4.2BSD and 4.3BSD as Examples of the UNIX System

JOHN S. QUARTERMAN, ABRAHAM SILBERSCHATZ, and JAMES L. PETERSON

Department of Computer Sciences, University of Texas, Austin, Texas 78712

This paper presents an in-depth examination of the 4.2 Berkeley Software Distribution, Virtual VAX-11 Version (4.2BSD), which is a version of the UNIX™ Time-Sharing System. There are notes throughout on 4.3BSD, the forthcoming system from the University of California at Berkeley. We trace the historical development of the UNIX system from its conception in 1969 until today, and describe the design principles that have guided this development. We then present the internal data structures and algorithms used by the kernel to support the user interface. In particular, we describe process management, memory management, the file system, the I/O system, and communications. These are treated in as much detail as the UNIX licenses will allow. We conclude with a brief description of the user interface and a set of bibliographic notes.

Categories and Subject Descriptors: C.2.4 [Computer-Communication Networks]: Distributed Systems—distributed applications; D.4.0 [Operating Systems]: General—UNIX; D.4.7 [Operating Systems]: Organization and Design—interactive systems; K.2 [History of Computing]: Software—UNIX

General Terms: Algorithms, Design, Human Factors, Performance, Reliability, Security

Additional Key Words and Phrases: Flexibility, portability, simplicity

INTRODUCTION

This paper presents an in-depth examination of the 4.2BSD operating system, the research UNIX® system developed for the Defense Advanced Research Projects Agency (DARPA) by the University of California at Berkeley. We have chosen 4.2BSD over UNIX System V (the UNIX system currently being licensed by AT&T) because concepts such as internetworking and demand paging are implemented in 4.2BSD but not in System V. Where 4.3BSD, the forthcoming system from Berkeley, differs functionally from 4.2BSD in the areas of interest, such differences are noted.

This paper is not a critique of the design and implementation of 4.2BSD or UNIX; it is an explanation. For comparisons of System V and 4.2BSD, see the literature, particularly the references given in Section 1.1, p. 380. Such comparisons are mostly beyond the scope of this paper.

The VAX® implementation is used because 4.2BSD was developed on the VAX, VAX, PDP, TOPS-20, and VMS are trademarks of Digital Equipment Corporation.

¹ UNIX is a trademark of AT&T Bell Laboratories.

² VAX, PDP, TOPS-20, and VMS are trademarks of Digital Equipment Corporation.

Chapter 14 of Operating Systems Concepts, Second Edition, by J. L. Peterson and A. Silberschatz (© 1985 by Addison-Wesley, Reading, Massachusetts) and this article were both derived from an earlier common manuscript by J. S. Quarterman. Consequently they share some text. Common portions are reprinted with the permission of Addison-Wesley.

Author's present address: James L. Peterson, MCC, 9430 Research Blvd., Austin, Texas 78759.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1986 ACM 0360-0300/85/1200-0379 $00.75

Computing Surveys, Vol. 17, No. 4, December 1985
CONTENTS

INTRODUCTION
1. OVERVIEW
   1.1 History
   1.2 Design Principles
2. PROCESSES
   2.1 User Interface
   2.2 Control Blocks
   2.3 CPU Scheduling
3. MEMORY MANAGEMENT
   3.1 Paging
   3.2 Swapping
4. FILE SYSTEM
   4.1 User Interface
   4.2 Implementations
   4.3 Data Structures on the Disk
   4.4 Layout and Allocation Policies
   4.5 Mapping a Pathname to an Inode
   4.6 Mapping a File Descriptor to an Inode
5. I/O SYSTEM
   5.1 Block Buffer Cache
   5.2 Raw Device Interfaces
   5.3 C-Lists
6. COMMUNICATIONS
   6.1 Signals
   6.2 Interprocess Communication
   6.3 Networking
   6.4 Distributed Systems
7. USER INTERFACE
   7.1 Shells and Commands
   7.2 Standard I/O
   7.3 Pipelines, Filters, and Shell Scripts
   7.4 The UNIX Philosophy
8. BIBLIOGRAPHIC NOTES
ACKNOWLEDGMENTS
REFERENCES

and that machine still represents a convenient point of reference, despite the recent proliferation of implementations on other hardware (such as the Motorola 68020 or National Semiconductor 32032). Also, details of implementation for non-VAX systems are usually proprietary to the companies that did them. And space does not permit examination of every implementation on every kind of hardware.

This paper is not a tutorial on how to use UNIX or 4.2BSD. It is assumed that the reader knows how to use the UNIX system. The presentation is closely limited to a technical examination of traditional operating system and networking concepts, most of which are implemented in the kernel. Students of operating systems and novice systems programmers (the intended readership) should find the organization and content appropriate.

The novice UNIX user will want to read Section 7 on the user interface before delving into the sections on kernel details. That section is as brief as possible, because the user interface and user programs in general are (regardless of their importance to the utility and popularity of the system) beyond the proper scope of this paper. Reading one of the several good books on using UNIX (see Section 8, Bibliographic Notes) would be good preparation for reading the paper.

The paper begins with a very brief overview of the history of the system and some description of the design philosophy behind it. The other sections cover process management, memory management, the file system, the I/O system, communications, and certain features of the user interface that distinguish the system. The paper concludes with a set of bibliographic notes.

1. OVERVIEW

This section is concerned with the history and design of the UNIX system, which was initially developed at Bell Laboratories as a private research project of two programmers. Its original elegant design and developments of the past fifteen years have made it an important and powerful operating system. We trace the history of the system E [Compton 1985; Ritchie 1978, 1984a, 1984b] and relate its design principles.

1.1 History

The first version of UNIX was developed at Bell Laboratories in 1969 by Ken Thompson to use an otherwise idle PDP-7. He was soon joined by Dennis Ritchie, and the two of them have since been the largest influence on what is commonly known as Research UNIX.

Ritchie, Thompson, and other early Research UNIX developers had previously worked on the Multics project [Peirce 1985], and Multics [Organick 1975] was a strong influence on the newer operating
Even the name UNIX is merely a pun on Multics, indicating that in areas where Multics attempted to do many things, UNIX tries to do one thing well. The basic organization of the file system, the idea of the command interpreter (the shell) being a user process, the use of a process per command, the original line-editing characters # and @, and many other things come directly from Multics.

Ideas from various other operating systems, such as Massachusetts Institute of Technology's CTSS, have also been used. The fork operation comes from Berkeley's GENIE (XCS-940) operating system.

The Research UNIX systems include UNIX Time-Sharing System, Sixth Edition (commonly known as Version 6), which was the first version widely available outside Bell Laboratories (in 1976) and ran on the PDP-11. (These version numbers correspond to the edition numbers of the UNIX Programmer's Manual that were current when the distributions were made.) Multiprogramming was added before Version 6, and after the system was rewritten in a high-level programming language, C [Kernighan 1978; Ritchie et al. 1978]. C was designed and implemented for this purpose by Dennis Ritchie. It is descended [Rosler 1984] from the language B, designed and implemented by Ken Thompson. B was itself descended from BCPL. C continues to evolve [Stroustrup 1984; Tuthill 1985a].

The first portable UNIX system was UNIX Time-Sharing System, Seventh Edition (Version 7), 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 system currently in development by the Research group at AT&T Bell Laboratories is UNIX Time-Sharing System, Eighth Edition (Version 8).

After the distribution of Version 7 in 1978, the Research group gave external distributions over to the UNIX Support Group (USG). USG had previously distributed such systems as UNIX Programmer's Work Bench (PWB) internally, and sometimes externally as well [Mohr 1985].

Their first external distribution after Version 7 was UNIX System III (System III), in 1982, which incorporated features of Version 7, 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 well as many features from PWB. USG released UNIX System V (System V) in 1983; it is largely derived from System III. The divestiture of the various Bell Operating Companies from AT&T has left AT&T in a position to market System V [Wilson 1985] aggressively. USG has metamorphosed into the UNIX System Development Laboratory (USDL), whose current distribution is UNIX System V Release 2 (V.2), released in 1984.

The ease with which the UNIX system can be modified has led to development work at numerous organizations such as Rand, Bolt, Beranek and Newman (BBN), the University of Illinois, Harvard, Purdue, and even DEC. But the most influential of the non-Bell Laboratories and non-AT&T UNIX development groups has been the University of California at Berkeley [McKusick 1985]. UNIX software from Berkeley is released in so-called Berkeley Software Distributions (BSD), hence the generic numbers 2BSD for the later PDP-11 distributions and 4BSD for the later VAX distributions.

Many of the features of the 4BSD terminal drivers are from TENEX/TOPS-20, and efficiency improvements have been made as a result of comparisons with VMS. The first Berkeley VAX UNIX work was the addition to 32V of virtual memory, demand paging, and page replacement in 1979 by Bill Joy and Ozalp Bagaoğlu to produce 3BSD. The large virtual memory space of 3BSD allowed the development of very large programs, such as Berkeley's own Franz Lisp. This memory management work convinced DARPA to fund Berkeley for the later development of a standard UNIX system for government use (4BSD).

One of the goals of this project was to provide support for the DARPA Internet networking protocols TCP/IP. This was done in a general manner, and it is possible to communicate among diverse network facilities, ranging from local networks (such as...
as Ethernets and token rings) to long-haul networks (such as DARPA's ARPANET).

It is sometimes convenient to refer to the Berkeley VAX UNIX systems following 3BSD as 4BSD, although there were actually several releases (indicated by decimal points in the release numbers), most notably 4.1BSD. 4BSD was the operating system of choice for VAXs from the beginning until the release of System III (1979–1982) and remains so for many research or networking installations. Most organizations would buy a 32V license and order 4BSD from Berkeley without ever bothering to get a 32V tape. Many installations inside the Bell System ran 4.1BSD (many still do, and many others run 4.2BSD).

The 4BSD work for DARPA was guided by a steering committee, which included many notable people from the UNIX and networking communities. 4.2BSD, first distributed in 1983, is the culmination of the original Berkeley DARPA UNIX project, although further research proceeds at Berkeley.

Berkeley was not the only organization involved in the development of 4.2BSD. Contributions (such as autoconfiguration, job control, and disk quotas) came from numerous universities and other organizations in Australia, Canada, Europe, and the United States. A few ideas, such as the `fcntl` system call, were taken from System V. (Licensing and pricing considerations have prevented the use of any actual code from System III or System V in 4BSD.) Not only are many contributions included in the distributions proper, but there is an accompanying set of user-contributed software, which is carried on the tapes containing the 4BSD distributions. The system was tested on the M68000-based workstation by Sun Microsystems, Inc., before its initial distribution. This simultaneous development contributed to the ease of further ports of 4.2BSD.

Berkeley accepts mail about bugs and their fixes at a well-known electronic address, and the consulting company mt.Xinu distributes a bug list compiled from such submissions. Many of the bug fixes may be incorporated in future distributions. There is constant discussion of UNIX in general (including 4.2BSD) in the DARPA Internet mailing list UNIX-WIZARDS, which appears on the USENET network as the news group net.unix-wizards; both the Internet and USENET are international in scope. There is another USENET news group dedicated to 4BSD bugs. While few ideas appear to be accepted by Berkeley directly from these lists and news groups (probably because of the difficulty of sifting through the sheer volume of submissions), discussions in them sometimes lead to new facilities being written that are later accepted.

Figure 1 is a sketch of the evolution of the several main branches of the UNIX system, especially those leading to 4.2BSD and System V [Chambers and Quarterman 1983; Uniejewski 1985]. The dates given are approximate, and there is no attempt to show all influences. Some of the systems named in the figure are not mentioned in the text, but are included to better show the relations among the ones that are discussed in the text.

We are aware at this writing of the imminent release of 4.3BSD and System V Release 2 Version 4. There are few functional changes in the kernel in 4.3BSD, although there are many performance improvements [Cabrera et al. 1985; Leffler et al. 1984; McKusick et al. 1985]. (Some of these 4.3BSD changes are noted in sections throughout this paper.) Although System V Release 2 Version 4 does introduce paging [Jung 1985; Miller 1984] (including copy-on-write and shared memory) to System V, there are few other functional changes.

Dozens of computer manufacturers, including almost all of those usually

Figure 1. UNIX history.
considered major by market share, have either announced or introduced computers that run the UNIX system or close derivatives, and numerous other companies sell related peripherals, software packages, support, training, documentation, or combinations of these. The hardware packages involved range from minis to mainframes to supercomputers. Most of these use ports of System V, 4.2BSD, or mixtures of the two, although there are still a variety of machines running software based on System III, 4.1BSD, and Version 7. There are even some Version 6 systems still in regular operation.

UNIX is also an excellent vehicle for academic study. For example, both the Tunis operating system [Holt 1983] and the Xinu operating system [Comer 1984] are based on the concepts of UNIX, but were developed explicitly for classroom study. Ritchie and Thompson were honored in 1983 by the ACM Turing award for their work on UNIX.

1.2 Design Principles

Unlike its most influential ancestor, Multics, UNIX was not designed by a joint project of several major institutions and did not have project goals and aspirations set out beforehand. In a series of papers presented at a prestigious professional conference. UNIX was, instead, orginated first by one programmer, Ken Thompson, and then another, Dennis Ritchie, as a system for their personal convenience, with no elaboration spelled out beforehand. This flexibility appears to have been one of the key factors in the development of the system. There were some design principles involved, however, even though they were not spelled out at the outset.

UNIX was designed by programmers for programmers. It has always been interactive. Multiple processes are supported, and it is easy for one process to create another process. There are standard and flexible ways of interconnecting the input and output of processes and otherwise coordinating several processes to do a task. It is not uncommon for a book typeset using the UNIX system to include both the source of programs written in a language such as C, FORTRAN, Pascal, or LISP, and the output of the same programs. The manuscript of the book itself may be used as the input of such programs, and their output may be statistics concerning the book. It is trivial for the programmer to set up a mechanism whereby a single word (e.g., "make") typed at a terminal causes the programs to be compiled and run and both their source and output to be typeset as part of the manuscript.

A main reason for this flexibility is the lack of numerous file types: There is only one data file type as far as the operating system is concerned. This is a sequence of bytes, which may be accessed either randomly or sequentially. There are no "access methods" and no "control blocks" in the data space of a UNIX user process, and there are few limitations on what a process's data space may be used for. The interface to the file system is very simple: A file is referred to by a character string for opening and by an integer for further manipulation.

Files are grouped in directories, which essentially form a tree-structured hierarchy. Users may create directories as easily as ordinary files.

It is not hard to build elaborate database access mechanisms on top of UNIX's one simple file type, as the half-dozen or more readily available databases for UNIX (starting with INGRES, which is distributed with 4BSD) attest.

Devices and ordinary files are treated as similarly as possible. Thus device dependencies or peculiarities are kept in the kernel as much as possible, and even in the kernel most of them are segregated in the device drivers.

A process may be informed of an exceptional condition (perhaps by another process) by means of a signal. True interprocess communication and networking may be accomplished in 4.2BSD by the use of sockets.

The size constraints of the PDP-11 (and earlier computers used for UNIX) have forced a certain elegance. Whereas other
4.2BSD and 4.3BSD as Examples of the UNIX System

<table>
<thead>
<tr>
<th>(the users)</th>
</tr>
</thead>
<tbody>
<tr>
<td>shells and commands</td>
</tr>
<tr>
<td>compilers and interpreters</td>
</tr>
<tr>
<td>system libraries</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>system call interface to the kernel</th>
</tr>
</thead>
<tbody>
<tr>
<td>signals</td>
</tr>
<tr>
<td>terminal handling</td>
</tr>
<tr>
<td>character I/O system</td>
</tr>
<tr>
<td>terminal drivers</td>
</tr>
<tr>
<td>file system</td>
</tr>
<tr>
<td>swapping</td>
</tr>
<tr>
<td>block I/O system</td>
</tr>
<tr>
<td>disk and tape drivers</td>
</tr>
<tr>
<td>CPU scheduling</td>
</tr>
<tr>
<td>page replacement</td>
</tr>
<tr>
<td>demand paging</td>
</tr>
<tr>
<td>virtual memory</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>kernel interface to the hardware</th>
</tr>
</thead>
<tbody>
<tr>
<td>terminal controllers</td>
</tr>
<tr>
<td>terminals</td>
</tr>
<tr>
<td>device controllers</td>
</tr>
<tr>
<td>disks and tapes</td>
</tr>
<tr>
<td>memory controllers</td>
</tr>
<tr>
<td>physical memory</td>
</tr>
</tbody>
</table>

systems have elaborate algorithms for dealing with pathological conditions, UNIX just does a controlled crash (a panic), and tries to prevent rather than cure such conditions. Whereas other systems would use brute force or macro expansion, UNIX mostly has had to have developed more subtle, or at least simpler, approaches.

In some instances, such as networking, PDP-11 size constraints unfortunately had the opposite effect. The original UNIX Version 6 ARPANET software was split into a kernel part and a part that ran as a user process, purely because of size constraints. This entailed not only performance penalties but also led to a rather convoluted design. The 4.2BSD networking code does not suffer from this, since it runs on processors (VAX, M68000, NS16032, etc.) that have a reasonably sized address space. The PDP-11 ports of this code require extensive kernel overlays.

Virtual memory and paging were not implemented on the PDP-11 because of the small number and huge size of the pages allowed by the hardware. Thus early versions of the INGRES database system ran as multiple (six or seven) processes, and Franz Lisp, with its need for huge data spaces in a single process, did not develop until the VAX permitted paging in 3BSD.

Even though some UNIX systems now try to do some things that require large address spaces, the size constraints imposed during the early development of the system have had, on the whole, a beneficial effect.

From the beginning, UNIX development systems have had all the UNIX sources available on line, and the developers have used the systems under development as their primary systems. This has greatly facilitated discovering deficiencies and their fixes, as well as new possibilities and their implementations.

Facilities for program development have always been a high priority. Such facilities include the program make (which may be used to determine which of a collection of program source files need to be compiled and then compile them) and the Source Code Control System (SCCS) (which is used to keep successive versions of files available without having to store the entire contents of each step).

The availability of sources for the operating system has also encouraged the plethora of UNIX variants existing today, but the benefits have outweighed the disadvantages. If something is broken, it can be fixed at a local site, rather than having to wait for the next release of the system. Such fixes, as well as new facilities, may be incorporated into later distributions. Binary licenses are becoming more popular with the growing number of small, inexpensive, UNIX systems, however.

The UNIX operating system may be considered for convenience of exposition to be layered roughly as depicted in Figure 2.

---

Figure 2. Layers of the UNIX system.
Everything below the system call interface and above the physical hardware is the kernel [Thompson 1978]. This paper is mostly concerned with the kernel, since that is where most of the traditional operating systems issues are addressed:

The kernel is the only UNIX code that cannot be substituted by a user to his own liking. For this reason, the kernel should make as few real decisions as possible. This does not mean to allow the user a million options to do the same thing. Rather, it means to allow only one way to do one thing, but have that way be the least-common divisor of all the options that might have been provided. [Thompson 1978, p. 193]

This is especially noticeable in the design of the system call interface [Joy et al. 1983]: "Throughout, simplicity has been substituted for efficiency. Complex algorithms are used only if their complexity can be localized" [Thompson 1978, p. 1932].

2. PROCESSES

In this section we describe how user processes are created and manipulated by other user processes, including the layout of a process’s address space. Then the kernel control blocks that keep track of processes are described. Finally, an overview of the CPU scheduler and its event mechanism is given [Thompson 1978].

2.1 User Interface

A process is a program in execution. To execute a new program, a new process is first produced by the fork system call, creating two almost identical processes, each with a copy of the original data space. Then the exec primitive may be used by one process to replace its virtual memory space with that for a new program (read from a file). A process may choose to terminate by using the exit system call, and its parent process may wait for that event by using the wait system call. Figure 3 shows two common scenarios for the use of these system calls.

Processes are named by their process identifier (pid), which is an integer. The fork system call returns the process identifier of the child process to the parent process, and returns zero in the child process; this is how a program can determine in which process it is running after a fork. The wait system call provides the process identifier of the child that terminated so the parent can tell which of possibly several children it was.

From the viewpoint of the calling process, one may liken fork and wait to a subroutine call and return, whereas exec is more like a goto.

The simplest form of communication between processes is by pipes. Pipes provide a reliably delivered byte stream between two processes. They may be created before the fork, and their end points may then be set up between the fork and the exec.

All user processes are descendants of one original process, which is called init and has process identifier 1. On each terminal port available for interactive use, init forks (with the fork system call) a copy of itself, which attempts to open the port for reading and writing. This new process has a new process identifier. The open succeeds when a directly connected terminal is turned on or a telephone call is accepted by a dial-up modem. Then the init process executes (with the exec system call) a program called getty.

Getty initializes terminal line parameters and prompts the user to type a login name, which getty collects. Getty then executes a program called login, passing the login name as an argument. Login prompts the user for a password, and collects it as the user types it. Login determines whether the user is allowed to log in by encrypting the typed password and comparing it with an encrypted string found according to the login name in the file /etc/passwd. If the comparison is successful, login sets the numeric user identifier (uid) of the process to that of the user logging in and executes a shell, or command interpreter. (The pathname of the shell and the user identifier are also found in /etc/passwd according to the user’s login name.) This shell is what the user ordinarily communicates with for the rest of the login session. The shell itself forks subprocesses for the commands that the user tells it to execute.
The same process identifier is used successively after the fork by the child init process, by getty, by login, and finally by the shell. When the user logs out, the shell dies and the original init process (process identifier 1) waits on it. After the wait succeeds, the process identifier formerly used by the shell may be reassigned by the kernel to a new process.

The user identifier is used by the kernel to determine the user’s permissions for certain system calls, especially those involving file accesses. There is also a group identifier (gid), which is used to provide similar privileges to a collection of users. In 4.2BSD, a user’s processes may be in several groups simultaneously. The login process puts the user’s shell in all the groups permitted to the user by the files /etc/passwd and /etc/group.

There are actually two user identifiers used by the kernel: The effective user identifier is the one used to determine file access permissions, and the real user identifier is used by some programs to determine who the original user was before the effective user identifier was set by a setuid. If the file being executed by exec has setuid indicated, the effective user identifier of the process is set to the user identifier of the owner of the file, while the real user identifier is left as it was. This allows certain processes to have more than ordinary privileges while still being executable by ordinary users. This setuid idea is patented by Dennis Ritchie [1979a] and is one of the most distinctive features of UNIX. For groups there is a similar distinction for effective and real group identifiers, and a similar setgid feature.

Every process has both a user and a system phase, which never execute simultaneously. Most ordinary work is done by the user process, but when a system call is
done, it is the system process, which has a different stack than the user process, that performs the system call.

The virtual address space of a user process is divided into text (executable instructions), data, and stack segments. The data and stack segments are always in the same address space, but may grow separately and, on most machines, in opposite directions: On a VAX the stack grows down as the data grow up toward it. The text segment is usually not writable, so that one copy may be shared among several processes.

How a process's virtual address space is mapped into physical main or secondary memory varies greatly from system to system but is usually transparent to the user.

2.2 Control Blocks

There are no system control blocks accessible in a user process's virtual address space, but there are such control blocks in the kernel associated with the process. Some of these control blocks are diagramed in Figure 4.

The most basic data structure associated with processes is called the process structure. There is an array of these, whose length is defined at system link time. Each process structure contains everything that is necessary to know about a process when it is swapped out, such as its unique process identifier (an integer), scheduling information (like the process's priority), and pointers to other control blocks. The process structures of running processes are kept linked together by the scheduler in a doubly linked list, and there are pointers from each process structure to the process's parent, its youngest living child, and various other relatives of interest, such as a list of processes sharing the same text.

Every process with sharable text (almost all, under 4.2BSD) has a pointer from its process structure to a text structure. The text structure records how many processes are using the text segment, a pointer into a list of their process structures, and where the page table for the text segment can be found on disk when it is swapped. The text structure itself is always resident in main memory: An array of such structures is allocated at system link time.

The page tables contain information on how the process's virtual address space is mapped to physical memory. When a process is in main memory, its page tables may be found by a pointer in the process structure. When the process is swapped, the process structure contains instead the address of the process on the swap device. There is no special separate page table for the text segment when it is resident in main memory; every process sharing it has entries for its pages in the process's page table. (Nonpaging UNIX systems ordinarily have the text structure point directly to the text segment, both in main memory and on the swap device.)

Information about the process that is needed only when the process is resident (i.e., information that is not in the process structure) is kept in the user structure (or u structure). A copy of the VAX hardware process control block is kept here for saving the process's general registers, stack pointer, program counter, and page table base registers when the process is not running. There is space to keep system call parameters and return values. All user and group identifiers associated with the process (not just the effective user identifier kept in the process structure) are kept here. Signals, timers, and quotas have data structures here. Of more obvious relevance to the ordinary user, the current directory is maintained here, and open files are kept track of. The kernel stack for the process (i.e., the stack of the system process phase) is also in the user structure.

On the VAX, the user structure is mapped into the high end of user address space, just above the process's stack. The user structure of the running process is still directly accessible to the kernel, however, since kernel address space is VAX system space, which is the high half of the whole 32-bit virtual address space, whereas process address space is the low half.

When a fork system call is done, a new process structure with a new process identifier is allocated for the child process, and the user structure is copied. The copying preserves open file descriptors, user and
group identifiers, signal handling, and most similar properties of a process. There is ordinarily no need for a new text structure, as the processes share their text; the appropriate counters and lists are merely updated. A new page table is constructed, and new main memory is allocated for the data and stack segments of the child process. (If enough memory cannot be found for the page table, the process is swapped until there is enough.)

The *vfork* system call does not copy the data and stack to the new process; rather the new process simply shares the page table of the old one. A new user structure and a new process structure are still created. A common use of this system call is by a shell to execute a command and wait on its completion. The parent process uses *vfork* to produce the child process. The child process only wishes to use *exec* to change its virtual address space completely into that of a new program, so that there is no need for a complete copy of the parent process. Such data structures as are necessary for manipulating pipes may be kept in registers between the *vfork* and the *exec*. Files may be closed in one process without affecting the other process, since the kernel data structures involved depend on the user structure, which is not shared. The kernel suspends the parent process until the child calls *exec* or exits.

When the parent process is large, *vfork* can produce substantial savings in system CPU time. It is a rather dangerous system call, however, since any memory change by the child process occurs in both processes until the *exec* occurs. An alternative is to share all pages by duplicating the page table, but to mark the entries of both page tables as *copy-on-write*. The hardware protection bits are set to trap any attempt to write in these shared pages. If such a trap occurs, a new frame is allocated and the shared page is copied to the new frame. The page tables are adjusted to show that this page is no longer shared (and therefore need no longer be write protected), and execution can resume. Hardware bugs with the VAX-11/750 prevented 4.2BSD from including a *copy-on-write* fork operation (although Tektronix has since implemented it).

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

2.3 CPU Scheduling

CPU scheduling in UNIX is designed to benefit interactive jobs. Processes are given small CPU time slices by an algorithm that reduces to round-robin for CPU-bound jobs, although there is a priority scheme. There is no preemption of one process by another when running in the kernel. A process may relinquish the CPU because it is waiting on I/O or because its time slice has expired.

Every process has a scheduling priority associated with it; the lower the numerical priority, the more likely the process is to run. Processes doing disk I/O or other important tasks have negative priorities and cannot be interrupted by signals. Ordinary user processes have positive priorities and thus are all less likely to be run than any system process, although user processes may have precedence over one another. The nice command may be used to affect this precedence according to its numerical priority argument.

The more CPU time a process accumulates, the lower (more positive) its priority becomes. The reverse is also true (process aging is employed to prevent starvation). Thus there is negative feedback in CPU scheduling, and it is difficult for a single process to take all CPU time.

Older UNIX systems used a 1-second quantum for the round-robin scheduling. 4.2BSD reschedules processes every 0.1 second and recomputes priorities every second. The round-robin scheduling is accomplished by the timeout mechanism, which tells the clock interrupt driver to call a certain subroutine after a specified interval. The subroutine to be called in this case causes the rescheduling and then resubmits a timeout to call itself again. The priority recomputation is also timed by a subroutine that resubmits a timeout for itself.

When a process chooses to relinquish the CPU, it goes to sleep on an event. The kernel primitive used for this is called sleep (not to be confused with the user-level library routine of the same name). It takes an argument that is by convention the address of a kernel data structure related to an event the process wants to occur before it is awakened. When the event occurs, the system process that knows about it calls wakeup with the address corresponding to the event, and all processes that had done a sleep on the same address are put in the queue to be scheduled to be run.

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

The process that actually does run is chosen by the scheduler, effectively at random. Sleep, however, also takes a second argument, which is the scheduling priority to be used for this purpose. The priority argument, if negative, also prevents the process from being prematurely awakened from its sleep by some exceptional event, such as a signal.

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

There are race conditions involved in the event mechanism. If a process checks to see if a flag in memory has been set by an interrupt routine, and then sleeps on an event, the process may sleep forever, because the event may occur between the test of the flag and the completion of the sleep primitive. This is prevented by raising the hardware processor priority during the critical section so that no interrupts can occur, and thus only the process desiring the event can run until it is sleeping. Hardware pro-
Processor priority is used in this manner to protect critical regions throughout the kernel and is the greatest obstacle to porting UNIX to multiple processor machines [Bach and Buroff 1984; Gobel and Marsh 1981] (although not a large enough obstacle to prevent such ports, as many have been done [Beck and Kasten 1985; Bell 1985]).

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

CPU scheduling, swapping, and paging interact: The lower the priority of a process, the more likely it is that its pages will be paged out and that it will be swapped in its entirety.

The CPU scheduling just described provides short-term scheduling, although the negative feedback property of the priority scheme provides some more long-term scheduling, since it largely determines the long-term job mix. Swapping has an intermediate range scheduling effect, although on a machine with sufficient memory, swapping occurs rarely.

3. MEMORY MANAGEMENT

Much of UNIX's early development was done on the PDP-11. That computer has only eight pages in its virtual address space, and those are of 8192 bytes each. (The larger machines like the PDP-11/70 allow separate instruction and address spaces, which effectively double the address space and number of pages, but this is still not much.) This large granularity is not conducive to sophisticated paging algorithms. The kernel was also restricted by the same virtual address space limitations, and so there was also little room to implement such algorithms. (In fact, the kernel was even more severely constrained as a result of dedicating part of one data page to interrupt vectors, a whole page to point at the per-process system page, and yet another for the UNIBUS register page.) Further, on the small PDP-11s, total physical main memory was limited to 256 kbytes. Thus UNIX swapped.

The advent of the VAX with its 512-byte pages and multigigabyte virtual address space permitted paging, although the variety of practical algorithms was limited by the lack of a hardware reference bit. Berkeley introduced paging to UNIX with 3BSD [Babaoglu and Joy 1981].

3.1 Paging

VAX 4.2BSD is a demand-paged virtual memory system. External fragmentation of memory is minimized by paging. (There is internal fragmentation, but this is negligible with a reasonably small page size.) Swapping is kept to a minimum because more jobs can be kept in main memory since not all of any job has to be resident.

Because the VAX has no hardware memory management page reference bit, many memory management algorithms (such as page fault frequency) are unusable. 4.2BSD used a modified Global Clock Least Recently Used (LRU) algorithm. A software clock hand linearly and repeatedly sweeps all frames of main memory that are available for paging. The missing hardware reference bit is simulated by marking a page as invalid (reclaimable) when the clock hand sweeps over it. If the page is referenced before the clock hand next reaches it, a page fault occurs, and the page is made valid again. But if the page has not been referenced when the clock hand reaches it again, it is reclaimed for other use. Various software conditions are also checked before a page is marked invalid, and many other measures are taken to reduce the high overhead of this use of page faults to simulate reference bits.

There is a way for a process to turn off the reference bit simulation for its pages, thus getting effectively random page replacement: Franz Lisp uses this feature during garbage collection. There are checks to make sure that a process's number of valid data pages does not fall too low, and to keep the page-out device from being flooded with requests. There is also a mechanism by which a process may limit the amount of main memory that it uses. There
is no provision, however, for a process to lock a specific set of its pages in main memory, because this is considered inappropriate in the research environments in which 4.2BSD is commonly used, where the paging characteristics of programs under development cannot be readily predicted and equitable sharing of all system resources among all processes is important. In general, page locking is less appropriate in a system like UNIX, where processes are numerous and readily created, than in a system like VMS.

All main memory frames available for paging are represented by the core map or cmap. This map records the disk block corresponding to a frame that is in use, an indication of what process page a frame in use is mapped into, and a free list of frames that are not mapped into process pages.

Paging in, that is, demand paging, is done in a straightforward manner. When a process needs a page and the page is not mapped into a memory frame, a page fault is produced by the hardware. This causes the kernel to allocate a frame of main memory, map it into the appropriate process page, and read the proper data into it from disk. Such pages come initially from the process's object file. That is, processes are not prepaged whole; instead, they are demand loaded.

There are some complications. If the required page is still in the process's page table but has been marked invalid by the last pass of the clock hand, it can be marked valid and used without any I/O transfer. Pages can be similarly retrieved from the memory free list. Every process's text segment is by default a shared read-only text segment. This is practical with paging, since there is no external fragmentation, and the swap space gained by sharing more than offsets the overhead involved, since the kernel virtual space is large. When the last process sharing a text segment dies, the disk block information and data for its text frames are left in the frames when they are put in the free list, so that if a new process sharing the same text executes soon, frames can be mapped into many of its text pages without being retrieved from disk. Processes that have been swapped out may also find some of their pages still in the free list when they are swapped back in.

UNIX processes have their data logically divided into initialized data and uninitialized data (bss). Uninitialized data are all zero at process execution and are represented in the process's object file only by an indication of the file's size. Pages for such uninitialized data do not have to be read from disk. A frame is found, mapped, and zero filled. New stack pages also do not need to be transferred from disk.

If the page has to be fetched from disk, it must be locked for the duration. Once the page is fetched and mapped properly, it must not be unlocked if raw physical I/O is being done on it.

Paging out, that is, the page replacement algorithm, is more interesting. The software clock hand cycles through cmap, checking conditions on each frame as it passes. If the frame is empty (has no process page mapped into it), it is left untouched, and the hand sweeps to the next page. If I/O is in progress on it, or some other software condition indicates the page is actually being used, then the frame is also left untouched. But if the page is not in use, the corresponding process page table entry is located. If the entry is valid, it is made invalid but reclaimable. If the entry is invalid (because the last sweep of the clock hand made it so), it is reclaimed. If the page has been modified (the VAX does have a dirty bit), it must first be written to disk before the frame is added to the free list.

The LRU clock hand is implemented in the pagedaemon, which is process 2 (the scheduler is process 0 and init is process 1). The pagedaemon's purpose is to keep the memory frame free list large enough that paging demands on memory will not exhaust it. This process spends most of its time sleeping, but a check is done several times per second (scheduled by a timeout) to see whether action is necessary, and process 2 is awakened if so. Whenever the number of free frames falls below a threshold (lotsfree), the process is awakened; thus,
if there is always a lot of free memory, the pagedaemon imposes no load on the system because it never runs.

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

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

The parameter lotsfree is usually one-fourth of the memory in the map the clock hand sweeps, and desfree and minfree are usually the same in different systems, but are limited to fractions of available memory.

Many peaks of memory demand caused by exec in a swapping system are smoothed by demand paging processes rather than by preloading them. Other peaks caused by the address space copying of fork (especially as used by the shells) are smoothed by the substitution of vfork (see Section 2.2) in many instances.

For I/O efficiency, the VAX 512-byte hardware pages are too small, so they are clustered in groups of two so that all paging I/O is actually done in 1024-byte (or larger) chunks. For still greater efficiency, adjacent frames that are ready to be paged in or out at the same time are done in the same I/O operation; this is called klustering.

On a page fault, several additional pages that are adjacent in both physical and process virtual space may also be read in on one disk transfer. Such prepaged frames are put on the bottom of the free list, so that they are likely to remain on the free list long enough for the process to claim them if they are needed. Since many processes may not actually use such prepaged frames, they are not immediately mapped into the process’s pages, because they would then stay there until the clock hand passed, even if they were initially marked invalid.

Large VAX systems now commonly have 8, 16, or even more megabytes of main memory. This leads to a problem with the reference bit simulation. The 4.2BSD clock hand may take a long time (minutes, or even tens of minutes) to complete a cycle around such large amounts of memory. Thus the second encounter of the hand with a given page (when it is checked to see if it is still valid) has little relevance to the first encounter (when the page is marked invalid), and the pagedaemon will have difficulty finding reclaimable page frames. 4.3BSD uses a second clock hand, which follows behind the first at a shorter distance than a complete cycle (see Figure 5). The front hand marks pages invalid, while the back hand reclaims frames whose pages are still invalid. The proper interval between the two hands is still a matter for research.

### 3.2 Swapping

Pre-3BSD UNIX systems used swapping exclusively to handle memory contention among processes: If there was too much contention, some processes were swapped

Computing Surveys, Vol. 17, No. 4, December 1985
out. Also, a few large processes could receive many small processes out of memory, and a process larger than nonkernel main memory could not be run at all. The system data segment (the u structure and kernel stack) and the user data segment (text, if non-sharable; data; and stack) were kept in contiguous main memory for swap transfer efficiency, so external fragmentation of memory could be a serious problem.

Allocation of both main memory and swap space was done first fit. When the size of a process increased (owing to either stack expansion or data expansion), a new piece of memory big enough for the whole process was allocated, the process copied, the old memory freed, and the appropriate tables updated. (Some attempt was made in some systems to find memory contiguous to the end of the current piece to avoid some copying, but the stack would still have to be copied on machines where it grew downward.) If no single large enough piece of main memory was available, the process was swapped out in such a way that it would be swapped back in with the new size.

There was no need to swap a sharable text segment out (more than once), because it was never writable, and there was no need to read in a text segment for a process when another instance was already in core. This was one of the main reasons for shared text: less swap traffic. The other reason was that multiple processes using the same text segment required less main memory. However, it was not practical on most machines for every process to have a shared text segment, since those segments required extra overhead in the kernel and also promoted external fragmentation of both main memory and swap space.

Decisions on which processes to swap in or out were made by the scheduler process, process 0 (also known as the swapper process). The scheduler woke up at least once every 4 seconds to check for processes to be swapped in or out. A process was more likely to be swapped out if it was idle, had been in main memory a long time, or was large; if no easy candidates were found, other processes were picked by age. A process was more likely to be swapped in if it had been swapped out a long time or was small. There were checks to prevent thrashing, basically by not letting a process be swapped out if it had not been in core a certain amount of time.

Many UNIX systems still use the swapping scheme described above. All AT&T USG/USDL systems, including System V, do. All Berkeley VAX UNIX systems, on the other hand, including 4.2BSD, depend primarily on paging for memory contention management and only secondarily on swapping. A scheme very similar in outline to the traditional one is used to determine what processes get swapped in or out, but the details differ and the influence of swapping is less.

If the paging situation is pathological, then jobs are swapped out as described above until the situation is acceptable. Otherwise, the process table is searched for a process deserving to be brought in (determined by how small it is and how long it has been swapped). The amount of memory the process will need is some fraction of its total virtual size, up to one-half if it has...
been swapped a long time. If there is not enough memory available, processes are swapped out until there is. The processes to be swapped out are chosen according to their being the oldest of the biggest jobs in core, or having been idle for a while, or, in case of desperation, simply being the oldest in core.

The age preferences used with swapping guard against thrashing, but paging does so more effectively. Ideally, given paging, processes will not actually be swapped out whole unless they are idle, since each process will only need a small working set of pages in main memory at any one time, and the pagedaemon will reclaim unused pages for use by other processes, so that most runnable processes will never be completely swapped out.

There is a swap allocation map, dmap, for each process's data and stack segment. Swap space is allocated in pieces that are multiples of a constant minimum size (e.g., 32 pages) and a power of 2. There is a maximum, which is determined by the size of the swap space partition on the disk. If several logical disk partitions may be used for swapping, they should be the same size for this reason. The several logical disk partitions should be on separate disk arms to minimize disk seeks.

4. FILE SYSTEM

Data are kept in files, which are organized in directories. Files, directories, and related data structures comprise the file system.

4.1 User Interface

An ordinary file in UNIX is a sequence of bytes. Different programs expect various levels of structure, but the kernel does not impose structure on files. For instance, the convention for text files is lines separated by a single new-line character (which is the line feed character in ASCII), but the kernel knows nothing about this convention.

Files are organized in directories in a hierarchical tree structure. Directories themselves files that contain information on how to find other files. A pathname to a file is a text string that identifies a file by specifying a path through the file system to the file. Syntactically it consists of individual file name elements separated by slash characters. In the example

/alpha/beta/gamma

the first slash indicates the root of the whole directory tree, called the root directory. The next element, alpha, is a subdirectory of the root, beta is a subdirectory of alpha, and gamma is a file in the directory beta. Whether gamma is an ordinary file or a directory itself cannot be told from the pathname syntax.

There are two kinds of pathnames, absolute pathnames and relative pathnames. Absolute pathnames start at the root of the file system and are distinguished by a slash at the beginning of the pathname; the previous example (/alpha/beta/gamma) is an absolute pathname. Relative pathnames start at the current directory, which is a property of the process accessing the pathname. The example

gamma

indicates a file named gamma in the current directory, which might or might not be /alpha/beta.

A file may be known by more than one name in one or more directories. Such multiple names are known as links and are all treated as equally important by the operating system. In 4.2BSD there is also the idea of a symbolic link, which is a file containing the pathname of another file. The two kinds of links are also known as hard links and soft links, respectively. Soft links, unlike hard links, may point to directories and may cross file system boundaries (see below).

The filename "." in a directory is a hard link to the directory itself, and the filename ".." is a hard link to the parent directory. Thus, if the current directory is /alpha/beta, then .. refers to /alpha and . refers to /alpha/beta itself.

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

Computing Surveys, Vol. 17, No. 4, December 1985
Figure 6. Example directory structure.
Figure 6 shows some directories, ordinary files, and special files that might appear in a real file system. The root of the whole tree is /, /vmunix is the binary object of the 4.2BSD kernel, which is used at system boot time. /etc/init is the executable binary of process 1, which is the ancestor of all other user processes. System maintenance commands and basic system parameter files appear in the directory /etc. Examples are /etc/passwd (which defines a user's login name, numerical identifier, login group, home directory, and command interpreter, and which contains the user's encrypted password) and /etc/group (which defines names for group identifiers and determines what users are in many groups).

Ordinary commands appear in the directories /bin (commands essential to system operation), /usr/bin (other commands, in a separate directory for historical reasons), /usr/ucb (commands from the University of California, Berkeley), and /usr/local (commands, added at the local site, which did not come with the 4.2BSD distribution). Library files appear in /lib (e.g., compiler passes and /lib/libc.a, which is the C library, containing utility routines and system call interfaces), /usr/lib (most text processing macros), and /usr/local/lib (locally added libraries). System parameter files that are useful to user programs appear in /usr/include. For instance, /usr/include/stdio.h contains parameters related to the standard I/O system (see Section 7.2).

Device special files (such as /dev/console, the interface to the system console terminal) ordinarily appear in /dev.

Finally, private user files appear under users' login directories, which are grouped in directories whose names vary from site to site. In the figure, /u0/avi would be a login directory for the user whose login name is avi.

A file is opened by the open system call, which takes a pathname and a permission mode (indicating whether the file should be open for reading, writing, or both) as arguments. This system call returns a small integer, called a file descriptor. This file descriptor may then be passed to a read or write system call (along with a pointer to a buffer and its size) to perform data transfers to or from the disk file or device. A file is closed by passing its file descriptor to the close system call. Each read or write updates the current offset into the file, which is used to determine the position in the file for the next read or write. This position can be set by the lseek system call. There is an additional system call, ioctl, for manipulating device parameters.

A new, empty file may be created by the creat system call, which returns a file descriptor as for open. New hard links to an existing file may be created with the link system call, and new soft links with the symlink system call. Either may be removed by the unlink system call. When the last hard link is removed (and the last process that has the file open closes it), the file is deleted. There may still be a symbolic link pointing to the nonexistent file: Attempts to reference such a link will produce an error.

Device special files may be created by the mknod system call. Directories are created by the mkdir system call (whose functions were accomplished in pre-4.2BSD systems by the mkdir command using the mknod and link system calls). Directories are removed by rmdir (or, in pre-4.2BSD systems, by the rmdir command using unlink several times). The current directory is set by the chdir system call.

The chown system call sets the owner and group of a file and chmod changes protection modes. Stat applied to a file name or fstat applied to a file descriptor may be used to read back such properties of a file. In 4.2BSD, the rename system call may be used to rename a file; in previous systems this was done by link and unlink.

The user ordinarily only knows of one file system, but the system may know this one virtual file system is actually composed of several physical file systems, each on a different device. A physical file system may not span multiple hardware devices. Since most physical disk devices are divided into several logical devices, there may be more than one file system per physical device, but no more than one per logical device. One file system, the root file system, is always available. Others may be mounted, that is, integrated into the directory...
hierarchy of the root file system. References to a directory that has a file system mounted on it are transparently converted by the kernel into references to the root directory of the mounted file system.

4.2 Implementations

The system call interface to the file system is simple and well defined. This has allowed the implementation of the file system itself to be changed without significant effect on the user. This happened with Version 7: The size of inodes doubled, the maximum file and file system sizes increased, and the details of free list handling and superblock information changed. Also at the time seek (with a 16-bit offset) became lseek (with a 32-bit offset) to allow for simple specification of offsets into the larger files then permitted. Few other changes were visible outside the kernel.

In 4.0BSD the size of the blocks used in the file system was increased from 512 to 1024 bytes. Although this entailed increased internal fragmentation of space on the disk, it allowed a factor-of-2 increase in throughput, due mainly to the greater number of data accessed on each disk transfer. This idea was later adopted by System V, along with a number of other ideas, device drivers, and programs.

The 4.2BSD file system implementation [McKusick et al. 1984] is radically different from that of Version 7 [Thompson 1978]. This reimplementation was done primarily for efficiency and robustness, and most of the changes done for those reasons are invisible outside the kernel. There were some new facilities introduced at the same time, such as symbolic links and long filenames, which are visible at both the system call and the user levels. Most of the changes required to implement these were not in the kernel, but rather in the programs that use them.

4.3 Data Structures on the Disk

The virtual file system seen by the user is supported by a data structure on a mass storage medium, usually a disk. This data structure is the file system. All accesses to it are buffered through the block I/O system, which is described in Section 5.1; for the moment, we consider only what resides on the disk.

A physical disk drive may be partitioned into several logical disks, and each logical disk may contain a file system. A file system cannot be split across more than one logical disk. The actual number of file systems on a drive varies according to the size of the disk and the purpose of the computer system as a whole. Some partitions may be used for purposes other than supporting file systems, such as swapping.

The first sector on the logical disk is the boot block, containing a primary bootstrap program, which may be used to call a secondary bootstrap program residing in the next 7.5 kbytes.

The data structures on the rest of the logical disk are organized into cylinder groups, as shown in Figure 7. Each of these occupies one or more consecutive cylinders of the disk so that disk accesses within the cylinder group require minimal disk head movement. Every cylinder group has a superblock, a cylinder block, an array of inodes, and some data blocks.

The superblock contains static parameters of the file system. These include the total size of the file system, the block and fragment sizes of the data blocks, and assorted parameters that affect allocation policies. The superblock is identical in each cylinder group, so that it may be recovered from any one of them in the event of disk corruption.

The cylinder block contains dynamic parameters of the particular cylinder group. These include a bit map for free data blocks and fragments and a bit map for free inodes. Statistics on recent progress of the allocation strategies are also kept here.
There is an array of inodes in each cylinder group. The *inode* is the locus of most information about a specific file on the disk. It contains the user and group identifiers of the file, its times of last modification and access, a count of the number of hard links (directory entries) to the file, and the type of the file (plain file, directory, symbolic link, character or block device, or socket).

Finally, the inode contains pointers to the blocks containing the data in the file. The first dozen of these pointers point to *direct blocks*; that is, they directly contain addresses of blocks that contain data of the file. Thus, blocks of small files may be referenced with few disk accesses, since a copy of the inode is kept in main memory while a file is open. By supposing a major block size of 4096 bytes for the file system, then up to 48 kbytes of data may be accessed directly from the information in the inode.

The next three data block pointers in the inode point to *indirect blocks*. The first of these is a *single indirect block*; that is, the inode contains the address of a disk block that itself contains not data but rather the addresses of blocks that do contain data. Then there is a *double indirect block* pointer, containing the address of a block that contains the addresses of blocks that finally contain data. The last pointer would contain the address of a *triple indirect block*, but there is no need for it. The minimum block size for a file system is set to 4096 bytes so that files with as many as $2^{32}$ bytes will only use double, not triple, indirection. That is, since each block pointer takes four bytes, we have the number of bytes accessible from each type of pointer shown in Figure 8.

The number $2^{32}$ is significant because the file offset in the file structure in main memory is kept in a 32-bit word, preventing files from being larger.

Most of the cylinder group is taken up by *data blocks*, which contain whatever the users have put in their files. There is no distinction between ordinary files and directories at this level: Their contents are kept in data blocks in the same manner; only the type field in the inode distinguishes them.

A directory contains a sequence of (filename, inode number) pairs. In Version 7, filenames were limited to 14 characters, and so directories were arrays of 16-byte records, the last 14 bytes of each record containing the filename (null padded) and the first two bytes the inode number. Empty array elements were indicated by zero inode numbers. In 4.2BSD, filenames are limited to 255 characters, and the size of individual directory entries is variable up to this size. These long filenames require more overhead in the kernel, but make it easier to choose meaningful filenames without worrying about filename sizes.

A data block size larger than the usual hardware disk sector size of 512 bytes is desirable for speed. Since UNIX file systems usually contain a very large proportion of small files, much larger blocks cannot be used alone because they cause excessive internal fragmentation, that is, wasted space. This is why the earlier 4BSD file system was limited to a 1024-byte block.

The 4.2BSD solution is to use two sizes: All the blocks of a file are of a large block size, for example, 8192, except for the last, which may be an appropriate multiple of a *fragment size*, for example, 1024, to fill out the file. Thus, an 18,000-byte file would have two 8192-byte blocks and one 2048-byte partial block (which would not be completely filled). If the file were large enough to use indirect blocks, those would each be of the major block size; the fragment size applies only to data blocks.

The *block* and *fragment* sizes are set at file system creation according to the

---

*The name inode (pronounced eye node) is derived from "index node" and was originally spelled "i-node." The hyphen mostly fell out of use over the years, but both spellings may still be seen, along with the variant I node.*

---

**Figure 8.** Data block pointers and file size.
intended use of the file system: If many small files are expected, the fragment size should be small; if repeated transfers of large files are expected, the basic block size should be large. Implementation details force a maximum block/fragment ratio of 8/1 and a minimum block size of 4096, so typical choices are 4096/512 for the former case and 8192/1024 for the latter.

Suppose that data are written to a file in transfer sizes of 1024 bytes, and the block and fragment sizes of the file system are 4096 and 512 bytes. The file system will allocate a 1024-byte fragment to contain the data from the first transfer. The next transfer will cause a 2048-byte fragment to be allocated to hold the total data, and the data from the original fragment must be recopied. The local allocation routines do attempt to find space on the disk immediately following the existing fragment so that no copying will be necessary. Nevertheless, in the worst case, up to seven copies to new, larger, fragments may be required as a 1024-byte fragment grows into an 8192-byte block.

When extending a file that already ends in a fragment, 4.3BSD allocates a new fragment in a space large enough for a whole block, so that following writes will be unlikely to require recopying.

Provisions have been made for a user program to discover the block size of the file system for a file, so that the program may do data transfers of that size in order to avoid this fragment recopying. This is often done automatically for programs by the user-level standard I/O library (see Section 7.2).

4.4 Layout and Allocation Policies

The kernel uses a (device number, inode number) pair to identify a file. The device number resolves matters to the logical disk, and thus to the file system, since there may be only one file system per logical disk. The inodes in the file system are numbered in sequence. In the Version 7 file system, all inodes are in an array immediately following a single superblock at the beginning of the logical disk (there are no cylinder groups). The data blocks follow the inodes, and the inumber is effectively just an index into this array. Things are a bit more complicated in 4.2BSD, but the inumber is still just the sequence number of the inode in the file system.

In the Version 7 file system, any block of a file can be anywhere on the disk between the end of the inode array and the end of the file system, and free blocks are kept in a linked list in the superblock. The only constraint on the order of allocation of disk blocks is the order in which the free list happens to be when they are allocated. Thus the blocks may be arbitrarily far from both the inode and each other. Furthermore, the more a file system of this kind is used, the more disorganized the blocks in a file become. This entropic process can only be reversed by reinitializing and restoring the entire file system, which is not a convenient thing to do.

The cylinder group was introduced in 4.2BSD to allow localization of the blocks in a file. The header information in a cylinder group (the superblock, the cylinder block, and the inodes) is not always at the beginning of the cylinder group. If it were, the header information for every cylinder group would be on the same disk platter, and a single disk head crash could wipe them all out. Therefore each cylinder group has its header information at a different offset from the beginning of the group.

It is very common for the directory listing command ls to read all the inodes of every file in a directory, making it desirable for all such inodes to be close together. For this reason the inode for a file is usually allocated from the same cylinder group as that of its parent directory's inode. Not everything can be localized, however, and so an inode for a new directory is put in a different cylinder group from that of its parent directory. The cylinder group chosen for the new directory inode is the one with the greatest number of unused inodes.

To reduce the disk head seeks involved in accessing the data blocks of a file, data blocks are allocated from the same cylinder group as much as possible. Since a single file cannot be allowed to take up all the blocks in a cylinder group, a file exceeding a certain size (generally 48 or 96 kbytes)
has further block allocation redirected to a different cylinder group, the new group being chosen from among those having more than average free space. If the file continues to grow, allocation is again redirected at each megabyte to yet another cylinder group. Thus, all the blocks of a small file are likely to be in the same cylinder group, and the number of long seeks involved in accessing a large file is kept small.

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

The 4.2BSD file system is capable of using 30 percent or more of the bandwidth of a typical disk, in contrast to about 3 percent or less in Version 7.

4.5 Mapping a Pathname to an Inode

The user refers to a file by a pathname, while the file system uses the inode as its definition of a file. Thus the kernel has to map the user's pathname to an inode.

First a starting directory is determined. If the first character of the pathname is "/", this is an absolute pathname, and the starting directory is the root directory. If the pathname starts with anything other than a slash, this is a relative pathname, and the starting directory is the current directory of the current process.

The starting directory is checked for existence, proper file type, and access permissions, and an error is returned if necessary. The inode of the starting directory is always available and is used to search the directory, if required.

The next element of the pathname, up to the next "/", is looked for as a name in the starting directory, and an error returned if it is not found. If it is found, the number of the element as found in the directory is used to retrieve the inode.

If there is another element in the pathname, an error is returned if the current inode does not refer to a directory or if access is denied; else this directory is searched as was the previous. This process continues until the end of the pathname is reached and the desired inode is returned.

If at any point an inode has a file system mounted on it, this is indicated by a bit in the inode structure. The mount table is then searched to find the device number of the mounted device and the in-core copy of its superblock. The superblock is used to find the inode of the root directory of the mounted file system, and that inode is used. Conversely, if a pathname element is ".." and the directory being searched is the root directory of a file system that is mounted, the mount table must be searched to find the inode that it is mounted on, and that inode is used.

Hard links are simply directory entries like any other. Symbolic links are handled for the most part by starting the search over with the pathname taken from the contents of the symbolic link. Infinite loops are prevented by counting the number of symbolic links encountered during a pathname search and returning an error when a limit (eight) is exceeded.

The addition of symbolic links exacerbated the slowness of pathname translation, so that it became possibly the main efficiency problem in 4.2BSD. The problem has been avoided in 4.3BSD by having a cache of filename-to-inode references. Since it is very common for the ls directory listing program (and other programs) to scan all the files in a directory one by one, a directory offset cache was also added, so that when a process requests a filename in the same directory as its previous request,
the search through the directory is started where the previous name was found.

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

Once the inode is found by, for instance, the open system call, a file structure is allocated to point to the inode and to be referred to by a file descriptor by the user.

4.6 Mapping a File Descriptor to an Inode

System calls that refer to open files take a file descriptor as argument to indicate the file. (A file descriptor is a small nonnegative integer returned by the open or creat system calls or other system calls that open or create files; see Section 4.2.) The file descriptor is used by the kernel to index an array of pointers for the current process (kept in the process's user structure) to locate a file structure. This file structure in turn points to the inode. The relations of these data structures are shown in Figure 9.

The read and write system calls do not take a position in the file as argument. Rather the kernel keeps a file offset that is updated after each read or write according to the number of data actually transferred. The offset can be set directly by the lseek system call. If the file descriptor indexed an array of inode pointers instead of file pointers, this offset would have to be kept in the inode. Since more than one process may open the same file, and each such process needs its own offset for the file, keeping the offset in the inode is inappropriate. Thus the file structure is used to contain the offset.

File structures are inherited by the child process after a fork, so several processes may also have the same offset into a file.

The fcntl system call manipulates the file structure (it can be used to make several file descriptors point to the same file structure, for instance), whereas the ioctl system call manipulates the inode.

The inode structure pointed to by the file structure is an in-core copy of the inode on the disk and is allocated out of a fixed-length table. The in-core inode has a few extra fields, such as a reference count of how many file structures are pointing at it, and the file structure has a similar reference count for how many file descriptors refer to it.

The 4.2BSD file structure may point to a socket instead of to an inode.

5. I/O SYSTEM

Many hardware device peculiarities are hidden from the user by high-level kernel facilities, such as the file system and the socket interface. Other such peculiarities are hidden from the bulk of the kernel itself by the I/O system [Ritchie et al. 1979a; Thompson 1978]. This consists of buffer caching systems, general device driver code,
and drivers for specific hardware devices, which must finally address peculiarities of the specific devices. The various kernel I/O systems are diagramed in Figure 10.

There are three main kinds of I/O in 4.2BSD: the socket interface and its related protocol implementations, block devices, and character devices.

The socket interface, together with protocols and network interfaces, is treated in Section 6 on communications.

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

Character devices include terminals (e.g., `/dev/ttyOO) and line printers (/dev/lpO), but also almost everything else (except network interfaces) that does not use the block buffer cache. For instance, there is /dev/ mem, which is an interface to physical main memory, and /dev/null, which is a bottomless sink for data and an endless source of end of file markers. Devices such as high-speed graphics interfaces may have their own buffers or may always do I/O directly into the user’s data space; they are in any case classed as character devices.

Terminal-like devices use c-lists, which are buffers smaller than those of the block buffer cache.

The names block and character for the two main device classes are not quite appropriate; structured and unstructured would be better. For each of these classes there is an array of entry points for the various drivers. A device is distinguished by a class and a device number, both of which are recorded in the inode of special files in the file system. The device number is in two parts. The major device number is used to index the array appropriate to the class to find entries into the appropriate device driver. The minor device number is interpreted by the device driver as, for example, a logical disk partition or a terminal line.

A device driver is connected to the rest of the kernel only by the entry points recorded in the array for its class, by its use of common buffering systems, and by its use of common low-level hardware support routines and data structures. This segregation is important for portability, and also in configuring systems.

### 5.1 Block Buffer Cache

The block buffer cache serves primarily to reduce the number of disk I/O transfers required by file system accesses through the disk drivers.

Since it is common for system parameter files, commands, or directories to be read repeatedly, it is possible for their data blocks to be in the buffer cache when they are needed, so that it is not necessary to retrieve them from the disk.

Processes may write or read data in sizes smaller than a file system block or fragment. The first time a small read is required

```
<table>
<thead>
<tr>
<th>socket</th>
<th>network protocols</th>
<th>mbufs</th>
<th>network interface drivers</th>
</tr>
</thead>
<tbody>
<tr>
<td>plain file</td>
<td>file system</td>
<td>block buffer cache</td>
<td>block device drivers</td>
</tr>
<tr>
<td>cooked block interface</td>
<td>raw block interface</td>
<td>character device drivers</td>
<td></td>
</tr>
<tr>
<td>cooked tty interface</td>
<td>cooked tty line disciplines</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Figure 10. Kernel I/O structure.
from a particular file system block, the block will be transferred from the disk into a kernel buffer. Succeeding reads of parts of the same block then usually require only copying from the kernel buffer to the user process's memory. Writes are treated similarly, in that a cache buffer is allocated (and arrangements are made on the file system on the disk) when the first write to a file system block is made, and succeeding writes to the part of the same block are then likely to only require copying into the kernel buffer, and no disk I/O.

If the system crashes while data for a particular block are in the cache but have not yet been written to disk, the file system on the disk will be incorrect and those data will be lost. To alleviate this problem, writes are periodically forced for dirty buffer blocks. This is done (usually every 30 seconds) by a user process, sync, exercising a kernel primitive of the same name. There is also a system call, fsync, which may be used to force all blocks of a single file to be written to disk immediately: This is useful for database consistency.

Corruption of a directory can detach an entire subtree of the file system. Thus writes to disk are forced for directories on every write into the cache. The cache still improves efficiency for directories because reads of the same blocks will find copies of the data still in the cache.

The problem also exists for the in-core copies of inodes (which are kept in a kernel array dedicated to this purpose, and separate from the block buffer cache) and superblocks. Write-throughs to the disk on writes into the kernel copies are also forced for these, although the write-through is somewhat selective for inodes.

Most magnetic tape accesses are, in practice, done through the appropriate raw tape device (see Section 5.2), bypassing the block buffer cache. When the cache is used, tape blocks must still be written in order; so the tape driver forces synchronous writes for them.

The block buffer cache consists of a number of buffer headers. Each buffer header contains a device number, the number of a block on the device, a pointer to a piece of physical memory, the size of the piece of memory, and the amount of that memory that contains useful data. The term buffer is usually used to refer to a buffer header and the associated useful data. The term empty buffer, however, refers to a buffer header with no associated memory or data.

The buffer headers are kept in several linked lists:

- **Reserved buffers**: blocks containing information that stays in main memory, such as superblocks of file systems.
- **The cache**: blocks not currently in use, but likely to be reused, in least recently used order.
- **The aged queue**: blocks unlikely to be reused, also in least recently used order.
- **The empty queue**: buffer headers with no memory or disk blocks associated with them.
- **Device driver active queues**: Each block device has a queue of buffers on which I/O is active or pending. A buffer may be on one of these active queues at a time and will not appear at the same time in the cache, the aged queue, or the empty queue.

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

On system start-up, a pool of memory pages is reserved for use with the block buffer system. The size of this pool, and the number of buffer headers, is determined according to the size of available main memory. All the pages are initially linked into buffer headers, which are all linked into the aged queue.

When a read is wanted from a device, the appropriate block is first searched for in the cache. If it is found, it is used as is, and no I/O transfer is necessary. If it is not found, a buffer is chosen in a similar manner as for a read. If the block is already in the buffer cache, the new data
are put in the buffer, and the buffer header is marked to indicate that the buffer has been modified. If the whole buffer is written, it is queued for I/O and put on the aged queue. Otherwise, no I/O transfer is initiated, and the buffer is left in the main cache queue.

Similarly, when a read of a buffer reaches the end of the buffer, it is put on the aged queue; otherwise, it is left in the cache. Thus buffers that have been involved in partial transfers are unlikely to be reused for other purposes before succeeding partial transfers happen, whereas blocks that have been completely used are made more readily available. Transfers of large files, done in transfers of the file system block size or larger, will usually use buffers out of the aged queue without disturbing buffers in the main cache.

The number of data in a buffer in 4.2BSD is variable, up to a maximum (usually 8 kbytes) over all file systems. The minimum size of a buffer is the fragment size (see Section 4.3) of the file system, usually 512 or 1024 bytes. The buffer header specifies the number of data in the buffer, which is either a full-sized block or one to seven contiguous fragments. Each buffer header also specifies the amount of memory allocated to the buffer, which is an integral number of memory clusters. Each memory cluster in the buffer pool is mapped to exactly one buffer at a time. When a buffer is allocated, if it has insufficient memory for the block to be stored, pages are taken from free buffers to fill out the buffer, and the now-empty free buffers are put on the empty queue. Many buffers are of the sizes of the major blocks of the file systems in use. UNIX file systems, however, frequently contain many small files as well as larger ones, and there are consequently many buffers of one, two, or three times the common fragment sizes.

The truncate system call may be used to reduce the size of a file without decreasing it to zero. If a file is truncated to a size that is not a multiple of the major block size of its file system, the last block in the file may be a fragment. If there is a block buffer corresponding to the new end of the file at that time, the buffer must shrink to match the now-smaller file block. In this case, a buffer is taken off the empty queue, and excess pages from the shrinking buffer are put into it.

The size of the buffer cache can have a profound effect on the performance of a system. If it is large enough, the percentage of cache hits can be quite high and the number of actual I/O transfers low.

There are some interesting interactions between this buffer cache, the file system, and the disk drivers. When data are written to a disk file, the buffers are put on a device driver's I/O queue. The disk device driver keeps this active queue sorted by disk address to minimize disk head seeks and to write data at times optimized for disk rotation. When data are read from a disk file, the block I/O system does some read-ahead; reads, however, are much more nearly asynchronous than writes, making long disk head seeks and missed disk revolutions more likely. Thus output to the disk through the file system is often faster than input for large transfers, counter to intuition.

5.2 Raw Device Interfaces

Almost every block device also has a character interface. These are called raw device interfaces and are accessed through separate special files; for example, /dev/rhpOh might be the raw interface corresponding to /dev/hpOh. Such an interface differs from the block interface in that the block buffer cache is bypassed.

Each disk driver maintains an active queue of pending transfers. Each record in the queue specifies whether it is for reading or writing, a main memory address for the transfer, a device address for the transfer (usually the 512-byte block number), and a transfer size (in bytes). It is simple to map the information from a block buffer to what is required for this queue.

It is almost as simple to map a piece of main memory corresponding to part of a user process's virtual address space into a buffer. For instance, this is what a raw disk interface does. Transfers directly between a user's virtual address space and a device are allowed by this mapping. The size of the transfer is limited by the physical device, some of which require an even number
of bytes. However, it may be larger than the largest block that the block buffer system can handle, and it may be a size that is not a multiple of the page cluster size (nor a multiple of 512 bytes). The software restricts the size of a single transfer to what will fit in a 16-bit word; this is an artifact of the system's PDP-11 history and of the PDP-11 derivation of many of the devices themselves.

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

The 4.2BSD file system implementation was actually written and largely tested as a user process that used a raw disk interface before the code was moved into the kernel.

5.3 C-Lists

Terminal drivers and the like use the character buffering system. This involves small (usually 256-byte) blocks of characters kept in linked lists. There are routines to enqueue and dequeue characters for such lists. Although all free character buffers are kept in a single free list, most device drivers that use them limit the number of characters that may be queued at one time for a single terminal port.

A write system call to such a device puts characters on an output queue for the device. An initial transfer is started, and hardware interrupts cause dequeueing of characters and further transfers. These interrupts are handled asynchronously by the kernel and independently of user processes. The stack used is a special interrupt stack, not the kernel stack of the system half of a user process. If there is not enough space on the output queue for all the data supplied by a write system call, the system call may block until space becomes available when interrupts transfer characters from the output queue to the device.

Input is similarly interrupt driven. Terminal drivers, however, typically support two input queues: the raw queue and the canonical queue. The raw queue collects characters exactly as they come from the terminal port. This is done asynchronously: the user process does not have to do a read system call first. If there is not enough space in the raw queue when an input interrupt occurs, the data are thrown away.

When a user process uses a read system call to attempt to get data from the device, any data already in the canonical queue may be returned immediately. If there are not enough data in the canonical queue, the read system call will block until interrupts cause enough data to be put on the canonical queue. Transfer of characters to the canonical queue is triggered when the interrupt routine puts an end of line character on the raw queue. If there is a process blocked on a read from the device, it is awakened and its system half does the transfer. Some conversion is done during the transfer: Carriage returns may be translated to line feeds and special handling of other characters may be done.

It is also possible to have the device driver bypass the canonical queue so that the read system call returns characters directly from the raw queue. This is known as raw mode.

6. COMMUNICATIONS

Many tasks can be accomplished in isolated processes, but many others require interprocess communication (IPC). Isolated computing systems have long served for many applications, but networking is increasingly important, especially with the increasing use of personal workstations. Resource sharing ranging up to true distributed systems is becoming more common.

6.1 Signals

Signals are a facility for handling exceptional conditions. They are not interprocess communication in any usual sense, but they are an important facility of UNIX and need to be discussed.

There are a few dozen signals, each corresponding to a certain condition. A signal may be generated by a keyboard interrupt, by an error in a process, such as a bad memory reference, or by a number of asynchronous events, such as timers or job control signals from the C shell. Almost any signal may also be generated by the kill
system call, which may, in turn, be exercised by the kill command from the shell.

The interrupt signal, SIGINT, is usually produced by typing the \"C (control C)\" character on a terminal keyboard. This signal is used to stop a command before it would otherwise complete. The quit signal, SIGQUIT, is usually produced by typing the \\ character (ASCII FS) and has a similar effect, except that it also causes the process involved to write its current memory image to a file called core in the current directory for use by debuggers. (Such characters that send signals or cause other special effects when typed at the terminal may all be set by the user. ASCII DEL and \U are two other common choices for the interrupt and quit characters.) The signals SIGSTOP and SIGCONT are used by the C shell in job control to stop and restart a process. SIGILL is produced by an illegal instruction and SIGSEGV by an attempt to address memory outside of a process's legal virtual memory space.

When a signal is generated, it is queued until the system half of the affected process next runs. This usually happens soon, since the signal causes the process to be awakened if it has been waiting for some other condition. The default action of a signal is for the process to kill itself and many signals have a side effect of producing a core file.

Arrangements can be made for most signals to be ignored (to have no effect) or for a routine in the user process (a signal handler) to be called. There is one signal (the kill signal, number nine, SIGKILL) that cannot be ignored or caught by a signal handler. SIGKILL is used, for instance, to kill a runaway process that is ignoring other likely signals, such as SIGTERM, which is the soft kill signal, which many processes expect to catch and then die gracefully.

Signals can be lost: If another of the same kind is sent before a previous one has been processed by the process it is directed to, the first one will be overwritten and only one will be seen by the process.

Signals have been reimplemented in 4.2BSD according to a new model that avoids many of the race conditions of the older model. In particular, while one signal is being handled by the user process, other signals for the same process are automatically blocked by the kernel. Still, signals are mostly appropriate for handling exceptional conditions, not for large-scale IPC.

### 6.2 Interprocess Communication

Interprocess communication (IPC) traditionally has not been one of UNIX’s strong points.

Most UNIX distributions have not permitted shared memory because the PDP-11 hardware did not encourage it. System V does support a shared memory facility, and one was planned for 4.2BSD but not implemented owing to time constraints. Shared memory presents a problem in a networked environment, since network accesses can never be as fast as memory accesses on the local machine. Although one could pretend that memory was shared between two separate machines by copying data across a network transparently, still the major benefit of shared memory, speed, would be lost.

The pipe is the IPC mechanism most characteristic of UNIX and is basic to the pipeline facility described in Section 7.3. A pipe permits a reliable unidirectional byte stream between two processes. It is traditionally implemented as an ordinary file, with a few exceptions. It has no name in the file system, being created instead by the pipe system call (the resulting file descriptor may then be manipulated by the read, write, close system calls, and by others, which may be used on ordinary files). A pipe’s size is fixed, and once all data previously written into one have been read out, writing recommences at the beginning of the file (pipes are not true circular buffers). One benefit of the small size (usually 4096 bytes) of pipes is that pipe data are seldom actually written to disk, being usually kept in memory by the block buffer cache system.

In 4.2BSD, pipes are implemented as a special case of the socket mechanism [Leflfer et al. 1983a], which provides a general interface not only to facilities such as pipes, which are local to one machine, but also to networking facilities.
A socket is an end point of communication. A socket in use usually has an address bound to it. The nature of the address depends on the communication domain of the socket. A characteristic property of a domain is that processes communicating in the same domain use the same address format. A single socket ordinarily communicates in only one domain.

The two domains currently implemented in 4.2BSD are the UNIX domain (AF_UNIX) and the Internet domain (AF_INET). The address format of the UNIX domain is that of ordinary file system pathnames, such as /alpha/beta/gamma. Processes communicating in the Internet domain use DARPA Internet communications protocols such as TCP/IP and Internet addresses, which consist of a 32-bit host number and a 16-bit port number.

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

- **SOCK_STREAM.** Stream sockets provide reliable duplex sequenced data streams. No data are lost or duplicated in delivery, and there are no record boundaries. This type is supported in the Internet domain by the TCP protocol. In the UNIX domain, pipes are implemented as a pair of communicating stream sockets.
- **SOCK_SEQPACKET.** Sequenced packet sockets would provide data streams like those of stream sockets, except that record boundaries would be preserved. This type is unsupported in 4.2BSD. The Xerox Network Services sequenced packet protocol is supported in the NS domain in 4.3BSD.
- **SOCK_DGRAM.** Datagram sockets transfer messages of variable size in either direction. There is no guarantee that such messages will arrive in the same order in which they were sent, or that they will be unduplicated, or that they will arrive at all, but the original message (record) size is preserved in any datagram that does arrive. This type is supported in the Internet domain by the UDP protocol.
- **SOCK_RDM.** Reliably delivered message sockets would transfer messages that would be guaranteed to arrive, and would otherwise be like the messages transferred using datagram sockets. This type is currently unsupported.
- **SOCK_RAW.** Raw sockets allow direct access by processes to the protocols that support the other socket types. For example, in the Internet domain it is possible to reach IP with SOCK_RAW, below TCP or UDP, which are reached by SOCK_STREAM and SOCK_DGRAM, respectively. This capability is useful for developing new protocols.

The socket facility has a set of system calls specific to it. The socket system call creates a socket. It takes as arguments specifications of the communication domain, the socket type, and the protocol to be used to support that type. The value returned is a small integer called a socket descriptor, which is in the same name space as file descriptors, indexes the same descriptor array in the user structure in the kernel, and has a file structure allocated for it. The file structure points not to an inode structure but rather to a socket structure, which keeps track of the socket's type, state, and the data in its input and output queues.

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

Many processes that communicate by using the socket IPC follow the client/server model. In this model, the server process provides a service to the client process. When the service is available, the server
process listens on a well-known address, and the client process uses connect, as above, to reach it.

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

There are also system calls for setting parameters of a connection and for returning the address of the foreign socket after an accept.

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

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

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

The select system call may be used to multiplex data transfers on several file descriptors and/or socket descriptors. It may even be used to allow one server process to listen for client connections for many services and fork a process for each connection as it is made. This is done by doing socket, bind, and listen for each service, and then doing select on all the socket descriptors. When select indicates activity on a descriptor, the server does accept on it and forks a process on the new descriptor returned by accept, leaving the parent process to do select again.

6.3 Networking

This section assumes some basic knowledge of the concepts of networking separate computer systems, or hosts, by means of network protocols over communication media to form networks [Tanenbaum 1981].

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

Many installations that have 4.2BSD systems have several VAXs or workstations such as Suns connected by networks. Although the 4.2BSD distribution does not support a true distributed operating system, still remote login, file copying across networks, remote process execution, etc., are trivial from the user's viewpoint.

4.2BSD supports the DARPA Internet protocols [RFCs n.d.; MIL-STD n.d.] UDP, TCP, IP, and ICMP on a wide range of Ethernet, token ring, and IMP (ARPA-NET) interfaces. The standard Internet application protocols (and their corresponding user interface and server programs) Telnet (remote login), FTP (file transfer), and SMTP (mail) are supported. 4.2BSD also provides the 4.2BSD-specific application programs (and underlying network protocols), rlogin (remote login), rcp
The framework in the kernel to support networking [Leffler et al. 1983] is accessible via the socket interface and is intended to facilitate the implementation of further protocols (4.3BSD includes the XEROX Network Services protocol suite). The first version of the code involved was written by Rob Gurwitz of BBN as an add-on package for 4.1BSD.

Several models of network layers are relevant to the 4.2BSD implementations. These models are diagramed in Figure 11. The Open System Interconnection (OSI) Reference Model for networking [ISO 1981] of the International Organization for Standardization (ISO) prescribes seven layers of network protocols and strict methods of communication between them. An implementation of a protocol may only communicate with a peer entity speaking the same protocol at the same layer, or with the protocol-protocol interface of a protocol in the layer immediately above or below in the same system.

The 4.2BSD networking implementation, and to a certain extent the socket facility, is more oriented toward the ARPA Reference Model (ARM) [Padlipsky 1983]. The ARPANET in its original form served as proof of concept for many networking concepts such as packet switching and protocol layering. It serves today as a communications utility for researchers. The ARM predates the ISO model and the latter was in large part inspired by the ARPANET research.

The ARPANET and its sibling networks that run IP and are connected together by gateways form the ARPA Internet. This is a large, functioning internetwork that appears to the naive user to be one large network, owing to the design of the protocols involved [Cerf and Cain 1983; Padlipsky 1985]. It is also a test bed for ongoing internet gateway research. The ISO protocols currently being designed and implemented take many features from this already functional DoD internetwork.

Whereas the ISO model is often interpreted as requiring a limit of one protocol per layer, the ARM allows several protocols in the same layer. There are only three protocol layers in the ARM:

- **Process/Applications** subsumes the Application, Presentation, and Session layers of the ISO model. Such user-level programs as the File Transfer Protocol (FTP) and Telnet (remote login) exist at this level.
- **Host–Host** corresponds to ISO’s Transport and the top part of its Network layers. Both the Transmission Control Protocol (TCP) and the Internet Protocol (IP) are in this layer, with TCP on top of IP. TCP corresponds to an ISO Transport protocol and IP performs the addressing functions of the ISO Network layer.
- **Network Interface** spans the lower part of the ISO Network layer and all of the Data Link layer. The protocols involved here depend on the physical network type. The ARPANET uses the...
IMP-Host protocols, while an Ethernet uses Ethernet protocols.

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

The networking framework in 4.2BSD is more generalized than either the ISO model or the ARM, although it is most closely related to the latter.

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

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

A protocol may communicate with another protocol or with the network interface that is appropriate for the network hardware. The general framework places little restriction on what protocols may communicate with what other protocols, or on how many protocols may be layered on top of one another. The user process may, by means of the SOCK_RAW socket type, access protocols underlying the transport protocols. This capability is used by routing processes and also for new protocol development.

There tends to be one network interface driver for each hardware model of network controller. The network interface is responsible for handling characteristics specific to the local network being addressed in such a way that the protocols using the interface need not be concerned with them.

The functions of the network interface depend largely on the network hardware, which is whatever is necessary for the network it is connected to. Some networks may support reliable transmission at this level, but most do not. Some provide broadcast addressing, but many do not.

There are projects in progress at various organizations to implement protocols other than the DARPA Internet ones, including the protocols ISO has thus far adopted to fit the OSI model.

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

Data are ordinarily passed between layers (socket/protocol, protocol/protocol, or protocol/network interface) in mbufs. This ability to pass the buffers containing the data eliminates some data copying, but there is still frequent need to remove or add protocol headers. It is also convenient and efficient for many purposes to be able to hold data of the size of the memory management page. Thus it is possible for its data to reside not in the mbuf itself, but elsewhere in memory. There is an mbuf page table for this purpose, as well as a pool of pages dedicated for this use.

6.4 Distributed Systems

4.2BSD is not a distributed operating system, so this section concentrates less on 4.2BSD and more on UNIX in general than most sections.

Several attempts have been made to provide some distributed facilities, ranging from the ability to mount a remote file system as if it were attached to the local machine, to LOCUS, which is a true distributed operating system.

There is at least one 4.2BSD implementation of an extended file system (not distributed with 4.2BSD) that allows mounting a file system on a disk connected to a remote machine as if it were directly connected to the local machine. It suffers from performance problems due to excessive generality: There is no record locking, and thus there can be no local caching of data; every byte read or written has to be transferred over the network.
Version 8 has a similar capability of mounting a file system from a remote system over a network as if it were local [Weinberger 1984]. The networking implementation that supports it is unrelated to the 4.2BSD implementation, as are the protocols used. A user process needs only the ordinary file access system calls to use either of these remote mount capabilities once the file system is mounted.

The IBIS distributed file system from Purdue [Tichy and Ruan 1984] also has this file access transparency and goes further. IBIS claims file location transparency; that is, the user does not need to know which host actually contains a given file. Efficiency and robustness are promoted by replication of data on several hosts. Files may also be migrated to the host from which they are most commonly accessed. IBIS is implemented on top of 4.2BSD and thus can use a variety of underlying network hardware. However, it evidently currently runs only on VAX CPUs.

The LOCUS Distributed UNIX System from Locus Computing Corporation [Butterfield and Popek 1984; Popek et al. 1981] is a true distributed operating system. It provides not only a distributed file system, but also transparent process execution on any of the participating CPUs that can support the process, even on a different CPU type than that from which the process is invoked. The system runs on a variety of CPU types. It uses network protocols designed specifically for its purposes, and which are not interoperable with other protocol suites above the network layer. Lower layers that may be used include Ethernet. The implementors claim object code compatibility with both System V and 4BSD.

LOCUS is an example of a closed system, in which every participant must be running the same software. Such systems have the advantage of being able to efface most evidence of multiple CPUs and file systems so that they appear to the user as a single computer system. They can also be very efficient.

Most research organizations and many businesses have a variety of hardware, from several vendors, supporting different operating systems and with different purposes. There may be personal workstations with bit-map screens, real-time systems for acquiring data, mainframes for bulk data storage and manipulation, and array processors or supercomputers for large-scale number crunching. It is unlikely that a closed system would be supported on all such equipment and operating systems. A single operating system or a single version of one may not be suited for all of the above tasks. Many vendors support their own operating systems, which are adapted to a particular kind of task, and would not be willing to adopt another vendor's operating system.

Nonetheless, it is necessary to share information in such a heterogeneous computing environment. Information sharing in the DARPA Internet, among a very diverse range of machines, has long been supported by the client/server model, which requires the individual machines to support only certain well-defined protocols, not entirely identical operating systems. This is the open system approach. 4.2BSD uses this model in providing the basic Internet services of remote login, file transfer, mail exchange, and remote job execution.

A recent useful application of the open system approach is Sun Microsystems' Network File System (NFS) [Walsh et al. 1985]. This allows file access transparency and a large degree of file location transparency. A foreign file system (or a part of one) may be mounted on the local system, just as a file system on a local disk may be mounted. A file on the foreign system then appears to the user on the local system as just another file in the file system. Although this has only been implemented thus far on 4.2BSD (on a variety of vendor hardware), there is no reason why it could not be implemented on a different operating system. Then one could mount a foreign VMS, TOPS-20, or MSDOS file system on a local UNIX file system, and access the VMS (or other) files as if they were UNIX files, using an apparent UNIX directory structure to find them. Similarly, a UNIX file system mounted on a VMS machine would appear as part of the VMS file system and directory structure. (This flexibility has led to problems of acceptance of
NFS by the UNIX community because some UNIX features, such as setuid capability, file locking, and device access, have been sacrificed for interoperability with other operating systems.)

NFS is not a distributed operating system; rather, it is a network (not distributed) file system. Thus it cannot provide the transparent process execution of a closed system like LOCUS. However, the basic services of remote login, file transfer, and remote process execution are already supported by 4.2BSD, and NFS does provide transparent file access and transparent file location on heterogeneous systems, owing to its open system design [Joy 1984; Morin 1985].

7. USER INTERFACE

Although most aspects of UNIX appropriate for discussion in this paper are implemented in the kernel, the nature of the user interface is sufficiently distinctive and different from those of most previous systems to deserve being discussed.

7.1 Shells and Commands

The command language interpreter in UNIX is a user process like any other. It is called a shell, as it surrounds the kernel of the operating system. It may be substituted for, and there are in fact several shells in general use [Korn 1983; Tuthill 1985]. The Bourne shell [Bourne 1978], written by Steve Bourne, is probably the most widely used, or at least the most widely available. The C shell [Joy 80], mostly the work of Bill Joy, is the most popular on BSD systems.

There are also a number of screen- or menu-oriented shells, but we describe here only the more traditional line-oriented interfaces.

The various common shells share much of their command language syntax. The shell indicates its readiness to accept another command by typing a prompt, and the user types a command on a single line, for example,

$ ls

The dollar sign is the usual Bourne shell prompt and the ls typed by the user is the list directory command. Most commands may also take arguments, which the user types after the command name on the same line and separated from it and each other by white space (spaces or tabs).

Although there are a very few commands built into the shells, a typical command is represented by an executable binary object file, which the shell finds and executes. The object file may be in one of several directories, a list of which is kept by the shell. This list is known as the search path and is settable by the user. The directories /bin and /usr/bin are almost always in the search path, and a typical search path on a BSD system might look like this:

(./usr/local /usr/ucb /bin /usr/bin)

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

Execution of a command is done by a fork (or ufork) system call followed by an exec of the object file (see Figure 3 in Section 2.1). The shell usually then does a wait to suspend its own execution until the command completes. There is a simple syntax (an ampersand at the end of the command line) to indicate that the shell should not wait. A command left running in this manner while the shell continues to interpret further commands is said to be a background command, or to be running in the background. Processes on which the shell does wait are said to run in the foreground.

The C shell in 4BSD systems provides a facility called job control (implemented partially in the kernel) that allows moving processes between foreground and background, and stopping and restarting them on various conditions. This allows most of the control of processes provided by windowing or layering interfaces, and requires no special hardware.

7.2 Standard I/O

Processes may open files as they like, but most processes expect three file descriptors (see Section 4.6 and Figure 9) to already be open, having been inherited across the exec
These file descriptors are numbers 0, 1, and 2, more commonly known as **standard input**, **standard output**, and **standard error**. Frequently, all three are open to the user's terminal. Thus the program can read what the user types by reading standard input, and the program can send output to the user's screen by writing to standard output. Most programs also accept a nonterminal file as standard input or standard output.

The standard error file descriptor is also open for writing and is used for error output, whereas standard output is used for ordinary output.

There is a user-level system library that many programs include because it buffers I/O for efficiency. This is the **standard I/O library**. It has routines called **fread**, **fwrite**, and **fseek**, which are analogous to the lower level **read**, **write**, and **lseek** system calls. Whereas the system calls are applied to a file descriptor, the standard I/O routines are applied to a stream, which is declared in C as a pointer to a structure that contains the file descriptor and the buffer. Writes by the program to a stream by **fwrite** do not cause a **write** system call until the whole buffer is filled. Similarly, if the buffer is empty when an **fread** is done, a **read** system call will be done to fill it, but succeeding **fread** calls will fetch data out of the buffer until the end of the buffer is reached. Thus the library minimizes the number of system calls and does the actual I/O transfers in efficient sizes, whereas the program retains the flexibility to read or write to the standard I/O system in transfers of any size appropriate to the program's algorithms. For maximum efficiency, the library normally sets the size of the buffer to the block size of the file system corresponding to the stream.

Use of the standard I/O library is indicated by the inclusion of the parameter file **stdio.h** in its source. Such a program will find streams already open for standard input, output, and error under the names **stdin**, **stdout**, and **stderr**, respectively. Other streams may be opened by **fopen**, which takes a filename and a mode argument and returns a stream open appropriately for reading or writing. A stream may be closed with **fclose**.

The shells have a simple syntax for changing what files are open for a process’s standard I/O streams, that is, for standard I/O reduction:

```
$ ls >filea  # either shell
$ pr <filea >fileb  # direct output of ls to file filea
$ lpr < fileb  # input from filea and output to fileb
$ lpr < fileb >&2  # direct both standard output and error to errs
```

The **ls** command produces a listing of the names of files in the current directory, the **pr** command formats the list into pages suitable for outputting on a printer, and the **lpr** command sends the formatted output to a printer.

### 7.3 Pipelines, Filters, and Shell Scripts

The above example of I/O redirection could have been done all in one command, as

```
% ls | pr | lpr
```

Each vertical bar tells the shell to arrange for the output of the preceding command to be passed as input to the following command. The mechanism that is used to carry the data is called a **pipe** (see Section 6.2) and the whole construction is called a **pipeline**. A pipe may be conveniently thought of as a simplex, reliable, byte stream, and is accessed by a file descriptor, like an ordinary file. In the example, the write end of one pipe would be set up (see Section 2.1) by the shell to be the standard output of **ls** and the standard input of **pr**; there would be another pipe between the **pr** and **lpr** commands.

A command like **pr** that passes its standard input to its standard output, performing some sort of processing on it, is called...
a filter. (Filters may also take names of input files as arguments, but never names of output files.) Very many UNIX commands (probably most) may be used as filters. Thus complicated functions may be pieced together as pipelines of common commands. Also, common functions, such as output formatting, need not be built into numerous commands, since the output of almost any program may be piped through `pr` (or some other appropriate filter).

All the common UNIX shells are also programming languages, with the usual high-level programming language control constructs, as well as variables internal to the shell. The execution of a command by the shell is analogous to a subroutine call.

A file of shell commands, a shell script, may be executed like any other command, with the appropriate shell being invoked automatically to read it. Shell programming may thus be used to combine ordinary programs conveniently for quite sophisticated applications without the necessity of programming in conventional languages.

The isolation of the command interpreter in a user process, the shell, both allowed the kernel to stay a reasonable size and permitted the shell to become rather sophisticated, as well as substitutable. The instantiation of most commands invoked from the shell as subprocesses of the shell facilitated the implementation of I/O redirection and pipelines, as well as making background processes (and later Berkeley's job control) easy to implement.

### 7.4 The UNIX Philosophy

There is something sometimes referred to as “the UNIX philosophy.” Part of it has been elaborated or at least alluded to above (see Section 1.2). Here is a statement of it that is both more explicit and also more oriented toward the levels of the operating system that the ordinary user sees:

1. Make each program do one thing well. To do a new job, build afresh rather than complicate old programs by adding new “features.”

2. Expect the output of every program to become the input of another, as yet unknown, program. Do not clutter output with extraneous information. Avoid stringently columnar or binary input formats. Do not insist on interactive input.

3. Design and build software, even operating systems, to be tried early, ideally within weeks. Do not hesitate to throw away the clumsy parts and rebuild them.

4. Use tools in preference to unskilled help to lighten a programming task, even if you have to detour to build the tools and expect to throw some of them out after you have finished with them [McIlroy et al. 1978].

These principles have led to the development of not only byte-oriented, typeless files, but also of pipes and pipelines, and the ability to combine existing programs to build new ones. Much of the power and popularity of UNIX is based on the facilities provided by the shells and other programs such as `make`, `awk`, `sed`, `lex`, `yacc`, `find`, `SCCS`, etc. The principles also lead indirectly to the use of programming languages such as C, which are not machine dependent, particularly not assembly language. That, in turn, leads to portability, which may well be the single greatest reason for the popularity of the system. Such matters are beyond the scope of this paper, but references are provided in the next section.

One may consider these ideas to be mere elaborations of structured programming principles, or ad hoc practical techniques, or “creeping elegance,” and there is some of all of that here. It is true that many users of UNIX, including many applications developers, do not seem to be aware of or at least do not use these principles any more, but their worth is still evidenced by the system itself, and there has been some effort of late to reacquaint people with them [Pike and Kernighan 1984].

This is a programmer’s philosophy, and the result is a programmer’s system. It does not limit the areas of application of the system, however, because a good programming environment makes it easy for the programmer to build user interfaces to fit applications to the needs of the end user.
8. BIBLIOGRAPHIC NOTES

The set of documentation that comes with UNIX systems is called the UNIX PROGRAMMER'S MANUAL (UPM) and is traditionally organized in two volumes [UPMV7 1983]. Volume 1 contains short entries for every command, system call, and subroutine package in the system and is also available on line via the man command. Volume 2—Supplementary Documents (usually bound as Volumes 2A and 2B) contains assorted papers relevant to the system and manuals for those commands or packages too complex to describe in a page or two. Berkeley systems add Volume 2C [UPMBC 1983] to contain documents concerning Berkeley-specific features. The whole 4.2BSD UPM has been published by the USENIX Association, reorganized in three logical volumes and five physical volumes.

There are many useful papers in the two special issues of The Bell System Technical Journal, BSTJ [1978] and BLTJ [1984]. The first of these contains the paper with the most pervasive influence [Ritchie and Thompson 1978], which also appears in UPMV7 [1983]. There are also many useful papers in various USENIX Conference Proceedings, especially USENIX [1983], USENIX [1984], and USENIX [1985].

Possibly the best book on general programming under UNIX, especially on the use of the shell and facilities such as yacc and sed, is Kernighan and Pike [1984]. Two others of interest are Bourne [1983] and McGilton and Morgan [1983].

ACKNOWLEDGMENTS

Many useful comments on drafts of this paper were received from John R. Chambers, William N. Joy, Mike Karels, Samuel J. Leffler, Bill Shannon, and Anthony I. Wasserman. Much of the accuracy of the paper is due to them, and to the others who reviewed it. All of the inaccuracies and other faults are due to the present authors.

An early draft of this paper was used by Peterson and Silberschatz in writing the chapter on 4.2BSD in Peterson and Silberschatz [1985]. Some text in the current paper is adapted from that chapter. Thanks to Bill Tuthill for the DITROFF macros for the Contents.

Particular thanks from J. S. Quarterman to Sam Leffler for technical assistance and forbearance over the years, to Jim Peterson for perfectionism, to Avi Silberschatz for patience, to John B. Chambers for persistent participation, and to Carol Kroll, who mediated.

REFERENCES


BLTJ 1984. The UNIX system. AT&T Bell Lab. Tech. J. 63, 8 (Oct.).


BSTJ 1978. UNIX time-sharing systems. Bell Syst. Tech. J. 57, 6, Pt. 2 (July–Aug.).


Peirce, N. 1985. Putting UNIX in perspective: An interview with Victor Yysotsky. The manager of AT&T's Multics project remembers the way it was. UNIX Rev. 3, 1, 58.


Received March 1985; revised November 1985; final revision accepted February 1986.

Computing Surveys, Vol. 17, No. 4, December 1985