top and bottom halves of the kernel also work together in implementing certain system operations, such as I/O. Typically, the top half will start an I/O operation, then relinquish the processor; then the requesting process will sleep, awaiting notification from the bottom half that the I/O request has completed.

Entry to the Kernel

When a process enters the kernel through a trap or an interrupt, the kernel must save the current machine state before it begins to service the event. For the HP300, the machine state that must be saved includes the program counter, the user stack pointer, the general-purpose registers and the processor status longword. The HP300 trap instruction saves the program counter and the processor status longword as part of the exception stack frame; the user stack pointer and registers must be saved by the software trap handler. If the machine state were not fully saved, the kernel could change values in the currently executing program in improper ways. Since interrupts may occur between any two user-level instructions (and, on some architectures, between parts of a single instruction), and because they may be completely unrelated to the currently executing process, an incompletely saved state could cause correct programs to fail in mysterious and not easily reproduceable ways.

The exact sequence of events required to save the process state is completely machine dependent, although the HP300 provides a good example of the general procedure. A trap or system call will trigger the following events:

- The hardware switches into kernel (supervisor) mode, so that memory-access checks are made with kernel privileges, references to the stack pointer use the kernel's stack pointer, and privileged instructions can be executed.
- The hardware pushes onto the per-process kernel stack the program counter, processor status longword, and information describing the type of trap. (On architectures other than the HP300, this information can include the system-call number and general-purpose registers as well.)
- An assembly-language routine saves all state information not saved by the hardware. On the HP300, this information includes the general-purpose registers and the user stack pointer, also saved onto the per-process kernel stack.

After this preliminary state saving, the kernel calls a C routine that can freely use the general-purpose registers as any other C routine would, without concern about changing the unsuspecting process's state.

There are three major kinds of handlers, corresponding to particular kernel entries:

1. *Syscall* () for a system call
2. *Trap* () for hardware traps and for software-initiated traps other than system calls
3. The appropriate device-driver interrupt handler for a hardware interrupt

Each type of handler takes its own specific set of parameters. For a system call, they are the system-call number and an exception frame. For a trap, they are the type of trap, the relevant floating-point and virtual-address information related to the trap, and an exception frame. (The exception-frame arguments for the trap and system call are not the same. The HP300 hardware saves different information based on different types of traps.) For a hardware interrupt, the only parameter is a unit (or board) number.

Return from the Kernel

When the handling of the system entry is completed, the user-process state is restored, and the kernel returns to the user process. Returning to the user process reverses the process of entering the kernel.

- An assembly-language routine restores the general-purpose registers and user-stack pointer previously pushed onto the stack.
- The hardware restores the program counter and program status longword, and switches to user mode, so that future references to the stack pointer use the user's stack pointer, privileged instructions cannot be executed, and memory-access checks are done with user-level privileges.

Execution then resumes at the next instruction in the user's process.

3.2 System Calls

The most frequent trap into the kernel (after clock processing) is a request to do a system call. System performance requires that the kernel minimize the overhead in fielding and dispatching a system call. The system-call handler must do the following work:

- Verify that the parameters to the system call are located at a valid user address, and copy them from the user's address space into the kernel
- Call a kernel routine that implements the system call

Result Handling

Eventually, the system call returns to the calling process, either successfully or unsuccessfully. On the HP300 architecture, success or failure is returned as the carry bit in the user process's program status longword: If it is zero, the return was successful; otherwise, it was unsuccessful. On the HP300 and many other machines, return values of C functions are passed back through a general-purpose register (for the HP300, data register 0). The routines in the kernel that implement system calls return the values that are normally associated with the global variable *errno*. After a system call, the kernel system-call handler leaves this value in the register. If the system call failed, a C library routine moves that value into *errno*, and sets the return register to -1. The calling process is expected to notice the value of the return register, and then to examine *errno*. The mechanism involving the carry bit and the global variable *errno* exists for historical reasons derived from the PDP-11.

There are two kinds of unsuccessful returns from a system call: those where kernel routines discover an error, and those where a system call is interrupted. The most common case is a system call that is interrupted when it has relinquished the processor to wait for an event that may not occur for a long time (such as terminal input), and a signal arrives in the interim. When signal handlers are initialized by a process, they specify whether system calls that they interrupt should be restarted, or whether the system call should return with an *interrupted system call* (EINTR) error.

When a system call is interrupted, the signal is delivered to the process. If the process has requested that the signal abort the system call, the handler then returns an error, as described previously. If the system call is to be restarted, however, the handler resets the process's program counter to the machine instruction that caused the system-call trap into the kernel. (This calculation is necessary because the program-counter value that was saved when the system-call trap was done is for the instruction after the trap-causing instruction.) The handler replaces the saved program-counter value with this address. When the process returns from the signal handler, it resumes at the program-counter value that the handler provided, and reexecutes the same system call.

Restarting a system call by resetting the program counter has certain implications. First, the kernel must not modify any of the input parameters in the process address space (it can modify the kernel copy of the parameters that it makes). Second, it must ensure that the system call has not performed any actions that cannot be repeated. For example, in the current system, if any characters have been read from the terminal, the read must return with a short count. Otherwise, if the call were to be restarted, the already-read bytes would be lost.

Returning from a System Call

While the system call is running, a signal may be posted to the process, or another process may attain a higher scheduling priority. After the system call completes, the handler checks to see whether either event has occurred.

The handler first checks for a posted signal. Such signals include signals that interrupted the system call, as well as signals that arrived while a system call was in progress, but were held pending until the system call completed. Signals that are ignored, by default or by explicit programmatic request, are never posted to the process. Signals with a default action have that action taken before the process runs again (i.e., the process may be stopped or terminated as appropriate). If a signal is to be caught (and is not currently blocked), the handler arranges to have the appropriate signal handler called, rather than to have the process return directly from the system call. After the handler returns, the process will resume execution at system-call return (or system-call execution, if the system call is being restarted).

After checking for posted signals, the handler checks to see whether any process has a priority higher than that of the currently running one. If such a process exists, the handler calls the context-switch routine to cause the higher-priority process to run. At a later time, the current process will again have the highest priority, and will resume execution by returning from the system call to the user process.

If a process has requested that the system do profiling, the handler also calculates the amount of time that has been spent in the system call, i.e., the system time accounted to the process between the latter's entry into and exit from the handler. This time is charged to the routine in the user's process that made the system call.

3.3 Traps and Interrupts

Traps

Traps, like system calls, occur synchronously for a process. Traps normally occur because of unintentional errors, such as division by zero or indirection through an invalid pointer. The process becomes aware-of the problem either by catching a signal or by being terminated. Traps can also occur because of a page fault, in which case the system makes the page available and restarts the process without the process being aware that the fault occurred.

The trap handler is invoked like the system-call handler. First, the process state is saved. Next, the trap handler determines the trap type, then arranges to post a signal or to cause a pagein as appropriate. Finally, it checks for pending signals and higher-priority processes, and exits identically to the system-call handler.

I/O Device Interrupts

Interrupts from I/O and other devices are handled by interrupt routines that are loaded as part of the kernel's address space. These routines handle the console terminal interface, one or more clocks, and several software-initiated interrupts used by the system for low-priority clock processing and for networking facilities.

Unlike traps and system calls, device interrupts occur asynchronously. The process that requested the service is unlikely to be the currently running process, and may no longer exist! The process that started the operation will be notified that the operation has finished when that process runs again. As occurs with traps and system calls, the entire machine state must be saved, since any changes could cause errors in the currently running process.

Device-interrupt handlers run only on demand, and are never scheduled by the kernel. Unlike system calls, interrupt handlers do not have a per-process context. Interrupt handlers cannot use any of the context of the currently running process (e.g., the process's user structure). The stack normally used by the kernel is part of a process context. On some systems (e.g., the HP300), the interrupts are caught on the per-process kernel stack of whichever process happens to be running. This approach requires that all the per-process kernel stacks be large enough to handle the deepest possible nesting caused by a system call and one or more interrupts, and that a per-process kernel stack always be available, even when a process is not running. Other architectures (e.g., the VAX), provide a systemwide interrupt stack that is used solely for device interrupts. This architecture allows the per-process kernel stacks to be sized based on only the requirements for handling a synchronous trap or system call. Regardless of the implementation, when an interrupt occurs, the system must switch to the correct stack (either explicitly, or as part of the hardware exception handling) before it begins to handle the interrupt.

The interrupt handler can never use the stack to save state between invocations. An interrupt handler must get all the information that it needs from the data structures that it shares with the top half of the kernel—generally, its global work queue. Similarly, all information provided to the top half of the kernel by the interrupt handler must be communicated the same way. In addition, because 4.4BSD requires a per-process context for a thread of control to sleep, an interrupt handler cannot relinquish the processor to wait for resources, but rather must always run to completion.

Software Interrupts

Many events in the kernel are driven by hardware interrupts. For high-speed devices such as network controllers, these interrupts occur at a high priority. A network controller must quickly acknowledge receipt of a packet and reenable the controller to accept more packets to avoid losing closely spaced packets. However, the further processing of passing the packet to the receiving process, although time consuming, does not need to be done quickly. Thus, a lower priority is possible for the further processing, so critical operations will not be blocked from executing longer than necessary.

The mechanism for doing lower-priority processing is called a *software interrupt.* Typically, a high-priority interrupt creates a queue of work to be done at a lower-priority level. After queueing of the work request, the high-priority interrupt arranges for the processing of the request to be run at a lower-priority level. When the machine priority drops below that lower priority, an interrupt is generated that calls the requested function. If a higher-priority interrupt comes in during request processing, that processing will be preempted like any other low-priority task. On some architectures, the interrupts are true hardware traps caused by software instructions. Other architectures implement the same functionality by monitoring flags set by the interrupt handler at appropriate times and calling the request-processing functions directly.

The delivery of network packets to destination processes is handled by a packet-processing function that runs at low priority. As packets come in, they are put onto a work queue, and the controller is immediately reenabled. Between packet arrivals, the packet-processing function works to deliver the packets. Thus, the controller can accept new packets without having to wait for the previous packet to be delivered. In addition to network processing, software interrupts are used to handle time-related events and process rescheduling.

3.4 Clock Interrupts

The system is driven by a clock that interrupts at regular intervals. Each interrupt is referred to as a *tick.* On the HP300, the clock ticks 100 times per second. At each tick, the system updates the current time of day as well as user-process and system timers.

Interrupts for clock ticks are posted at a high hardware-interrupt priority. After the process state has been saved, the *hardclock()* routine is called. It is important that the *hardclock()* routine finish its job quickly:

- If *hardclock()* runs for more than one tick, it will miss the next clock interrupt. Since *hardclock()* maintains the time of day for the system, a missed interrupt will cause the system to lose time.
- Because of *hardclock()s* high interrupt priority, nearly all other activity in the system is blocked while *hardclock()* is running. This blocking can cause network controllers to miss packets, or a disk controller to miss the transfer of a sector coming under a disk drive's head.

So that the time spent in *hardclock()* is minimized, less critical time-related processing is handled by a lower-priority software-interrupt handler called *softclock().* In addition, if multiple clocks are available, some time-related processing can be handled by other routines supported by alternate clocks.

The work done by *hardclock()* is as follows:

- Increment the current time of day.
- If the currently running process has a virtual or profiling interval timer (see Section 3.6), decrement the timer and deliver a signal if the timer has expired.
- If the system does not have a separate clock for statistics gathering, the *hardclock()* routine does the operations normally done by *statclock(),* as described in the next section.

- If *softclock()* needs to be called, and the current interrupt-priority level is low, call *softclock()* directly.

Statistics and Process Scheduling

On historic 4BSD systems, the *hardclock( )* routine collected resource-utilization statistics about what was happening when the clock interrupted. These statistics were used to do accounting, to monitor what the system was doing, and to determine future scheduling priorities. In addition, *hardclock( )* forced context switches so that all processes would get a share of the CPU.

This approach has weaknesses because the clock supporting *hardclock( )* interrupts on a regular basis. Processes can become synchronized with the system clock, resulting in inaccurate measurements of resource utilization (especially CPU) and inaccurate profiling [McCanne & Torek, 1993]. It is also possible to write programs that deliberately synchronize with the system clock to outwit the scheduler.

On architectures with multiple high-precision, programmable clocks, such as the HP300, randomizing the interrupt period of a clock can improve the system resource-usage measurements significantly. One clock is set to interrupt at a fixed rate; the other interrupts at a random interval chosen from times distributed uniformly over a bounded range.

To allow the collection of more accurate profiling information, 4.4BSD supports profiling clocks. When a profiling clock is available, it is set to run at a tick rate that is relatively prime to the main system clock (five times as often as the system clock, on the HP300).

The *statclock( )* routine is supported by a separate clock if one is available, and is responsible for accumulating resource usage to processes. The work done by *statclock()* includes

- Charge the currently running process with a tick; if the process has accumulated four ticks, recalculate its priority. If the new priority is less than the current priority, arrange for the process to be rescheduled.
- Collect statistics on what the system was doing at the time of the tick (sitting idle, executing in user mode, or executing in system mode). Include basic information on system I/O, such as which disk drives are currently active.

Timeouts

The remaining time-related processing involves processing timeout requests and periodically reprioritizing processes that are ready to run. These functions are handled by the *softclockO* routine.

When *hardclockO* completes, if there were any *softclockO* functions to be done, *hardclock( )* schedules a softclock interrupt, or sets a flag that will cause *softclock( )* to be called. As an optimization, if the state of the processor is such that the *softclock( )* execution will occur as soon as the hardclock interrupt returns, *hardclock( )* simply lowers the processor priority and calls *softclock( )* directly, avoiding the cost of returning from one interrupt only to reenter another. The savings can be substantial over time, because interrupts are expensive and these interrupts occur so frequently.

The primary task of the *softclock( )* routine is to arrange for the execution of periodic events, such as

- Process real-time timer (see Section 3.6)
- Retransmission of dropped network packets
- Watchdog timers on peripherals that require monitoring
- System process-rescheduling events

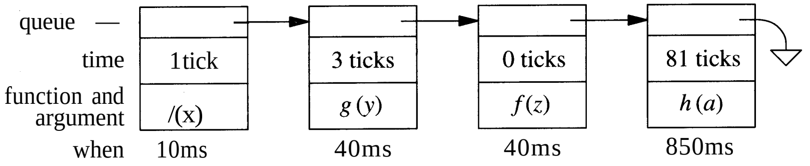
An important event is the scheduling that periodically raises or lowers the CPU priority for each process in the system based on that process's recent CPU usage (see Section 4.4). The rescheduling calculation is done once per second. The scheduler is started at boot time, and each time that it runs, it requests that it be invoked again 1 second in the future.

On a heavily loaded system with many processes, the scheduler may take a long time to complete its job. Posting its next invocation 1 second after each completion may cause scheduling to occur less frequently than once per second. However, as the scheduler is not responsible for any time-critical functions, such as maintaining the time of day, scheduling less frequently than once a second is normally not a problem.

The data structure that describes waiting events is called the *callout queue.* Figure 3.2 shows an example of the callout queue. When a process schedules an event, it specifies a function to be called, a pointer to be passed as an argument to the function, and the number of clock ticks until the event should occur.

The queue is sorted in time order, with the events that are to occur soonest at the front, and the most distant events at the end. The time for each event is kept as a difference from the time of the previous event on the queue. Thus, the *hardclock( )* routine needs only to check the time to expire of the first element to determine whether *softclock( )* needs to run. In addition, decrementing the time to expire of the first element decrements the time for all events. The *softclock( )* routine executes events from the front of the queue whose time has decremented to zero until it finds an event with a still-future (positive) time. New events are added to the queue much less frequently than the queue is checked to see whether

Figure 3.2 Timer events in the callout queue.



any events are to occur. So, it is more efficient to identify the proper location to place an event when that event is added to the queue than to scan the entire queue to determine which events should occur at any single time.

The single argument is provided for the callout-queue function that is called, so that one function can be used by multiple processes. For example, there is a single real-time timer function that sends a signal to a process when a timer expires. Every process that has a real-time timer running posts a timeout request for this function; the argument that is passed to the function is a pointer to the process structure for the process. This argument enables the timeout function to deliver the signal to the correct process.

Timeout processing is more efficient when the timeouts are specified in ticks. Time updates require only an integer decrement, and checks for timer expiration require only a comparison against zero. If the timers contained time values, decrementing and comparisons would be more complex. If the number of events to be managed were large, the cost of the linear search to insert new events correctly could dominate the simple linear queue used in 4.4BSD. Other possible approaches include maintaining a heap with the next-occurring event at the top [Barkley & Lee, 1988], or maintaining separate queues of short-, medium- and long-term events [Varghese & Lauck, 1987].

3.5 Memory-Management Services

The memory organization and layout associated with a 4.4BSD process is shown in Fig. 3.3. Each process begins execution with three memory segments, called text, data, and stack. The data segment is divided into initialized data and uninitialized data (also known as bss). The text is read-only and is normally shared by all processes executing the file, whereas the data and stack areas can be written by, and are private to, each process. The text and initialized data for the process are read from the executable file.

An *executable file* is distinguished by its being a plain file (rather than a directory, special file, or symbolic link) and by its having 1 or more of its execute bits set. In the traditional *a.out* executable format, the first few bytes of the file contain a *magic number* that specifies what type of executable file that file is. Executable files fall into two major classes:

1. Files that must be read by an *interpreter*
2. Files that are directly executable

In the first class, the first 2 bytes of the file are the two-character sequence #! followed by the pathname of the interpreter to be used. (This pathname is currently limited by a compile-time constant to 30 characters.) For example, **#!/bin/sh** refers to the Bourne shell. The kernel executes the named interpreter, passing the name of the file that is to be interpreted as an argument. To prevent loops, 4.4BSD allows only one level of interpretation, and a file's interpreter may not itself be interpreted.

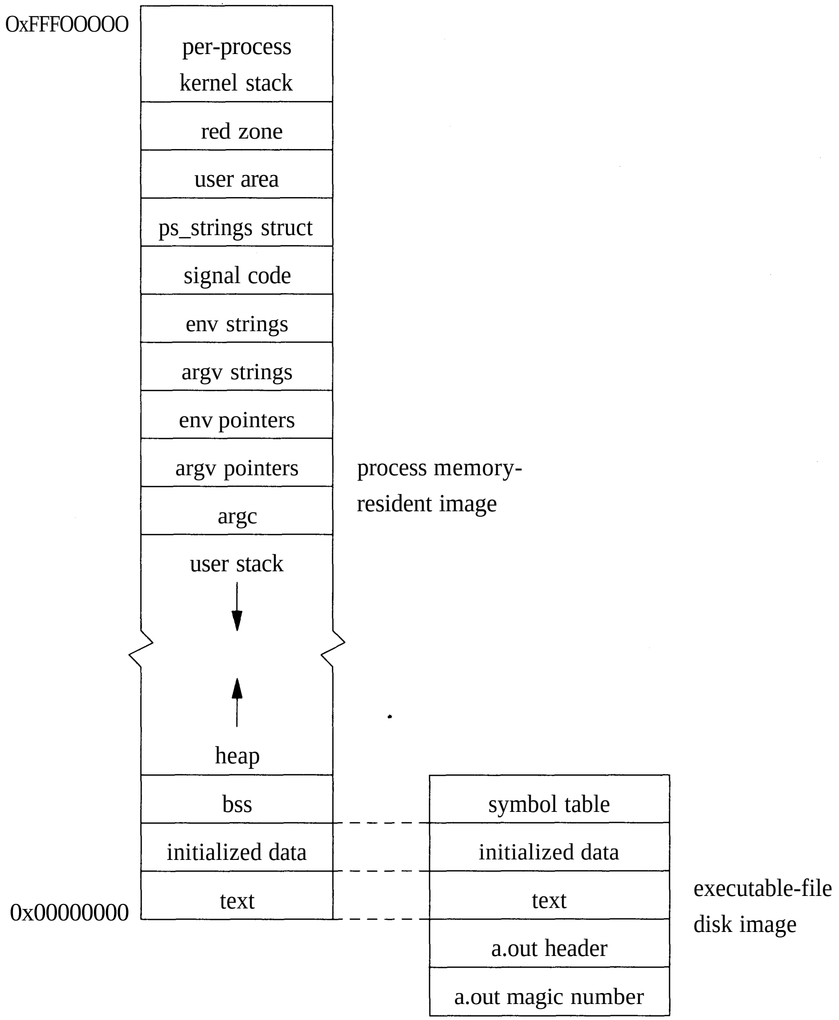


Figure 3.3 Layout of a UNIX process in memory and on disk.

For performance reasons, most files are directly executable. Each directly executable file has a magic number that specifies whether that file can be paged and whether the text part of the file can be shared among multiple processes. Following the magic number is an *exec* header that specifies the sizes of text, initialized data, uninitialized data, and additional information for debugging. (The debugging information is not used by the kernel or by the executing program.) Following the header is an image of the text, followed by an image of the initialized data. Uninitialized data are not contained in the executable file because they can be created on demand using zero-filled memory.

To begin execution, the kernel arranges to have the text portion of the file mapped into the low part of the process address space. The initialized data portion of the file is mapped into the address space following the text. An area equal to the uninitialized data region is created with zero-filled memory after the initialized data region. The stack is also created from zero-filled memory. Although the stack should not need to be zero filled, early UNIX systems made it so. In an attempt to save some startup time, the developers modified the kernel to not zero fill the stack, leaving the random previous contents of the page instead. Numerous programs stopped working because they depended on the local variables in their *main* procedure being initialized to zero. Consequently, the zero filling of the stack was restored.

Copying into memory the entire text and initialized data portion of a large program causes a long startup latency. 4.4BSD avoids this startup time by *demand paging* the program into memory, rather than preloading the program. In demand paging, the program is loaded in small pieces (pages) as it is needed, rather than all at once before it begins execution. The system does demand paging by dividing up the address space into equal-sized areas called pages. For each page, the kernel records the offset into the executable file of the corresponding data. The first access to an address on each page causes a page-fault trap in the kernel. The page-fault handler reads the correct page of the executable file into the process memory. Thus, the kernel loads only those parts of the executable file that are needed. Chapter 5 explains paging details.

The uninitialized data area can be extended with zero-filled pages using the system call *sbrk,* although most user processes use the library routine *malloc( )* a more programmer-friendly interface to *sbrk.* This allocated memory, which grows from the top of the original data segment, is called the *heap.* On the HP300, the stack grows down from the top of memory, whereas the heap grows up from the bottom of memory.

Above the user stack are areas of memory that are created by the system when the process is started. Directly above the user stack is the number of arguments *(argc),* the argument vector *(argv),* and the process environment vector *(envp)* set up when the program was executed. Above them are the argument and environment strings themselves. Above them is the signal code, used when the system delivers signals to the process; above that is the *structps_strings* structure, used by *ps* to locate the *argv* of the process. At the top of user memory is the user area (u.), the *red zone,* and the per-process kernel stack. The red zone may or may not be present in a port to an architecture. If present, it is implemented as a page of read-only memory immediately below the per-process kernel stack. Any attempt to allocate below the fixed-size kernel stack will result in a memory fault, protecting the user area from being overwritten. On some architectures, it is not possible to mark these pages as read-only, or having the kernel stack attempt to write a write protected page would result in unrecoverable system failure. In these cases, other approaches can be taken—for example, checking during each clock interrupt to see whether the current kernel stack has grown too large.

In addition to the information maintained in the user area, a process usually requires the use of some global system resources. The kernel maintains a linked list of processes, called the *process table,* which has an entry for each process in the system. Among other data, the process entries record information on scheduling and on virtual-memory allocation. Because the entire process address space, including the user area, may be swapped out of main memory, the process entry must record enough information to be able to locate the process and to bring that process back into memory. In addition, information needed while the process is swapped out (e.g., scheduling information) must be maintained in the process entry, rather than in the user area, to avoid the kernel swapping in the process only to decide that it is not at a high-enough priority to be run.

Other global resources associated with a process include space to record information about descriptors and page tables that record information about physical-memory utilization.

3.6 Timing Services

The kernel provides several different timing services to processes. These services include timers that run in real time and timers that run only while a process is executing.

Real Time

The system's time offset since January 1, 1970, Universal Coordinated Time (UTC), also known as the Epoch, is returned by the system call *gettimeofday.* Most modern processors (including the HP300 processors) maintain a battery-backup time-of-day register. This clock continues to run even if the processor is turned off. When the system boots, it consults the processor's time-of-day register to find out the current time. The system's time is then maintained by the clock interrupts. At each interrupt, the system increments its global time variable by an amount equal to the number of microseconds per tick. For the HP300, running at 100 ticks per second, each tick represents 10,000 microseconds.

Adjustment of the Time

Often, it is desirable to maintain the same time on all the machines on a network. It is also possible to keep more accurate time than that available from the basic processor clock. For example, hardware is readily available that listens to the set of radio stations that broadcast UTC synchronization signals in the United States. When processes on different machines agree on a common time, they will wish to change the clock on their host processor to agree with the networkwide time value. One possibility is to change the system time to the network time using the *settimeofday* system call. Unfortunately, the *settimeofday* system call will result in time running backward on machines whose clocks were fast. Time running backward can confuse user programs (such as **make)** that expect time to invariably increase. To avoid this problem, the system provides the *adjtime* system call [Gusella et al, 1994]. The *adjtime* system call takes a time delta (either positive or negative) and changes the rate at which time advances by 10 percent, faster or slower, until the time has been corrected. The operating system does the speedup by incrementing the global time by 11,000 microseconds for each tick, and does the slowdown by incrementing the global time by 9,000 microseconds for each tick. Regardless, time increases monotonically, and user processes depending on the ordering of file-modification times are not affected. However, time changes that take tens of seconds to adjust will affect programs that are measuring time intervals by using repeated calls to *gettimeofday.*

External Representation

Time is always exported from the system as microseconds, rather than as clock ticks, to provide a resolution-independent format. Internally, the kernel is free to select whatever tick rate best trades off clock-interrupt-handling overhead with timer resolution. As the tick rate per second increases, the resolution of the system timers improves, but the time spent dealing with hardclock interrupts increases. As processors become faster, the tick rate can be increased to provide finer resolution without adversely affecting user applications.

All filesystem (and other) timestamps are maintained in UTC offsets from the Epoch. Conversion to local time, including adjustment for daylight-savings time, is handled externally to the system in the C library.

Interval Time

The system provides each process with three interval timers. The *real* timer decrements in real time. An example of use for this timer is a library routine maintaining a wakeup-service queue. A SIGALRM signal is delivered to the process when this timer expires. The real-time timer is run from the timeout queue maintained by the *softclock()* routine (see Section 3.4).

The *profiling* timer decrements both in process virtual time (when running in user mode) and when the system is running on behalf of the process. It is designed to be used by processes to profile their execution statistically. A SIGPROF signal is delivered to the process when this timer expires. The profiling timer is implemented by the *hardclock( )* routine. Each time that *hardclock( )* runs, it checks to see whether the currently running process has requested a profiling timer; if it has, *hardclock( )* decrements the timer, and sends the process a signal when zero is reached.

The *virtual* timer decrements in process virtual time. It runs only when the process is executing in user mode. A SIGVTALRM signal is delivered to the process when this timer expires. The virtual timer is also implemented in *hardclock()* as the profiling timer is, except that it decrements the timer for the current process only if it is executing in user mode, and not if it is running in the kernel.

User, Group, and Other Identifiers

One important responsibility of an operating system is to implement access-control mechanisms. Most of these access-control mechanisms are based on the notions of individual users and of groups of users. Users are named by a 32-bit number called a *user identifier* (UID). UIDs are not assigned by the kernel—they are assigned by an outside administrative authority. UIDs are the basis for accounting, for restricting access to privileged kernel operations, (such as the request used to reboot a running system), for deciding to what processes a signal may be sent, and as a basis for filesystem access and disk-space allocation. A single user, termed the *superuser* (also known by the user name *roof),* is trusted by the system and is permitted to do any supported kernel operation. The superuser is identified not by any specific name, such as *root,* but instead by a UID of zero.

Users are organized into *groups.* Groups are named by a 32-bit number called a *group identifier* (GID). GIDs, like UIDs, are used in the filesystem access-control facilities and in disk-space allocation.

The state of every 4.4BSD process includes a UID and a set of GIDs. A process's filesystem-access privileges are defined by the UID and GIDs of the process (for the filesystem hierarchy beginning at the process's root directory). Normally, these identifiers are inherited automatically from the parent process when a new process is created. Only the superuser is permitted to alter the UID or GID of a process. This scheme enforces a strict compartmentalization of privileges, and ensures that no user other than the superuser can *gain* privileges.

Each file has three sets of permission bits, for read, write, or execute permission for each of owner, group, and other. These permission bits are checked in the following order:

1. If the UID of the file is the same as the UID of the process, only the owner permissions apply; the group and other permissions are not checked.
2. If the UIDs do not match, but the GID of the file matches one of the GIDs of the process, only the group permissions apply; the owner and other permissions are not checked.
3. Only if the UID and GIDs of the process fail to match those of the file are the permissions for all others checked. If these permissions do not allow the requested operation, it will fail.

The UID and GIDs for a process are inherited from its parent. When a user logs in, the login program (see Section 14.6) sets the UID and GIDs before doing the *exec* system call to run the user's login shell; thus, all subsequent processes will inherit the appropriate identifiers.

Often, it is desirable to grant a user limited additional privileges. For example, a user who wants to send mail must be able to append the mail to another user's mailbox. Making the target mailbox writable by all users would permit a user other than its owner to modify messages in it (whether maliciously or unintentionally). To solve this problem, the kernel allows the creation of programs that are granted additional privileges while they are running. Programs that run with a different UID are called *set-user-identifier* (setuid) programs; programs that run with an additional group privilege are called *set-group-identifier* (setgid) programs [Ritchie, 1979]. When a setuid program is executed, the permissions of the process are augmented to include those of the UID associated with the program. The UID of the program is termed the *effective UID* of the process, whereas the original UID of the process is termed the *real UID.* Similarly, executing a setgid program augments a process's permissions with those of the program's GID, and the *effective GID* and *real GID* are defined accordingly.

Systems can use setuid and setgid programs to provide controlled access to files or services. For example, the program that adds mail to the users' mailbox runs with the privileges of the superuser, which allow it to write to any file in the system. Thus, users do not need permission to write other users' mailboxes, but can still do so by running this program. Naturally, such programs must be written carefully to have only a limited set of functionality!

The UID and GIDs are maintained in the per-process area. Historically, GIDs were implemented as one distinguished GID (the effective GID) and a supplementary array of GIDs, which was logically treated as one set of GIDs. In 4.4BSD, the distinguished GID has been made the first entry in the array of GIDs. The supplementary array is of a fixed size (16 in 4.4BSD), but may be changed by recompiling the kernel.

4.4BSD implements the setgid capability by setting the zeroth element of the supplementary groups array of the process that executed the setgid program to the group of the file. Permissions can then be checked as it is for a normal process. Because of the additional group, the setgid program may be able to access more files than can a user process that runs a program without the special privilege. The login program duplicates the zeroth array element into the first array element when initializing the user's supplementary group array, so that, when a setgid program is run and modifies the zeroth element, the user does not lose any privileges.

The setuid capability is implemented by the effective UID of the process being changed from that of the user to that of the program being executed. As it will with setgid, the protection mechanism will now permit access without any change or special knowledge that the program is running setuid. Since a process can have only a single UID at a time, it is possible to lose some privileges while running setuid. The previous real UID is still maintained as the real UID when the new effective UID is installed. The real UID, however, is not used for any validation checking.

A setuid process may wish to revoke its special privilege temporarily while it is running. For example, it may need its special privilege to access a restricted file at only the start and end of its execution. During the rest of its execution, it should have only the real user's privileges. In 4.3BSD, revocation of privilege was done by switching of the real and effective UIDs. Since only the effective UID is used for access control, this approach provided the desired semantics and provided a place to hide the special privilege. The drawback to this approach was that the real and effective UIDs could easily become confused.

In 4.4BSD, an additional identifier, the *saved UID,* was introduced to record the identity of setuid programs. When a program is *exec'ed,* its effective UID is copied to its saved UID. The first line of Table 3.1 shows an unprivileged program for which the real, effective, and saved UIDs are all those of the real user. The second line of Table 3.1 show a setuid program being run that causes the effective UID to be set to its associated special-privilege UID. The special-privilege UID has also been copied to the saved UID.

Also added to 4.4BSD was the new *seteuid* system call that sets only the effective UID; it does not affect the real or saved UIDs. The *seteuid* system call is permitted to set the effective UID to the value of either the real or the saved UID. Lines 3 and 4 of Table 3.1 show how a setuid program can give up and then reclaim its special privilege while continuously retaining its correct real UID. Lines 5 and 6 show how a setuid program can run a subprocess without granting the latter the special privilege. First, it sets its effective UID to the real UID. Then, when it *exec's* the subprocess, the effective UID is copied to the saved UID, and all access to the special-privilege UID is lost.

A similar saved GID mechanism permits processes to switch between the real GID and the initial effective GID.

Host Identifiers

An additional identifier is defined by the kernel for use on machines operating in a networked environment. A string (of up to 256 characters) specifying the host's name is maintained by the kernel. This value is intended to be defined uniquely for each machine in a network. In addition, in the Internet domain-name system, each machine is given a unique 32-bit number. Use of these identifiers permits applications to use networkwide unique identifiers for objects such as processes, files, and users, which is useful in the construction of distributed applications [Gifford, 1981]. The host identifiers for a machine are administered outside the kernel.

Table 3.1 Actions affecting the real, effective, and saved UIDs. R—real user identifier; S—special-privilege user identifier.

Action	Real	Effective	Saved
1. exec-normal	R	R	R
2. exec-setuid	R	S	S
3. *seteuid(R)*	R	R	S
4. *seteuid(S)*	R	S	S
5. *seteuid(R)*	R	R	S
6. exec-normal	R	R	R

The 32-bit host identifier found in 4.3BSD has been deprecated in 4.4BSD, and is supported only if the system is compiled for 4.3BSD compatibility.

Process Groups and Sessions

Each process in the system is associated with a *process group.* The group of processes in a process group is sometimes referred to as a *job*, and manipulated as a single entity by processes such as the shell. Some signals (e.g., SIGINT) are delivered to all members of a process group, causing the group as a whole to suspend or resume execution, or to be interrupted or terminated.

Sessions were designed by the IEEE POSIX. 1003.1 Working Group with the intent of fixing a long-standing security problem in UNIX—namely, that processes could modify the state of terminals that were trusted by another user's processes. A *session* is a collection of process groups, and all members of a process group are members of the same session. In 4.4BSD, when a user first logs onto the system, they are entered into a new session. Each session has a *controlling process*, which is normally the user's login shell. All subsequent processes created by the user are part of process groups within this session, unless they explicitly create a new session. Each session also has an associated *login name*, which is usually the user's login name. This name can be changed by only the superuser.

Each session is associated with a terminal, known as its *controlling terminal.* Each controlling terminal has a process group associated with it. Normally, only processes that are in the terminal's current process group read from or write to the terminal, allowing arbitration of a terminal between several different jobs. When the controlling process exits, access to the terminal is taken away from any remaining processes within the session.

Newly created processes are assigned process IDs distinct from all already-existing processes and process groups, and are placed in the same process group and session as their parent. Any process may set its process group equal to its process ID (thus creating a new process group) or to the value of any process group within its session. In addition, any process may create a new session, as long as it is not already a process-group leader. Sessions, process groups, and associated topics are discussed further in Section 4.8 and in Section 10.5.

3.8 Resource Services

All systems have limits imposed by their hardware architecture and configuration to ensure reasonable operation and to keep users from accidentally (or maliciously) creating resource shortages. At a minimum, the hardware limits must be imposed on processes that run on the system. It is usually desirable to limit processes further, below these hardware-imposed limits. The system measures resource utilization, and allows limits to be imposed on consumption either at or below the hardware-imposed limits.

Process Priorities

The 4.4BSD system gives CPU scheduling priority to processes that have not used CPU time recently. This priority scheme tends to favor processes that execute for only short periods of time—for example, interactive processes. The priority selected for each process is maintained internally by the kernel. The calculation of the priority is affected by the per-process *nice* variable. Positive *nice* values mean that the process is willing to receive less than its share of the processor. Negative values of *nice* mean that the process wants more than its share of the processor. Most processes run with the default *nice* value of zero, asking neither higher nor lower access to the processor. It is possible to determine or change the *nice* currently assigned to a process, to a process group, or to the processes of a specified user. Many factors other than *nice* affect scheduling, including the amount of CPU time that the process has used recently, the amount of memory that the process has used recently, and the current load on the system. The exact algorithms that are used are described in Section 4.4.

Resource Utilization

As a process executes, it uses system resources, such as the CPU and memory. The kernel tracks the resources used by each process and compiles statistics describing this usage. The statistics managed by the kernel are available to a process while the latter is executing. When a process terminates, the statistics are made available to its parent via the *wait* family of system calls.

The resources used by a process are returned by the system call *getrusage.* The resources used by the current process, or by all the terminated children of the current process, may be requested. This information includes

- The amount of user and system time used by the process
- The memory utilization of the process
- The paging and disk I/O activity of the process
- The number of voluntary and involuntary context switches taken by the process
- The amount of interprocess communication done by the process

The resource-usage information is collected at locations throughout the kernel. The CPU time is collected by the *statclock()* function, which is called either by the system clock in *hardclock( )* or, if an alternate clock is available, by the alternate-clock interrupt routine. The kernel scheduler calculates memory utilization by sampling the amount of memory that an active process is using at the same time that it is recomputing process priorities. The *vm_fault()* routine recalculates the paging activity each time that it starts a disk transfer to fulfill a paging request (see Section 5.11). The I/O activity statistics are collected each time that the process has to start a transfer to fulfill a file or device I/O request, as well as when the general system statistics are calculated. The IPC communication activity is updated each time that information is sent or received.

Resource Limits

The kernel also supports limiting of certain per-process resources. These resources include

- The maximum amount of CPU time that can be accumulated
- The maximum bytes that a process can request be locked into memory
- The maximum size of a file that can be created by a process
- The maximum size of a process's data segment
- The maximum size of a process's stack segment
- The maximum size of a core file that can be created by a process
- The maximum number of simultaneous processes allowed to a user
- The maximum number of simultaneous open files for a process
- The maximum amount of physical memory that a process may use at any given moment

For each resource controlled by the kernel, two limits are maintained: a *soft limit* and a *hard limit.* All users can alter the soft limit within the range of 0 to the corresponding hard limit. All users can (irreversibly) lower the hard limit, but only the superuser can raise the hard limit. If a process exceeds certain soft limits, a signal is delivered to the process to notify it that a resource limit has been exceeded. Normally, this signal causes the process to terminate, but the process may either catch or ignore the signal. If the process ignores the signal and fails to release resources that it already holds, further attempts to obtain more resources will result in errors.

Resource limits are generally enforced at or near the locations that the resource statistics are collected. The CPU time limit is enforced in the process context-switching function. The stack and data-segment limits are enforced by a return of allocation failure once those limits have been reached. The file-size limit is enforced by the filesystem.

Filesystem Quotas

In addition to limits on the size of individual files, the kernel optionally enforces limits on the total amount of space that a user or group can use on a filesystem. Our discussion of the implementation of these limits is deferred to Section 7.4.

System-Operation Services

There are several operational functions having to do with system startup and shutdown. The bootstrapping operations are described in Section 14.2. System shutdown is described in Section 14.7.

Accounting

The system supports a simple form of resource accounting. As each process terminates, an accounting record describing the resources used by that process is written to a systemwide accounting file. The information supplied by the system comprises

- The name of the command that ran
- The amount of user and system CPU time that was used
- The elapsed time the command ran
- The average amount of memory used
- The number of disk I/O operations done
- The UID and GID of the process
- The terminal from which the process was started

The information in the accounting record is drawn from the run-time statistics that were described in Section 3.8. The granularity of the time fields is in sixty-fourths of a second. To conserve space in the accounting file, the times are stored in a 16-bit word as a *floating-point* number using 3 bits as a base-8 exponent, and the other 13 bits as the fractional part. For historic reasons, the same floating-point-conversion routine processes the count of disk operations, so the number of disk operations must be multiplied by 64 before it is converted to the floating-point representation.

There are also flags that describe how the process terminated, whether it ever had superuser privileges, and whether it did an *exec* after a fork

The superuser requests accounting by passing the name of the file to be used for accounting to the kernel. As part of a process exiting, the kernel appends an accounting record to the accounting file. The kernel makes no use of the accounting records; the records' summaries and use are entirely the domain of user-level accounting programs. As a guard against a filesystem running out of space because of unchecked growth of the accounting file, the system suspends accounting when the filesystem is reduced to only 2 percent remaining free space. Accounting resumes when the filesystem has at least 4 percent free space.

The accounting information has certain limitations. The information on run time and memory usage is only approximate because it is gathered statistically. Accounting information is written only when a process exits, so processes that are still running when a system is shut down unexpectedly do not show up in the accounting file. (Obviously, long-lived system daemons are among such processes.) Finally, the accounting records fail to include much information needed to do accurate billing, including usage of other resources, such as tape drives and printers.

Exercises

3.1 Describe three types of system activity.

3.2 When can a routine executing in the top half of the kernel be preempted? When can it be interrupted?

3.3 Why are routines executing in the bottom half of the kernel precluded from using information located in the user area?

3.4 Why does the system defer as much work as possible from high-priority interrupts to lower-priority software-interrupt processes?

3.5 What determines the shortest (nonzero) time period that a user process can request when setting a timer?

3.6 How does the kernel determine the system call for which it has been invoked?

3.7 How are initialized data represented in an executable file? How are uninitialized data represented in an executable file? Why are the representations different?

3.8 Describe how the "#!" mechanism can be used to make programs that require emulation appear as though they were normal executables.

3.9 Is it possible for a file to have permissions set such that its owner cannot read it, even though a group can? Is this situation possible if the owner is a member of the group that can read the file? Explain your answers.

*3.10 Describe the security implications of not zero filling the stack region at program startup.

*3.11 Why is the conversion from UTC to local time done by user processes, rather than in the kernel?

*3.12 What is the advantage of having the kernel, rather than an application, restart an interrupted system call?

*3.13 Describe a scenario in which the sorted-difference algorithm used for the callout queue does not work well. Suggest an alternative data structure that runs more quickly than does the sorted-difference algorithm for your scenario.

*3.14 The SIGPROF profiling timer was originally intended to replace the *profil* system call to collect a statistical sampling of a program's program counter. Give two reasons why the *profil* facility had to be retained.

**3.15 What weakness in the process-accounting mechanism makes the latter unsuitable for use in a commercial environment?

eferences

Barkley & Lee, 1988.
R. E. Barkley & T. P. Lee, "A Heap-Based Callout Implementation to Meet Real-Time Needs," *USENIX Association Conference Proceedings,* pp. 213-222, June 1988.

Gifford, 1981.
D. Gifford, "Information Storage in a Decentralized Computer System," PhD Thesis, Electrical Engineering Department, Stanford University, Stanford, CA, 1981.

Gusellaetal, 1994.
R. Gusella, S. Zatti, & J. M. Bloom, 'The Berkeley UNIX Time Synchronization Protocol," in *4.4BSD System Manager's Manual,* pp. 12:1-10, O'Reilly & Associates, Inc., Sebastopol, CA, 1994.

McCanne & Torek, 1993.
S. McCanne & C. Torek, "A Randomized Sampling Clock for CPU Utilization Estimation and Code Profiling," *USENIX Association Conference Proceedings,* pp. 387-394, January 1993.

Ritchie, 1979.
D. M. Ritchie, "Protection of Data File Contents," *United States Patent,* no. 4,135,240, United States Patent Office, Washington, D.C., January 16, 1979. Assignee: Bell Telephone Laboratories, Inc., Murray Hill, NJ, Appl. No.: 377,591, Filed: Jul. 9, 1973.

Varghese & Lauck, 1987.
G. Varghese & T. Lauck, "Hashed and Hierarchical Timing Wheels: Data Structures for the Efficient Implementation of a Timer Facility," *Proceedings of the Eleventh Symposium on Operating Systems Principles,* pp. 25-38,November1987.

PART 2

Processes

CHAPTER 4

Process Management

4.1 Introduction to Process Management

A *process* is a program in execution. A process must have system resources, such as memory and the underlying CPU. The kernel supports the illusion of concurrent execution of multiple processes by scheduling system resources among the set of processes that are ready to execute. This chapter describes the composition of a process, the method that the system uses to switch between processes, and the scheduling policy that it uses to promote sharing of the CPU. Later chapters study process creation and termination, signal facilities, and process-debugging facilities.

Two months after the developers began the first implementation of the UNIX operating system, there were two processes: one for each of the terminals of the PDP-7. At age 10 months, and still on the PDP-7, UNIX had many processes, the *fork* operation, and something like the *wait* system call. A process executed a new program by reading in a new program on top of itself. The first PDP-11 system (First Edition UNIX) saw the introduction of *exec*. All these systems allowed only one process in memory at a time. When a PDP-11 with memory management (a KS-11) was obtained, the system was changed to permit several processes to remain in memory simultaneously, to reduce swapping. But this change did not apply to multiprogramming because disk I/O was synchronous. This state of affairs persisted into 1972 and the first PDP-11/45 system. True multiprogramming was finally introduced when the system was rewritten in C. Disk I/O for one process could then proceed while another process ran. The basic structure of process management in UNIX has not changed since that time [Ritchie, 1988].

A process operates in either *user mode* or *kernel mode*. In user mode, a process executes application code with the machine in a nonprivileged protection mode. When a process requests services from the operating system with a system call, it switches into the machine's privileged protection mode via a protected mechanism, and then operates in kernel mode.

The resources used by a process are similarly split into two parts. The resources needed for execution in user mode are defined by the CPU architecture and typically include the CPU's general-purpose registers, the program counter, the processor-status register, and the stack-related registers, as well as the contents of the memory segments that constitute the 4.4BSD notion of a program (the text, data, and stack segments).

Kernel-mode resources include those required by the underlying hardware—such as registers, program counter, and stack pointer—and also by the state required for the 4.4BSD kernel to provide system services for a process. This *kernel state* includes parameters to the current system call, the current process's user identity, scheduling information, and so on. As described in Section 3.1, the kernel state for each process is divided into several separate data structures, with two primary structures: *the process structure* and the *user structure.*

The process structure contains information that must always remain resident in main memory, along with references to a number of other structures that remain resident; whereas the user structure contains information that needs to be resident only when the process is executing (although user structures of other processes also may be resident). User structures are allocated dynamically through the memory-management facilities. Historically, more than one-half of the process state was stored in the user structure. In 4.4BSD, the user structure is used for only the per-process kernel stack and a couple of structures that are referenced from the process structure. Process structures are allocated dynamically as part of process creation, and are freed as part of process exit.

Multiprogramming

The 4.4BSD system supports transparent multiprogramming: the illusion of concurrent execution of multiple processes or programs. It does so by *context switching*—that is, by switching between the execution context of processes. A mechanism is also provided for *scheduling* the execution of processes—that is, for deciding which one to execute next. Facilities are provided for ensuring consistent access to data structures that are shared among processes.

Context switching is a hardware-dependent operation whose implementation is influenced by the underlying hardware facilities. Some architectures provide machine instructions that save and restore the hardware-execution context of the process, including the virtual-address space. On the others, the software must collect the hardware state from various registers and save it, then load those registers with the new hardware state. All architectures must save and restore the software state used by the kernel.

Context switching is done frequently, so increasing the speed of a context switch noticeably decreases time spent in the kernel and provides more time for execution of user applications. Since most of the work of a context switch is expended in saving and restoring the operating context of a process, reducing the amount of the information required for that context is an effective way to produce faster context switches.

Scheduling

Fair scheduling of processes is an involved task that is dependent on the types of executable programs and on the goals of the scheduling policy. Programs are characterized according to the amount of computation and the amount of I/O that they do. Scheduling policies typically attempt to balance resource utilization against the time that it takes for a program to complete. A process's priority is periodically recalculated based on various parameters, such as the amount of CPU time it has used, the amount of memory resources it holds or requires for execution, and so on. An exception to this rule is real-time scheduling, which must ensure that processes finish by a specified deadline or in a particular order; the 4.4BSD kernel does not implement real-time scheduling.

4.4BSD uses a priority-based scheduling policy that is biased to favor *interactive programs,* such as text editors, over long-running batch-type jobs. Interactive programs tend to exhibit short bursts of computation followed by periods of inactivity or I/O. The scheduling policy initially assigns to each process a high execution priority and allows that process to execute for a fixed *time slice.* Processes that execute for the duration of their slice have their priority lowered, whereas processes that give up the CPU (usually because they do I/O) are allowed to remain at their priority. Processes that are inactive have their priority raised. Thus, jobs that use large amounts of CPU time sink rapidly to a low priority, whereas interactive jobs that are mostly inactive remain at a high priority so that, when they are ready to run, they will preempt the long-running lower-priority jobs. An interactive job, such as a text editor searching for a string, may become compute bound briefly, and thus get a lower priority, but it will return to a high priority when it is inactive again while the user thinks about the result.

The system also needs a scheduling policy to deal with problems that arise from not having enough main memory to hold the execution contexts of all processes that want to execute. The major goal of this scheduling policy is to minimize *thrashing*—a phenomenon that occurs when memory is in such short supply that more time is spent in the system handling page faults and scheduling processes than in user mode executing application code.

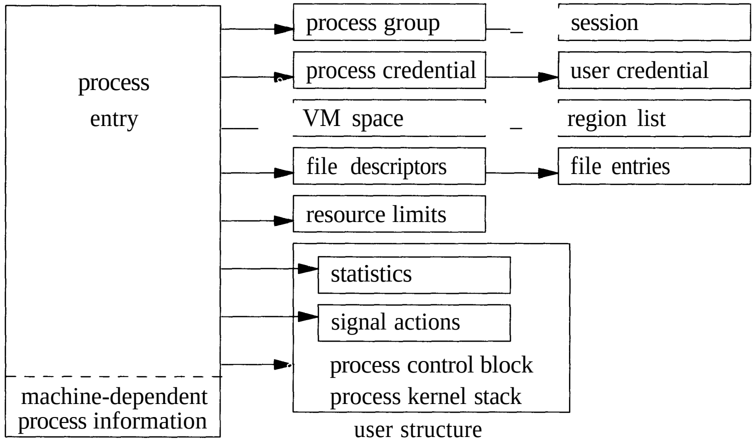
The system must both detect and eliminate thrashing. It detects thrashing by observing the amount of free memory. When the system has few free memory pages and a high rate of new memory requests, it considers itself to be thrashing. The system reduces thrashing by marking the least-recently run process as not being allowed to run. This marking allows the pageout daemon to push all the pages associated with the process to backing store. On most architectures, the kernel also can push to backing store the user area of the marked process. The effect of these actions is to cause the process to be swapped out (see Section 5.12). The memory freed by blocking the process can then be distributed to the remaining processes, which usually can then proceed. If the thrashing continues, additional processes are selected for being blocked from running until enough memory becomes available for the remaining processes to run effectively. Eventually, enough processes complete and free their memory that blocked processes can resume execution. However, even if there is not enough memory, the blocked processes are allowed to resume execution after about 20 seconds. Usually, the thrashing condition will return, requiring that some other process be selected for being blocked (or that an administrative action be taken to reduce the load).

The orientation of the scheduling policy toward an interactive job mix reflects the original design of 4.4BSD for use in a time-sharing environment. Numerous papers have been written about alternative scheduling policies, such as those used in batch-processing environments or real-time systems. Usually, these policies require changes to the system in addition to alteration of the scheduling policy [Khanna et al, 1992].

4.2 Process State

The layout of process state was completely reorganized in 4.4BSD. The goal was to support multiple *threads* that share an address space and other resources. Threads have also been called *lightweight processes* in other systems. A thread is the unit of execution of a process; it requires an address space and other resources, but it can share many of those resources with other threads. Threads sharing an address space and other resources are scheduled independently, and can all do system calls simultaneously. The reorganization of process state in 4.4BSD was designed to support threads that can select the set of resources to be shared, known as *variable-weight processes* [Aral et al, 1989]. Unlike some other implementations of threads, the BSD model associates a process ID with each thread, rather than with a collection of threads sharing an address space.

Figure 4.1 Process state.



The developers did the reorganization by moving many components of process state from the process and user structures into separate substructures for each type of state information, as shown in Fig. 4.1. The process structure references all the substructures directly or indirectly. The use of global variables in the user structure was completely eliminated. Variables moved out of the user structure include the open file descriptors that may need to be shared among different threads, as well as system-call parameters and error returns. The process structure itself was also shrunk to about one-quarter of its former size. The idea is to minimize the amount of storage that must be allocated to support a thread. The 4.4BSD distribution did not have kernel-thread support enabled, primarily because the C library had not been rewritten to be able to handle multiple threads.

All the information in the substructures shown in Fig. 4.1 can be shared among threads running within the same address space, except the per-thread statistics, the signal actions, and the per-thread kernel stack. These unshared structures need to be accessible only when the thread may be scheduled, so they are allocated in the user structure so that they can be moved to secondary storage when memory resources are low. The following sections describe the portions of these structures that are relevant to process management. The VM space and its related structures are described more fully in Chapter 5.

The Process Structure

In addition to the references to the substructures, the process entry shown in Fig. 4.1 contains the following categories of information:

- **Process identification.** The process identifier and the parent-process identifier
- **Scheduling.** The process priority, user-mode scheduling priority, recent CPU utilization, and amount of time spent sleeping
- **Process state.** The run state of a process (runnable, sleeping, stopped); additional status flags; if the process is sleeping, the *wait channel* the identity of the event for which the process is waiting (see Section 4.3), and a pointer to a string describing the event
- **Signal state.** Signals pending delivery, signal mask, and summary of signal actions
- **Tracing.** Process tracing information
- **Machine state.** The machine-dependent process information
- **Timers.** Real-time timer and CPU-utilization counters

The process substructures shown in Fig. 4.1 have the following categories of information:

- **Process-group identification.** The process group and the session to which the process belongs

- **User credentials.** The real, effective, and saved user and group identifiers
- **Memory management.** The structure that describes the allocation of virtual address space used by the process
- **File descriptors.** An array of pointers to file entries indexed by the process open file descriptors; also, the open file flags and current directory
- **Resource accounting.** The *rusage* structure that describes the utilization of the many resources provided by the system (see Section 3.8)
- **Statistics.** Statistics collected while the process is running that are reported when it exits and are written to the accounting file; also, includes process timers and profiling information if the latter is being collected
- **Signal actions.** The action to take when a signal is posted to a process
- **User structure.** The contents of the user structure (described later in this section)

A process's state has a value, as shown in Table 4.1. When a process is first created with *a fork* system call, it is initially marked as SIDL. The state is changed to SRUN when enough resources are allocated to the process for the latter to begin execution. From that point onward, a process's state will fluctuate among SRUN (runnable—e.g., ready to execute), SSLEEP (waiting for an event), and SSTOP (stopped by a signal or the parent process), until the process terminates. A deceased process is marked as SZOMB until its termination status is communicated to its parent process.

The system organizes process structures into two lists. Process entries are on the *zombproc* list if the process is in the SZOMB state; otherwise, they are on the *allproc* list. The two queues share the same linkage pointers in the process structure, since the lists are mutually exclusive. Segregating the dead processes from the live ones reduces the time spent both by the *wait* system call, which must scan the zombies for potential candidates to return, and by the scheduler and other functions that must scan all the potentially runnable processes.

Table 4.1 Process states.

State	Description
SIDL	intermediate state in process creation
SRUN	runnable
SSLEEP	awaiting an event
SSTOP	process stopped or being traced
SZOMB	intermediate state in process termination

Most processes, except the currently executing process, are also in one of two queues: a *run queue* or a *sleep queue.* Processes that are in a runnable state are placed on a run queue, whereas processes that are blocked awaiting an event are located on a sleep queue. Stopped processes not also awaiting an event are on neither type of queue. The two queues share the same linkage pointers in the process structure, since the lists are mutually exclusive. The run queues are organized according to process-scheduling priority, and are described in Section 4.4. The sleep queues are organized in a hashed data structure that optimizes finding of a sleeping process by the event number (wait channel) for which the process is waiting. The sleep queues are described in Section 4.3.

Every process in the system is assigned a unique identifier termed the *process identifier, (PID).* PIDs are the common mechanism used by applications and by the kernel to reference processes. PIDs are used by applications when the latter are sending a signal to a process and when receiving the exit status from a deceased process. Two PIDs are of special importance to each process: the PID of the process itself and the PID of the process's parent process.

The *p_pglist* list and related lists *(p_pptr, p_children,* and *p_siblings)* are used in locating related processes, as shown in Fig. 4.2. When a process spawns a child process, the child process is added to its parent's *p_children* list. The child process also keeps a backward link to its parent in its *p_pptr* field. If a process has more than one child process active at a time, the children are linked together through their *p_sibling* list entries. In Fig. 4.2, process B is a direct descendent of process A, whereas processes C, D, and E are descendents of process B and are siblings of one another. Process B typically would be a shell that started a pipeline (see Sections 2.4 and 2.6) including processes C, D, and E. Process A probably would be the system-initialization process **init** (see Section 3.1 and Section 14.6).

CPU time is made available to processes according to their *scheduling priority.* A process has two scheduling priorities, one for scheduling user-mode execution and one for scheduling kernel-mode execution. The *p_usrpri* field in the process structure contains the user-mode scheduling priority, whereas the *p_priority* field holds the current kernel-mode scheduling priority. The current priority may be

Figure 4.2 Process-group hierarchy.

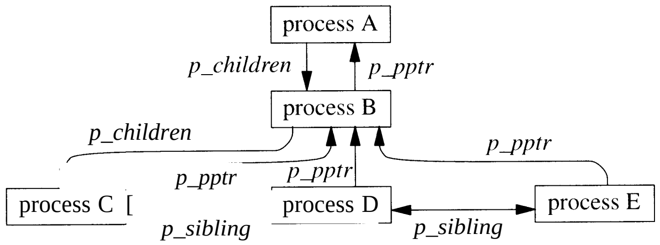


Table 4.2 Process-scheduling priorities.

Priority	Value	Description
PSWP	0	priority while swapping process
PVM	4	priority while waiting for memory
PINOD	8	priority while waiting for file control information
PRIBIO	16	priority while waiting on disk I/O completion
PVFS	20	priority while waiting for a kernel-level filesystem lock
PZERO	22	baseline priority
PSOCK	24	priority while waiting on a socket
PWAIT	32	priority while waiting for a child to exit
PLOCK	36	priority while waiting for user-level filesystem lock
PPAUSE	40	priority while waiting for a signal to arrive
PUSER	50	base priority for user-mode execution

different from the user-mode priority when the process is executing in kernel mode. Priorities range between 0 and 127, with a lower value interpreted as a higher priority (see Table 4.2). User-mode priorities range from PUSER (50) to 127; priorities less than PUSER are used only when a process is *asleep*—that is, awaiting an event in the kernel—and immediately after such a process is awakened. Processes in the kernel are given a higher priority because they typically hold shared kernel resources when they awaken. The system wants to run them as quickly as possible once they get a resource, so that they can use the resource and return it before another process requests it and gets blocked waiting for it.

Historically, a kernel process that is asleep with a priority in the range PZERO to PUSER would be awakened by a signal; that is, it might be awakened and marked runnable if a signal is posted to it. A process asleep at a priority below PZERO would never be awakened by a signal. In 4.4BSD, a kernel process will be awakened by a signal only if it sets the PCATCH flag when it sleeps. The PCATCH flag was added so that a change to a sleep priority does not inadvertently cause a change to the process's interruptibility.

For efficiency, the sleep interface has been divided into two separate entry points: *sleep*() for brief, noninterruptible sleep requests, and *tsleepO* for longer, possibly interrupted sleep requests. The *sleep*() interface is short and fast, to handle the common case of a short sleep. The *tsleep( )* interface handles all the special cases including interruptible sleeps, sleeps limited to a maximum time duration, and the processing of restartable system calls. The *tsleep( )* interface also includes a reference to a string describing the event that the process awaits; this string is externally visible. The decision of whether to use an interruptible sleep is dependent on how long the process may be blocked. Because it is complex to be prepared to handle signals in the midst of doing some other operation, many sleep

requests are not interruptible; that is, a process will not be scheduled to run until the event for which it is waiting occurs. For example, a process waiting for disk I/O will sleep at an uninterruptible priority.

For quickly occurring events, delaying to handle a signal until after they complete is imperceptible. However, requests that may cause a process to sleep for a long period, such as while a process is waiting for terminal or network input, must be prepared to have their sleep interrupted so that the posting of signals is not delayed indefinitely. Processes that sleep at interruptible priorities may abort their system call because of a signal arriving before the event for which they are waiting has occurred. To avoid holding a kernel resource permanently, these processes must check why they have been awakened. If they were awakened because of a signal, they must release any resources that they hold. They must then return the error passed back to them by *tsleep( ),* which will be EINTR if the system call is to be aborted after the signal, or ERESTART if it is to be restarted. Occasionally, an event that is supposed to occur quickly, such as a tape I/O, will get held up because of a hardware failure. Because the process is sleeping in the kernel at an uninterruptible priority, it will be impervious to any attempts to send it a signal, even a signal that should cause it to exit unconditionally. The only solution to this problem is to change *sleep*()s on hardware events that may hang to be interruptible. In the remainder of this book, we shall always use *sleep* () when referencing the routine that puts a process to sleep, even when the *tsleep( )* interface may be the one that is being used.

The User Structure

The *user structure* contains the process state that may be swapped to secondary storage. The structure was an important part of the early UNIX kernels; it stored much of the state for each process. As the system has evolved, this state has migrated to the process entry or one of its substructures, so that it can be shared. In 4.4BSD, nearly all references to the user structure have been removed. The only place that user-structure references still exist are in the *fork* system call, where the new process entry has pointers set up to reference the two remaining structures that are still allocated in the user structure. Other parts of the kernel that reference these structures are unaware that the latter are located in the user structure; the structures are always referenced from the pointers in the process table. Changing them to dynamically allocated structures would require code changes in only *fork* to allocate them, and *exit* to free them. The user-structure state includes

- The user- and kernel-mode execution states
- The accounting information
- The signal-disposition and signal-handling state
- Selected process information needed by the debuggers and in core dumps
- The per-process execution stack for the kernel

The current execution state of a process is encapsulated in a *process control block (PCB).* This structure is allocated in the user structure and is defined by the machine architecture; it includes the general-purpose registers, stack pointers, program counter, processor-status longword, and memory-management registers.

Historically, the user structure was mapped to a fixed location in the virtual address space. There were three reasons for using a fixed mapping:

1. On many architectures, the user structure could be mapped into the top of the user-process address space. Because the user structure was part of the user address space, its context would be saved as part of saving of the user-process state, with no additional effort.
2. The data structures contained in the user structure (also called the *u-dot* (u.) *structure,* because all references in C were of the form *u.)* could always be addressed at a fixed address.
3. When a parent forks, its run-time stack is copied for its child. Because the kernel stack is part of the *u.* area, the child's kernel stack is mapped to the same addresses as its parent kernel stack. Thus, all its internal references, such as frame pointers and stack-variable references, work as expected.

On modern architectures with virtual address caches, mapping the user structure to a fixed address is slow and inconvenient. Thus, reason 1 no longer holds. Since the user structure is never referenced by most of the kernel code, reason 2 no longer holds. Only reason 3 remains as a requirement for use of a fixed mapping. Some architectures in 4.4BSD remove this final constraint, so that they no longer need to provide a fixed mapping. They do so by copying the parent stack to the child-stack location. The machine-dependent code then traverses the stack, relocating the embedded stack and frame pointers. On return to the machine-independent fork code, no further references are made to local variables; everything just returns all the way back out of the kernel.

The location of the kernel stack in the user structure simplifies context switching by localizing all a process's kernel-mode state in a single structure. The kernel stack grows down from the top of the user structure toward the data structures allocated at the other end. This design restricts the stack to a fixed size. Because the stack traps page faults, it must be allocated and memory resident before the process can run. Thus, it is not only a fixed size, but also small; usually it is allocated only one or two pages of physical memory. Implementors must be careful when writing code that executes in the kernel to avoid using large local variables and deeply nested subroutine calls, to avoid overflowing the run-time stack. As a safety precaution, some architectures leave an invalid page between the area for the run-time stack and the page holding the other user-structure contents. Thus, overflowing the kernel stack will cause a kernel-access fault, instead of disastrously overwriting the fixed-sized portion of the user structure. On some architectures, interrupt processing takes place on a separate *interrupt stack,* and the size of the kernel stack in the user structure restricts only that code executed as a result of traps and system calls.

Context Switching

The kernel switches among processes in an effort to share the CPU effectively; this activity is called *context switching.* When a process executes for the duration of its time slice or when it blocks because it requires a resource that is currently unavailable, the kernel finds another process to run and context switches to it. The system can also interrupt the currently executing process to service an asynchronous event, such as a device interrupt. Although both scenarios involve switching the execution context of the CPU, switching between processes occurs *synchronously* with respect to the currently executing process, whereas servicing interrupts occurs *asynchronously* with respect to the current process. In addition, interprocess context switches are classified as *voluntary* or *involuntary.* A voluntary context switch occurs when a process blocks because it requires a resource that is unavailable. An involuntary context switch takes place when a process executes for the duration of its time slice or when the system identifies a higher-priority process to run.

Each type of context switching is done through a different interface. Voluntary context switching is initiated with a call to the *sleep*() routine, whereas an involuntary context switch is forced by direct invocation of the low-level context-switching mechanism embodied in the *mi_switch()* and *setrunnable()* routines. Asynchronous event handling is managed by the underlying hardware and is effectively transparent to the system. Our discussion will focus on how asynchronous event handling relates to synchronizing access to kernel data structures.

Process State

Context switching between processes requires that both the kernel- and user-mode context be changed; to simplify this change, the system ensures that all a process's user-mode state is located in one data structure: the user structure (most kernel state is kept elsewhere). The following conventions apply to this localization:

- **Kernel-mode hardware-execution state.** Context switching can take place in only kernel mode. Thus, the kernel's hardware-execution state is defined by the contents of the PCB that is located at the beginning of the user structure.
- **User-mode hardware-execution state.** When execution is in kernel mode, the user-mode state of a process (such as copies of the program counter, stack pointer, and general registers) always resides on the kernel's execution stack that is located in the user structure. The kernel ensures this location of user-mode state by requiring that the system-call and trap handlers save the contents of the user-mode execution context each time that the kernel is entered (see Section 3.1).
- The **process structure.** The process structure always remains resident in memory.
- **Memory resources.** Memory resources of a process are effectively described by the contents of the memory-management registers located in the PCB and by the values present in the process structure. As long as the process remains in

memory, these values will remain valid, and context switches can be done without the associated page tables being saved and restored. However, these values need to be recalculated when the process returns to main memory after being swapped to secondary storage.

Low-Level Context Switching

The localization of the context of a process in the latter's user structure permits the kernel to do context switching simply by changing the notion of the current user structure and process structure, and restoring the context described by the PCB within the user structure (including the mapping of the virtual address space). Whenever a context switch is required, a call to the *mi_switch()* routine causes the highest-priority process to run. The *mi_switch()* routine first selects the appropriate process from the scheduling queues, then resumes the selected process by loading that process's context from its PCB. Once *mi_switch()* has loaded the execution state of the new process, it must also check the state of the new process for a nonlocal return request (such as when a process first starts execution after a *fork;* see Section 4.5).

Voluntary Context Switching

A *voluntary* context switch occurs whenever a process must await the availability of a resource or the arrival of an event. Voluntary context switches happen frequently in normal system operation. For example, a process typically blocks each time that it requests data from an input device, such as a terminal or a disk. In 4.4BSD, voluntary context switches are initiated through the *sleep*() or *tsleep( )* routines. When a process no longer needs the CPU, it invokes *sleep( )* with a scheduling priority and a *wait channel.* The priority specified in a *sleep( )* call is the priority that should be assigned to the process when that process is awakened. This priority does not affect the user-level scheduling priority.

The wait channel is typically the address of some data structure that identifies the resource or event for which the process is waiting. For example, the address of a disk buffer is used while the process is waiting for the buffer to be filled. When the buffer is filled, processes sleeping on that wait channel will be awakened. In addition to the resource addresses that are used as wait channels, there are some addresses that are used for special purposes:

- The global variable l*bolt* is awakened by the scheduler once per second. Processes that want to wait for up to 1 second can sleep on this global variable. For example, the terminal-output routines sleep on l*bolt* while waiting for output-queue space to become available. Because queue space rarely runs out, it is easier simply to check for queue space once per second during the brief periods of shortages than it is to set up a notification mechanism such as that used for managing disk buffers. Programmers can also use the l*bolt* wait channel as a crude watchdog timer when doing debugging.

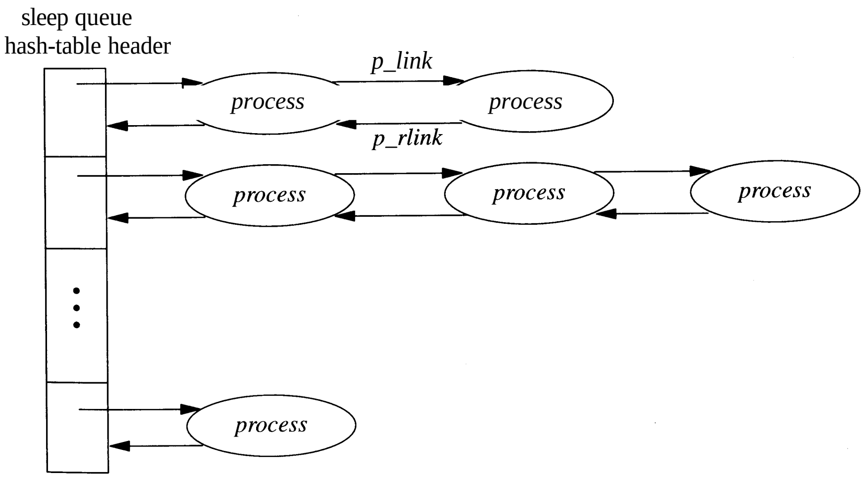
- When a parent process does a *wait* system call to collect the termination status of its children, it must wait for one of those children to exit. Since it cannot know which of its children will exit first, and since it can sleep on only a single wait channel, there is a quandary as to how to wait for the next of multiple events. The solution is to have the parent sleep on its own process structure. When a child exits, it awakens its parent's process-structure address, rather than its own. Thus, the parent doing the *wait* will awaken independent of which child process is the first to exit.

- When a process does a *sigpause* system call, it does not want to run until it receives a signal. Thus, it needs to do an interruptible sleep on a wait channel that will never be awakened. By convention, the address of the user structure is given as the wait channel.

Sleeping processes are organized in an array of queues (see Fig. 4.3). The *sleep()* and *wakeup( )* routines hash wait channels to calculate an index into the sleep queues. The *sleep*() routine takes the following steps in its operation:

1. Prevent interrupts that might cause process-state transitions by raising the hardware-processor priority level to *splhigh* (hardware-processor priority levels are explained in the next section).
2. Record the wait channel in the process structure, and hash the wait-channel value to locate a sleep queue for the process.
3. Set the process's priority to the priority that the process will have when the process is awakened, and set the SSLEEP flag.

Figure 4.3 Queueing structure for sleeping processes.



4. Place the process at the *end* of the sleep queue selected in step 2.

5. Call *mi_switch( )* to request that a new process be scheduled; the hardware priority level is implicitly reset as part of switching to the other process.

A sleeping process is not selected to execute until it is removed from a sleep queue and is marked runnable. This operation is done by the *wakeup( )* routine, which is called to signal that an event has occurred or that a resource is available. *Wakeup( )* is invoked with a wait channel, and it awakens *all* processes sleeping on that wait channel. All processes waiting for the resource are awakened to ensure that none are inadvertently left sleeping. If only one process were awakened, it might not request the resource on which it was sleeping, and so any other processes waiting for that resource would be left sleeping forever. A process that needs an empty disk buffer in which to write data is an example of a process that may not request the resource on which it was sleeping. Such a process can use any available buffer. If none is available, it will try to create one by requesting that a dirty buffer be written to disk and then waiting for the I/O to complete. When the I/O finishes, the process will awaken and will check for an empty buffer. If several are available, it may not use the one that it cleaned, leaving any other processes waiting for the buffer that it cleaned sleeping forever.

To avoid having excessive numbers of processes awakened, kernel programmers try to use wait channels with fine enough granularity that unrelated uses will not collide on the same resource. Thus, they put locks on each buffer in the buffer cache, rather than putting a single lock on the buffer cache as a whole. The problem of many processes awakening for a single resource is further mitigated on a uniprocessor by the latter's inherently single-threaded operation. Although many processes will be put into the run queue at once, only one at a time can execute. Since the kernel is nonpreemptive, each process will run its system call to completion before the next one will get a chance to execute. Unless the previous user of the resource blocked in the kernel while trying to use the resource, each process waiting for the resource will be able get and use the resource when it is next run.

A *wakeup( )* operation processes entries on a sleep queue from *front* to *back.* For each process that needs to be awakened, *wakeup( )*

1. Removes the process from the sleep queue

2. Recomputes the user-mode scheduling priority if the process has been sleeping longer than 1 second

3. Makes the process runnable if it is in a SSLEEP state, and places the process on the run queue if it is not swapped out of main memory; if the process has been swapped out, the *swapin* process will be awakened to load it back into memory (see Section 5.12); if the process is in a SSTOP state, it is left on the queue until it is explicitly restarted by a user-level process, either by a *ptrace* system call or by a *continue* signal (see Section 4.7)

If *wakeup( )* moved any processes to the run queue and one of them had a scheduling priority higher than that of the currently executing process, it will also request that the CPU be rescheduled as soon as possible.

The most common use of *sleep*() and *wakeup( )* is in scheduling access to shared data structures; this use is described in the next section on *synchronization.*

Synchronization

Interprocess synchronization to a resource typically is implemented by the association with the resource of two flags; a *locked* flag and a *wanted* flag. When a process wants to access a resource, it first checks the locked flag. If the resource is not currently in use by another process, this flag should not be set, and the process can simply set the locked flag and use the resource. If the resource is in use, however, the process should set the wanted flag and call *sleep 0* with a wait channel associated with the resource (typically the address of the data structure used to describe the resource). When a process no longer needs the resource, it clears the locked flag and, if the wanted flag is set, invokes *wakeupO* to awaken all the processes that called *sleep 0* to await access to the resource.

Routines that run in the bottom half of the kernel do not have a context and consequently cannot wait for a resource to become available by calling *sleep ( )* When the top half of the kernel accesses resources that are shared with the bottom half of the kernel, it cannot use the locked flag to ensure exclusive use. Instead, it must prevent the bottom half from running while it is using the resource. Synchronizing access with routines that execute in the bottom half of the kernel requires knowledge of when these routines may run. Although interrupt priorities are machine dependent, most implementations of 4.4BSD order them according to Table *43.* To block interrupt routines at and below a certain priority level, a critical section must make an appropriate *set-priority-level* call. All the set-priority-

Table 4.3 Interrupt-priority assignments, ordered from lowest to highest.

Name	**Blocks**
splO()	nothing (normal operating mode)
splsoftclock()	low-priority clock processing
spinet()	network protocol processing
spltty()	terminal multiplexers and low-priority devices
splbio()	disk and tape controllers and high-priority devices
splimp ( )	network device controllers
splclock( )	high-priority clock processing
splhigh()	all interrupt activity