The Design and Implementation of the FreeBSD Operating System

FreeBSD version 5.2

Marshall Kirk McKusick
George V. Neville-Neil

UNIX/Operating Systems

As in earlier Addison-Wesley books on the UNIX-based usick and George Neville-Neil deliver here the most comprehensive, up-to-date, and authoritative technical information on the internal structure of open source FreeBSD. Readers involved in technical and sales support can learn the capabilities and limitations of the system; applications developers can learn effectively and efficiently how to interface to the system; system administrators can learn how to maintain, tune, and configure the system; and systems programmers can learn how to extend, enhance, and interface to the system.

The authors provide a concise overview of FreeBSD's design and implementation. Then, while explaining key design decisions, they detail the concepts, data structures, and algorithms used in implementing the systems facilities. As a result, readers can use this book as both a practical reference and an in-depth study of a contemporary, portable, open source operating system.

This book:

- Details the many performance improvements in the virtual memory system
- Describes the new symmetric multiprocessor support
- Includes new sections on threads and their scheduling
- Introduces the new jail facility to ease the hosting of multiple domains
- Updates information on networking and interprocess communication

Already widely used for Internet services and firewalls, high-availability servers, and general timesharing systems, the lean quality of FreeBSD also suits the growing area of embedded systems. Unlike Linux, FreeBSD does not require users to publicize any changes they make to the source code.

Marshall Kirk McKusick writes books and articles, consults, and teaches classes on UNIX- and BSD-related subjects. While at the University of California at Berkeley, he implemented the 4.2BSD fast file system, and was the research computer scientist at the Berkeley Computer Systems Research Group (CSRG) overseeing the development and release of 4.3BSD and 4.4BSD. He has twice served as the president of the board of the USENIX Association.

George V. Neville-Neil works on network and operating system code for fun and profit and teaches programming. He also serves on the editorial board of *Queue* magazine and is a member of the USENIX Association, ACM, and IEEE.

www.awprofessional.com/
www.FreeBSD.org

Cover design by Chuti Prasertsith
Cover art by John Lasseter

♻ Text printed on recycled paper

Addison-Wesley
Pearson Education

55999
9 780201 702453
ISBN 0-201-70245-2

$59.99 US
$86.99 CANADA

Page 2 of 43

The Design and Implementation
of the FreeBSD Operating System
Mc
Nevil
FT MEADE
GenColl
QA 76
.76
.063
M398745
2005
Copy 2

TX 6-034-316

TX0006034316

The Design and Implementation of the FreeBSD Operating System

Marshall Kirk McKusick

George V. Neville-Neil

Addison-Wesley

Boston • San Francisco • New York • Toronto • Montreal
London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

The publisher offers discounts on this book when ordered in quantity for bulk purchases and special sales. For more information, please contact:

U.S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside of the U.S., please contact:

International Sales
(317) 581-3793
international@pearsontechgroup.com

Visit Addison-Wesley on the Web: www.awprofessional.com

Library of Congress Cataloging-in-Publication Data

McKusick, Marshall Kirk.
The design and implementation of the FreeBSD operating system / Marshall Kirk McKusick, George V. Neville-Neil.
p. cm.
Includes bibliographical references and index.
ISBN 0-201-70245-2 (hc : alk. paper)
1. FreeBSD. 2. Free computer software. 3. Operating systems (Computers) I. Neville-Neil, George V. II. Title.

QA76.76.O63M398745 2004
005.3--dc22 2004010590

Text printed on recycled and acid-free paper.
ISBN 0-201-70245-2
1 2 3 4 5 6 7 8 9 10–CRW–0807060504
First Printing, July 2004

Dedication

This book is dedicated to the BSD community.
Without the contributions of that community's members,
there would be nothing about which to write.

Preface

This book follows the earlier authoritative and full-length descriptions of the design and implementation of the 4.3BSD and 4.4BSD versions of the UNIX system developed at the University of California at Berkeley. Since the final Berkeley release in 1994, several groups have continued development of BSD. This book details FreeBSD, the system with the largest set of developers and the most widely distributed releases. Although the FreeBSD distribution includes nearly 1000 utility programs in its base system and nearly 10,000 optional utilities in its ports collection, this book concentrates almost exclusively on the kernel.

UNIX-like Systems

UNIX-like systems include the traditional vendor systems such as Solaris and HP-UX; the Linux-based distributions such as Red Hat, Debian, Suse, and Slackware; and the BSD-based distributions such as FreeBSD, NetBSD, OpenBSD, and Darwin. They run on computers ranging from laptops to the largest supercomputers. They are the operating system of choice for most multiprocessor, graphics, and vector-processing systems, and are widely used for the original purpose of timesharing. The most common platform for providing network services (from FTP to WWW) on the Internet, they are collectively the most portable operating system ever developed. This portability is due partly to their implementation language, C [Kernighan & Ritchie, 1989] (which is itself a widely ported language), and partly to the elegant design of the system.

Since its inception in 1969 [Ritchie & Thompson, 1978], the UNIX system has developed in several divergent and rejoining streams. The original developers continued to advance the state of the art with their Ninth and Tenth Edition UNIX inside AT&T Bell Laboratories, and then their Plan 9 successor to UNIX. Meanwhile, AT&T licensed UNIX System V as a product before selling it to Novell. Novell passed the UNIX trademark to X/OPEN and sold the source code and distribution rights to Santa Cruz Operation (SCO). Both System V and Ninth Edition

UNIX were strongly influenced by the Berkeley Software Distributions produced by the Computer Systems Research Group (CSRG) of the University of California at Berkeley. The Linux operating system, although developed independently of the other UNIX variants, implements the UNIX interface. Thus, applications developed to run on other UNIX-based platforms can be easily ported to run on Linux.

Berkeley Software Distributions

The distributions from Berkeley were the first UNIX-based systems to introduce many important features including the following:

- Demand-paged virtual-memory support
- Automatic configuration of the hardware and I/O system
- A fast and recoverable filesystem
- The socket-based interprocess-communication (IPC) primitives
- The reference implementation of TCP/IP

The Berkeley releases found their way into the UNIX systems of many vendors and were used internally by the development groups of many other vendors. The implementation of the TCP/IP networking protocol suite in 4.2BSD and 4.3BSD, and the availability of those systems, played a key role in making the TCP/IP networking protocol suite a world standard. Even the non-UNIX vendors such as Microsoft have adopted the Berkeley socket design in their Winsock IPC interface.

The BSD releases have also been a strong influence on the POSIX (IEEE Std 1003.1) operating-system interface standard, and on related standards. Several features—such as reliable signals, job control, multiple access groups per process, and the routines for directory operations—have been adapted from BSD for POSIX.

Early BSD releases contained licensed UNIX code, thus requiring recipients to have an AT&T source license to be able to obtain and use BSD. In 1988, Berkeley separated its distribution into AT&T licensed and freely redistributable code. The freely redistributable code was licensed separately and could be obtained, used, and redistributed by anyone. The final freely redistributable 4.4BSD-Lite2 release from Berkeley in 1994 contained nearly the entire kernel and all the important libraries and utilities.

Two groups, NetBSD and FreeBSD, sprang up in 1993 to begin supporting and distributing systems built from the freely redistributable releases being done by Berkeley. The NetBSD group emphasized portability and the minimalist approach, porting the systems to nearly forty platforms and pushing to keep the system lean to aid embedded applications. The FreeBSD group emphasized maximal support for the PC architecture and pushed to ease installation for, and market their system to, as wide an audience as possible. In 1995, the OpenBSD group split from the NetBSD group to develop a distribution that emphasized security. Over the years there has been a healthy competition among the BSD distributions, with many ideas and much code flowing between them.

Material Covered in this Book

This book is about the *internal* structure of the FreeBSD 5.2 kernel and about the concepts, data structures, and algorithms used in implementing FreeBSD's system facilities. Its level of detail is similar to that of Bach's book about UNIX System V [Bach, 1986]; however, this text focuses on the facilities, data structures, and algorithms used in the FreeBSD variant of the UNIX operating system. The book covers FreeBSD from the system-call level down—from the interface to the kernel to the hardware itself. The kernel includes system facilities, such as process management, virtual memory, the I/O system, filesystems, the *socket* IPC mechanism, and network protocol implementations. Material above the system-call level—such as libraries, shells, commands, programming languages, and other user interfaces—is excluded, except for some material related to the terminal interface and to system startup. Following the organization first established by Organick's book about Multics [Organick, 1975], this book is an in-depth study of a contemporary operating system.

Where particular hardware is relevant, the book refers to the Intel Personal Computer (PC) architecture. Because FreeBSD has emphasized development on the PC, that is the architecture with the most complete support, so it provides a convenient point of reference.

Use by Computer Professionals

FreeBSD is widely used to support the core infrastructure of many companies worldwide. Because it can be built with a small footprint, it is also seeing increased use in embedded applications. The licensing terms of FreeBSD do not require the distribution of changes and enhancements to the system. The licensing terms of Linux require that all changes and enhancements to the kernel be made available in source form at minimal cost. Thus, companies that need to control the distribution of their intellectual property build their products using FreeBSD.

This book is of direct use to the professionals who work with FreeBSD systems. Individuals involved in technical and sales support can learn the capabilities and limitations of the system; applications developers can learn how to effectively and efficiently interface to the system; system administrators without direct experience with the FreeBSD kernel can learn how to maintain, tune, and configure the system; and systems programmers can learn how to extend, enhance, and interface to the system.

Readers who will benefit from this book include operating-system implementors, system programmers, UNIX application developers, administrators, and curious users. The book can be read as a companion to the source code of the system, falling as it does between the manual pages and the code in detail of treatment. But this book is neither exclusively a UNIX programming manual nor a user tutorial (for a tutorial, see Libes & Ressler [1988]). Familiarity with the use of some version of the UNIX system (see, for example, Stevens [1992]) and with the C programming language (see, for example, Kernighan & Ritchie [1989]) would be extremely useful.

CHAPTER 1

History and Goals

1.1 History of the UNIX System

The UNIX system has been in wide use for over 30 years and has helped to define many areas of computing. Although numerous individuals and organizations have contributed (and still contribute) to the development of the UNIX system, this book primarily concentrates on the BSD thread of development.

• Bell Laboratories, which invented UNIX

• The Computer Systems Research Group (CSRG) at the University of California at Berkeley, which gave UNIX virtual memory and the reference implementation of TCP/IP

• The FreeBSD project, the NetBSD project, and the OpenBSD project, which continue the work started by the CSRG

• The Darwin operating system at the core of Apple's OS X. Darwin is based on FreeBSD.

Origins

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

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

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

Research UNIX

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

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

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

Most of the system source code was written in C rather than in assembly language. The prevailing belief at the time was that an operating system had to be written in assembly language to provide reasonable efficiency and to get access to the hardware. The C language itself was at a sufficiently high level to allow it to be compiled easily for a wide range of computer hardware, without its being so complex or restrictive that systems programmers had to revert to assembly language to get reasonable efficiency or functionality. Access to the hardware was provided through assembly-language stubs for the 3 percent of the operating-system functions—such as context switching—that needed them. Although the success of UNIX does not stem solely from its being written in a high-level language, the use of C was a critical first step [Kernighan & Ritchie, 1978; Kernighan & Ritchie, 1989; Ritchie et al., 1978]. Ritchie's C language is descended [Rosler, 1984] from Thompson's B language, which was itself descended from BCPL [Richards & Whitby-Strevens, 1980]. C continues to evolve [Tuthill, 1985; ISO, 1999].

The second important distinction of UNIX was its early release from Bell Laboratories to other research environments in source form. By providing source, the system's founders ensured that other organizations would be able not only to use the system, but also to tinker with its inner workings. The ease with which new ideas could be adopted into the system always has been key to the changes that have been made to it. Whenever a new system that tried to upstage UNIX came along, somebody would dissect the newcomer and clone its central ideas into UNIX. The unique ability to use a small, comprehensible system, written in a high-level language, in an environment swimming in new ideas led to a UNIX system that evolved far beyond its humble beginnings. Though recipients of the source code had to be licensed, campuswide licenses were cheaply available to universities. Thus, many people became versed in the way that UNIX worked, setting the stage for the open-source world that would follow.

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

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

AT&T UNIX System III and System V

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

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

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

Berkeley Software Distributions

The most influential of the non-Bell Laboratories and non-AT&T UNIX development groups was the University of California at Berkeley [DiBona et al., 1999]. Software from Berkeley was released in Berkeley Software Distributions (BSD)—for example, as 4.4BSD. Berkeley was the source of the BSD name, and their distributions were the first distinct identity for the BSD operating system. The first Berkeley VAX UNIX work was the addition to 32V of virtual memory, demand paging, and page replacement in 1979 by William Joy and Ozalp Babaoğlu, to produce 3BSD [Babaoğlu & Joy, 1981]. The reason for the large virtual-memory space of 3BSD was the development of what at the time were large programs, such as Berkeley's *Franz* LISP. This memory-management work convinced the Defense Advanced Research Projects Agency (DARPA) to fund the Berkeley team for the later development of a standard system (4BSD) for DARPA's contractors to use.

A goal of the 4BSD project was to provide support for the DARPA Internet networking protocols, TCP/IP [Comer, 2000]. The networking implementation was general enough to communicate among diverse network facilities, ranging from local networks, such as Ethernets and token rings, to long-haul networks, such as DARPA's ARPANET.

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

The 4BSD work for DARPA was guided by a steering committee that included many notable people from both commercial and academic institutions. The culmination of the original Berkeley DARPA UNIX project was the release of 4.2BSD in 1983; further research at Berkeley produced 4.3BSD in mid-1986. The next releases included the 4.3BSD Tahoe release of June 1988 and the 4.3BSD Reno release of June 1990. These releases were primarily ports to the Computer Consoles Incorporated hardware platform. Interleaved with these releases were two unencumbered networking releases: the 4.3BSD Net1 release of March 1989 and the 4.3BSD Net2 release of June 1991. These releases extracted nonproprietary code from 4.3BSD; they could be redistributed freely in source and binary form to companies that and individuals who were not covered by a UNIX source license. The final CSRG release requiring an AT&T source license was 4.4BSD in June 1993. Following a year of litigation (see Section 1.3), the free-redistributable 4.4BSD-Lite was released in April 1994. The final CSRG release was 4.4BSD-Lite Release 2 in June 1995.

UNIX in the World

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

1.2 BSD and Other Systems

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

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

Other UNIX variants have adopted many 4BSD features. AT&T UNIX System V [AT&T, 1987], the IEEE POSIX.1 standard [P1003.1, 1988], and the related

National Bureau of Standards (NBS) Federal Information Processing Standard (FIPS) have adopted the following.

• Job control (Chapter 2)

• Reliable signals (Chapter 4)

• Multiple file-access permission groups (Chapter 6)

• Filesystem interfaces (Chapter 8)

The X/OPEN Group (originally consisting of only European vendors but now including most U.S. UNIX vendors) produced the *X/OPEN Portability Guide* [X/OPEN, 1987] and, more recently, the *Spec 1170 Guide*. These documents specify both the kernel interface and many of the utility programs available to UNIX system users. When Novell purchased UNIX from AT&T in 1993, it transferred exclusive ownership of the UNIX name to X/OPEN. Thus, all systems that want to brand themselves as UNIX must meet the X/OPEN interface specifications. To date, no BSD system has ever been put through the X/OPEN interface-specification tests, so none of them can be called UNIX. The X/OPEN guides have adopted many of the POSIX facilities. The POSIX.1 standard is also an ISO International Standard, named SC22 WG15. Thus, the POSIX facilities have been accepted in most UNIX-like systems worldwide.

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

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

The Influence of the User Community

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

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

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

1.3 The Transition of BSD to Open Source

Up through the release of 4.3BSD Tahoe, all recipients of BSD had to first get an AT&T source license. That was because the BSD systems were never released by Berkeley in a binary-only format; the distributions always contained the complete source to every part of the system. The history of the UNIX system, and the BSD system in particular, had shown the power of making the source available to the users. Instead of passively using the system, they actively worked to fix bugs, improve performance and functionality, and even add completely new features.

With the increasing cost of the AT&T source licenses, vendors that wanted to build stand-alone TCP/IP-based networking products for the PC market using the BSD code found the per-binary costs prohibitive. So they requested that Berkeley break out the networking code and utilities and provide them under licensing terms that did not require an AT&T source license. The TCP/IP networking code clearly did not exist in 32/V and thus had been developed entirely by Berkeley and its contributors. The BSD-originated networking code and supporting utilities were released in June 1989 as Networking Release 1, the first freely redistributable code from Berkeley.

The licensing terms were liberal. A licensee could release the code modified or unmodified in source or binary form with no accounting or royalties to Berkeley. The only requirements were that the copyright notices in the source file be left intact and that products that incorporated the code include in their documentation that the product contained code from the University of California and its contributors. Although Berkeley charged a $1000 fee to get a tape, anyone was free to get a copy from somebody who already had it. Indeed, several large sites put it up for anonymous FTP shortly after it was released. Though the code was freely available, several hundred organizations purchased tapes, which helped to fund the CSRG and encouraged further development.

Networking Release 2

With the success of the first open-source release, the CSRG decided to see how much more of BSD they could spring free. Keith Bostic led the charge by soliciting people to rewrite the UNIX utilities from scratch based solely on their published descriptions. Their only compensation would be to have their name listed among the Berkeley contributors next to the name of the utility that they rewrote. The contributions started slowly and were mostly for the trivial utilities. But as the list of completed utilities grew, and Bostic continued to hold forth for contributions at public events such as Usenix, the rate of contributions continued to grow. Soon the list crossed 100 utilities, and within 18 months nearly all the important utilities and libraries had been rewritten.

The kernel proved to be a bigger task because it could not easily be rewritten from scratch. The entire kernel was reviewed, file by file, removing code that had originated in the 32/V release. When the review was completed, there were only six remaining kernel files that were still contaminated and that could not be trivially rewritten. While consideration was given to rewriting those six files so that a complete kernel could be released, the CSRG decided to release just the less controversial set. The CSRG sought permission for the expanded release from folks higher up in the university administration. After much internal debate and verification of the methods used for detecting proprietary code, the CSRG was given permission to do the release.

The initial thought was to come up with a new name for the second freely redistributable release. However, getting a new license written and approved by the university lawyers would have taken many months. So, the new release was named Networking Release 2, since that could be done with just a revision of the approved Networking Release 1 license agreement. This second greatly expanded freely redistributable release began shipping in June 1991. The redistribution terms and cost were the same as the terms and cost of the first networking release. As before, several hundred individuals and organizations paid the $1000 fee to get the distribution from Berkeley.

Closing the gap from the Networking Release 2 distribution to a fully functioning system did not take long. Within six months of the release, Bill Jolitz had written replacements for the six missing files. He promptly released a fully compiled and bootable system for the 386-based PC architecture in January 1992, which he called 386/BSD. Jolitz's 386/BSD distribution was done almost entirely on the net. He simply put it up for anonymous FTP and let anyone who wanted it download it for free. Within weeks he had a huge following.

Unfortunately, the demands of keeping a full-time job meant that Jolitz could not devote the time needed to keep up with the flood of incoming bug fixes and enhancements to 386/BSD. So within a few months of the release of 386/BSD, a group of avid 386/BSD users formed the NetBSD group to pool their collective resources to help maintain and later enhance the system. By early 1993 they were doing releases that became known as the NetBSD distribution. The NetBSD group chose to emphasize the support of as many platforms as possible and continued the research-style development done by the CSRG. Until 1998, their distribution was done solely over the net with no distribution media available. Their group continues to target primarily the hard-core technical users.

The FreeBSD group was formed a few months after the NetBSD group with a charter to support just the PC architecture and to go after a larger and less technically advanced audience, much as Linux had done. They built elaborate installation scripts and began shipping their system on a low-cost CD-ROM in December 1993. The combination of ease of installation and heavy promotion on the net and at major trade shows, such as Comdex, led to a large, rapid growth curve. FreeBSD quickly rose to have the largest installed base of all the Networking Release 2-derived systems.

FreeBSD also rode the wave of Linux popularity by adding a Linux emulation mode that allows Linux binaries to run on the FreeBSD platform. This feature allows FreeBSD users to use the ever growing set of applications available for Linux while getting the robustness, reliability, and performance of the FreeBSD system.

In 1995, OpenBSD spun off from the NetBSD group. Their technical focus was aimed at improving the security of the system. Their marketing focus was to make the system easier to use and more widely available. Thus, they began producing and selling CD-ROMs, with many of the ease of installation ideas from the FreeBSD distribution.

The Lawsuit

In addition to the groups organized to freely redistribute systems originating from the Networking Release 2 tape, a company, Berkeley Software Design Incorporated (BSDI), was formed to develop and distribute a commercially supported version of the code. Like the other groups, it started by adding the six missing files that Bill Jolitz had written for his 386/BSD release. BSDI began selling its system, including both source and binaries, in January 1992 for $995. It began running advertisements touting its 99 percent discount over the price charged for System V source plus binary systems. Interested readers were told to call 1-800-ITS-UNIX.

Shortly after BSDI began its sales campaign, it received a letter from UNIX System Laboratory (USL) (a mostly owned subsidiary of AT&T spun off to develop and sell UNIX) [Ritchie, 2004]. The letter demanded that BSDI stop promoting its product as UNIX and in particular that it stop using the deceptive phone number. Although the phone number was promptly dropped and the advertisements changed to explain that the product was not UNIX, USL was still unhappy and filed suit to enjoin BSDI from selling its product. The suit alleged that the BSDI product contained USL proprietary code and trade secrets. USL sought to get an injunction to halt BSDI's sales until the lawsuit was resolved claiming that it would suffer irreparable harm from the loss of its trade secrets if the BSDI distributions continued.

At the preliminary hearing for the injunction, BSDI contended that it was simply using the sources being freely distributed by the University of California plus six additional files. BSDI was willing to discuss the content of any of the six added files but did not believe it should be held responsible for the files being

[illegible] Proceedings [illegible], 1989.

Organick 1972

E. I. Organick, *The Multics System: An Examination of Its Structure*, MIT Press, Cambridge, MA, 1972.

[illegible]

[illegible] Mach Approach [illegible] Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, December 1987.

Young 1989

M. W. Young, [illegible] *Memory Management* [illegible] Report CMU-CS-89-202, Department of Computer Science, Carnegie-Mellon University, November 1989.

CHAPTER 6

I/O System Overview

6.1 I/O Mapping from User to Device

Computers store and retrieve data through supporting peripheral I/O devices. These devices typically include mass-storage devices, such as disk drives, archival-storage devices, and network interfaces. Storage devices such as disks are accessed through I/O controllers that manage the operation of their attached devices according to I/O requests from the CPU.

Many hardware device peculiarities are hidden from the user by high-level kernel facilities, such as the filesystem and socket interfaces. Other such peculiarities are hidden from the bulk of the kernel itself by the I/O system. The I/O system consists of buffer-caching systems, general device-driver code, and drivers for specific hardware devices that must finally address peculiarities of the specific devices. An overview of the entire kernel is shown in Figure 6.1 (on page 216). The bottom third of the figure comprises the various I/O systems.

There are three main kinds of I/O in FreeBSD: the *character-device* interface, the *filesystem*, and the *socket* interface with its related network devices. The character interface appears in the filesystem name space and provides *unstructured* access to the underlying hardware. The network devices do not appear in the filesystem; they are accessible through only the socket interface. Character devices are described in Section 6.2. The filesystem is described in Chapter 8. Sockets are described in Chapter 11.

A character-device interface comes in two styles that depend on the characteristics of the underlying hardware device. For some character-oriented hardware devices, such as terminal multiplexers, the interface is truly character oriented, although higher-level software, such as the terminal driver, may provide a line-oriented interface to applications. However, for block-oriented devices such as disks, a character-device interface is an *unstructured* or *raw* interface. For this

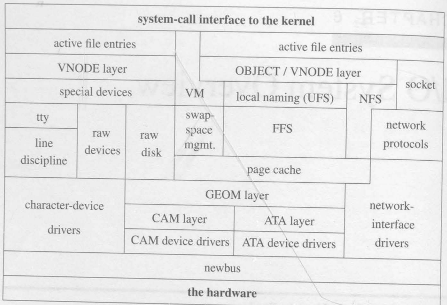

Figure 6.1 Kernel I/O structure.

interface, I/O operations do not go through the filesystem or the page cache; instead, they are made directly between the device and buffers in the application's virtual address space. Consequently, the size of the operations must be a multiple of the underlying *block size* required by the device, and, on some machines, the application's I/O buffer must be aligned on a suitable boundary.

Internal to the system, I/O devices are accessed through a set of entry points provided by each device's *device driver*. For a character-device interface, it accesses a *cdevsw* structure. A *cdevsw* structure is created for each device as the device is configured either at the time that the system is booted or later when the device is attached to the system.

Devices are identified by a *device number* that is constructed from a *major* and a *minor* device number. The *major device number* uniquely identifies the type of device (really of the device driver). Historically it was used as the index of the device's entry in the character-device table. FreeBSD 5.2 has no character-device table. As devices are configured, entries are created for the device in the **/dev** filesystem. Each entry in the **/dev** filesystem has a direct reference to its corresponding *cdevsw* entry. FreeBSD 5.2 assigns a unique major device number to each device when it is configured to provide compatibility for applications that look at it. But it is not used internally by the kernel or the device driver.

The *minor device* number is selected and interpreted solely by the device driver and is used by the driver to identify to which, of potentially many, hardware devices an I/O request refers. For disks, for example, minor device numbers identify a specific controller, disk drive, and partition. The minor device number may also specify a section of a device—for example, a channel of a multiplexed device, or optional handling parameters.

Device Drivers

A device driver is divided into three main sections:

1. Autoconfiguration and initialization routines
2. Routines for servicing I/O requests (the top half)
3. Interrupt service routines (the bottom half)

The autoconfiguration portion of a driver is responsible for *probing* for a hardware device to see whether the latter is present and to initialize the device and any associated software state that is required by the device driver. This portion of the driver is typically called only once, either when the system is initialized or for transient devices when they are connected to the system. Autoconfiguration is described in Section 14.4.

The section of a driver that services I/O requests is invoked because of system calls or by the virtual-memory system. This portion of the device driver executes synchronously in the top half of the kernel and is permitted to block by calling the *sleep*() routine. We commonly refer to this body of code as the *top half* of a device driver.

Interrupt service routines are invoked when the system fields an interrupt from a device. Consequently, these routines cannot depend on any per-process state. Historically they did not have a thread context of their own, so they could not block. In FreeBSD 5.2 an interrupt has its own thread context, so it can block if it needs to do so. However, the cost of extra thread switches is sufficiently high that for good performance device drivers should attempt to avoid blocking. We commonly refer to a device driver's interrupt service routines as the *bottom half* of a device driver.

In addition to these three sections of a device driver, an optional *crash-dump* routine may be provided. This routine, if present, is invoked when the system recognizes an unrecoverable error and wishes to record the contents of physical memory for use in postmortem analysis. Most device drivers for disk controllers provide a crash-dump routine. The use of the crash-dump routine is described in Section 14.6.

I/O Queueing

Device drivers typically manage one or more queues of I/O requests in their normal operation. When an input or output request is received by the top half of the driver, it is recorded in a data structure that is placed on a per-device queue for processing. When an input or output operation completes, the device driver

• A pointer to the thread whose data area is described by the *uio* structure (the pointer is NULL if the *uio* structure describes an area within the kernel)

All I/O within the kernel is described with *iovec* and *uio* structures. System calls such as *read* and *write* that are not passed an *iovec* create a *uio* to describe their arguments; this *uio* structure is passed to the lower levels of the kernel to specify the parameters of an I/O operation. Eventually, the *uio* structure reaches the part of the kernel responsible for moving the data to or from the process address space: the filesystem, the network, or a device driver. In general, these parts of the kernel do not interpret *uio* structures directly. Instead, they arrange a kernel buffer to hold the data and then use *uiomove*() to copy the data to or from the buffer or buffers described by the *uio* structure. The *uiomove*() routine is called with a pointer to a kernel data area, a data count, and a *uio* structure. As it moves data, it updates the counters and pointers of the *iovec* and *uio* structures by a corresponding amount. If the kernel buffer is not as large as the areas described by the *uio* structure, the *uio* structure will point to the part of the process address space just beyond the location completed most recently. Thus, while servicing a request, the kernel may call *uiomove*() multiple times, each time giving a pointer to a new kernel buffer for the next block of data.

Character device drivers that do not copy data from the process generally do not interpret the *uio* structure. Instead, there is one low-level kernel routine that arranges a direct transfer to or from the address space of the process. Here, a separate I/O operation is done for each *iovec* element, calling back to the driver with one piece at a time.

6.5 The Virtual-Filesystem Interface

In early UNIX systems, the file entries directly referenced the local filesystem *inode*. An inode is a data structure that describes the contents of a file; it is more fully described in Section 8.2. This approach worked fine when there was a single filesystem implementation. However, with the advent of multiple filesystem types, the architecture had to be generalized. The new architecture had to support importing of filesystems from other machines including other machines that were running different operating systems.

One alternative would have been to connect the multiple filesystems into the system as different file types. However, this approach would have required massive restructuring of the internal workings of the system, because current directories, references to executables, and several other interfaces used inodes instead of file entries as their point of reference. Thus, it was easier and more logical to add a new object-oriented layer to the system below the file entry and above the inode. This new layer was first implemented by Sun Microsystems, which called it the virtual-node, or *vnode*, layer. Interfaces in the system that had referred previously to inodes were changed to reference generic vnodes. A vnode used by a local filesystem would refer to an inode. A vnode used by a remote filesystem would refer to a protocol control block that described the location and naming information necessary to access the remote file.

Contents of a Vnode

The vnode is an extensible object-oriented interface. It contains information that is generically useful independent of the underlying filesystem object that it represents. The information stored in a vnode includes the following:

- Flags are used for identifying generic attributes. An example generic attribute is a flag to show that a vnode represents an object that is the root of a filesystem.
- The various reference counts include the number of file entries that are open for reading and/or writing that reference the vnode, the number of file entries that are open for writing that reference the vnode, and the number of pages and buffers that are associated with the vnode.
- A pointer to the mount structure describes the filesystem that contains the object represented by the vnode.
- Various information to do file read-ahead.
- A reference to the *vm_object* associated with the vnode.
- A reference to state about special devices, sockets, and fifos.
- A mutex to protect the flags and counters within the vnode.
- A lock-manager lock to protect parts of the vnode that may change while it has an I/O operation in progress.
- Fields used by the name cache to track the names associated with the vnode.
- A pointer to the set of vnode operations defined for the object. These operations are described in the next subsection.
- A pointer to private information needed for the underlying object. For the local filesystem, this pointer will reference an inode; for NFS, it will reference an nfsnode.
- The type of the underlying object (e.g., regular file, directory, character device, etc.) is given. The type information is not strictly necessary, since a vnode client could always call a vnode operation to get the type of the underlying object. However, because the type often is needed, the type of underlying objects does not change, and it takes time to call through the vnode interface, the object type is cached in the vnode.
- There are clean and dirty buffers associated with the vnode. Each valid buffer in the system is identified by its associated vnode and the starting offset of its data within the object that the vnode represents. All the buffers that have been modified but have not yet been written back are stored on their vnode dirty-buffer list. All buffers that have not been modified or have been written back since they

were last modified are stored on their vnode clean list. Having all the dirty buffers for a vnode grouped onto a single list makes the cost of doing an *fsync* system call to flush all the dirty blocks associated with a file proportional to the amount of dirty data. In some UNIX systems, the cost is proportional to the smaller of the size of the file or the size of the buffer pool. The list of clean buffers is used to free buffers when a file is deleted. Since the file will never be read again, the kernel can immediately cancel any pending I/O on its dirty buffers and reclaim all its clean and dirty buffers and place them at the head of the buffer free list, ready for immediate reuse.

- A count is kept of the number of buffer write operations in progress. To speed the flushing of dirty data, the kernel does this operation by doing asynchronous writes on all the dirty buffers at once. For local filesystems, this simultaneous push causes all the buffers to be put into the disk queue so that they can be sorted into an optimal order to minimize seeking. For remote filesystems, this simultaneous push causes all the data to be presented to the network at once so that it can maximize their throughput. System calls that cannot return until the data are on stable store (such as *fsync*) can sleep on the count of pending output operations, waiting for the count to reach zero.

The position of vnodes within the system was shown in Figure 6.1. The vnode itself is connected into several other structures within the kernel, as shown in Figure 6.7. Each mounted filesystem within the kernel is represented by a generic mount structure that includes a pointer to a filesystem-specific control block. All the vnodes associated with a specific mount point are linked together on a list headed by this generic mount structure. Thus, when it is doing a *sync* system call for a filesystem, the kernel can traverse this list to visit all the files active within that filesystem. Also shown in the figure are the lists of clean and dirty buffers associated with each vnode. Finally, there is a free list that links together all the vnodes in the system that are inactive (not currently referenced). The free list is used when a filesystem needs to allocate a new vnode so that the latter can open a new file; see Section 6.4.

Vnode Operations

Vnodes are designed as an object-oriented interface. Thus, the kernel manipulates them by passing requests to the underlying object through a set of defined operations. Because of the many varied filesystems that are supported in FreeBSD, the set of operations defined for vnodes is both large and extensible. Unlike the original Sun Microsystems vnode implementation, the one in FreeBSD allows dynamic addition of vnode operations either at system boot time or when a new filesystem is dynamically loaded into the kernel. As part of activating a filesystem, it registers the set of vnode operations that it is able to support. The kernel then builds a table that lists the union of all operations supported by any filesystem. From that table, it builds an operations vector for each filesystem. Supported operations are filled in with the entry point registered by the filesystem. Filesystems may opt to

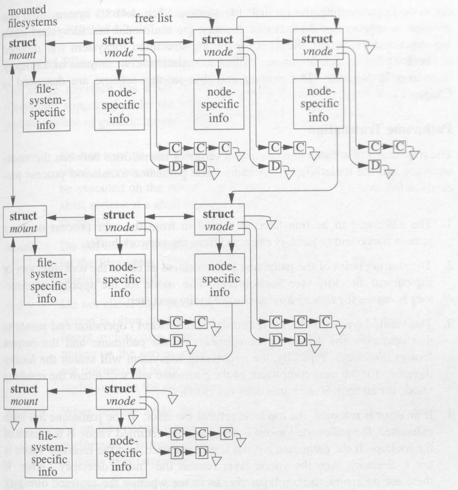

Figure 6.7 Vnode linkages. Key: D—dirty buffer; C—clean buffer.

have unsupported operations filled in with either a default routine (typically a routine to bypass the operation to the next lower layer; see Section 6.7) or a routine that returns the characteristic error "operation not supported" [Heidemann & Popek, 1994].

In 4.3BSD, the local filesystem code provided both the semantics of the hierarchical filesystem naming and the details of the on-disk storage management. These functions are only loosely related. To enable experimentation with other disk-storage techniques without having to reproduce the entire naming semantics, 4.4BSD split the naming and storage code into separate modules. The vnode-level operations define a set of hierarchical filesystem operations. Below the naming layer are a separate set of operations defined for storage of variable-sized objects using a flat name space. About 60 percent of the traditional filesystem code became the name-space management, and the remaining 40 percent became the code implementing the on-disk file storage. The 4.4BSD system used this division to support two distinct disk layouts: the traditional fast filesystem and a log-structured filesystem. Support for the log-structured filesystem was dropped in FreeBSD due to lack of anyone willing to maintain it but remains as a primary filesystem in NetBSD. The naming and disk-storage scheme are described in Chapter 8.

Pathname Translation

The translation of a pathname requires a series of interactions between the vnode interface and the underlying filesystems. The pathname-translation process proceeds as follows:

1. The pathname to be translated is copied in from the user process or, for a remote filesystem request, is extracted from the network buffer.
2. The starting point of the pathname is determined as either the root directory or the current directory (see Section 2.7). The vnode for the appropriate directory becomes the *lookup directory* used in the next step.
3. The vnode layer calls the filesystem-specific *lookup*() operation and passes to that operation the remaining components of the pathname and the current *lookup directory*. Typically, the underlying filesystem will search the *lookup directory* for the next component of the pathname and will return the resulting vnode (or an error if the name does not exist).
4. If an error is returned, the top level returns the error. If the pathname has been exhausted, the pathname lookup is done, and the returned vnode is the result of the lookup. If the pathname has not been exhausted, and the returned vnode is not a directory, then the vnode layer returns the "not a directory" error. If there are no errors, the top layer checks to see whether the returned directory is a mount point for another filesystem. If it is, then the *lookup directory* becomes the mounted filesystem; otherwise, the *lookup directory* becomes the vnode returned by the lower layer. The lookup then iterates with step 3.

Although it may seem inefficient to call through the vnode interface for each pathname component, doing so usually is necessary. The reason is that the underlying filesystem does not know which directories are being used as mount points. Since a mount point will redirect the lookup to a new filesystem, it is important that the current filesystem not proceed past a mounted directory. Although it might be possible for a local filesystem to be knowledgeable about which directories are mount points, it is nearly impossible for a server to know which of the directories within its exported filesystems are being used as mount points by its clients. Consequently, the conservative approach of traversing only a single pathname component per *lookup*() call is used. There are a few instances where a filesystem will know that there are no further mount points in the remaining path, and will traverse the rest of the pathname. An example is crossing into a *portal*, described in Section 6.7.

Exported Filesystem Services

The vnode interface has a set of services that the kernel exports from all the filesystems supported under the interface. The first of these is the ability to support the update of generic mount options. These options include the following:

noexec	Do not execute any files on the filesystem. This option is often used when a server exports binaries for a different architecture that cannot be executed on the server itself. The kernel will even refuse to execute shell scripts; if a shell script is to be run, its interpreter must be invoked explicitly.
nosuid	Do not honor the set-user-id or set-group-id flags for any executables on the filesystem. This option is useful when a filesystem of unknown origin is mounted.
nodev	Do not allow any special devices on the filesystem to be opened. This option is often used when a server exports device directories for a different architecture. The filesystem would be mounted with the *nodev* option on the server, since the values of the major and minor numbers are nonsensical to the server. The major and minor numbers are meaningful only on the clients that import them.
noatime	When reading a file, do not update its access time. This option is useful on filesystems where there are many files being frequently read and performance is more critical than updating the file access time (which is rarely ever important).
sync	Request that all I/O to the file system be done synchronously.

It is not necessary to unmount and remount the filesystem to change these flags; they may be changed while a filesystem is mounted. In addition, a filesystem that is mounted read-only can be upgraded to allow writing. Conversely, a filesystem that allows writing may be downgraded to read-only provided that no files are open for modification. The system administrator can forcibly downgrade the filesystem to read-only by requesting that any files open for writing have their access revoked.

Another service exported from the vnode interface is the ability to get information about a mounted filesystem. The *statfs* system call returns a buffer that gives the numbers of used and free disk blocks and inodes, along with the filesystem mount point, and the device, location, or program from which the filesystem is mounted. The *getfsstat* system call returns information about all the mounted filesystems. This interface avoids the need to track the set of mounted filesystems outside the kernel, as is done in many other UNIX variants.

CHAPTER 8

Local Filesystems

8.1 Hierarchical Filesystem Management

The operations defined for local filesystems are divided into two parts. Common to all local filesystems are hierarchical naming, locking, quotas, attribute management, and protection. These features, which are independent of how data are stored, are provided by the UFS code described in the first seven sections of this chapter. The other part of the local filesystem, the filestore, is concerned with the organization and management of the data on the storage media. Storage is managed by the datastore filesystem operations described in the final two sections of this chapter.

The vnode operations defined for doing hierarchical filesystem operations are shown in Table 8.1 (on page 296). The most complex of these operations is that for doing a lookup. The filesystem-independent part of the lookup is described in Section 6.5. The algorithm used to lookup a pathname component in a directory is described in Section 8.3.

There are five operators for creating names. The operator used depends on the type of object being created. The *create* operator creates regular files and also is used by the networking code to create AF_LOCAL domain sockets. The *link* operator creates additional names for existing objects. The *symlink* operator creates a symbolic link (see Section 8.3 for a discussion of symbolic links). The *mknod* operator creates character special devices (for compatibility with other UNIX systems that still use them); it is also used to create fifos. The *mkdir* operator creates directories.

There are three operators for changing or deleting existing names. The *rename* operator deletes a name for an object in one location and creates a new name for the object in another location. The implementation of this operator is complex when the kernel is dealing with the movement of a directory from one part of the filesystem tree to another. The *remove* operator removes a name. If the

Table 8.1 Hierarchical filesystem operations.

Operation done	Operator names
pathname searching	lookup
name creation	create, mknod, link, symlink, mkdir
name change/deletion	rename, remove, rmdir
attribute manipulation	access, getattr, setattr
object interpretation	open, readdir, readlink, mmap, close
process control	advlock, ioctl, poll
object management	lock, unlock, inactive, reclaim

removed name is the final reference to the object, the space associated with the underlying object is reclaimed. The *remove* operator operates on all object types except directories; they are removed using the *rmdir* operator.

Three operators are supplied for object attributes. The kernel retrieves attributes from an object using the *getattr* operator and stores them using the *setattr* operator. Access checks for a given user are provided by the *access* operator.

Five operators are provided for interpreting objects. The *open* and *close* operators have only peripheral use for regular files, but when they are used on special devices, they notify the appropriate device driver of device activation or shutdown. The *readdir* operator converts the filesystem-specific format of a directory to the standard list of directory entries expected by an application. Note that the interpretation of the contents of a directory is provided by the hierarchical filesystem-management layer; the filestore code considers a directory as just another object holding data. The *readlink* operator returns the contents of a symbolic link. As with directories, the filestore code considers a symbolic link as just another object holding data. The *mmap* operator prepares an object to be mapped into the address space of a process.

Three operators are provided to allow process control over objects. The *poll* operator allows a process to find out whether an object is ready to be read or written. The *ioctl* operator passes control requests to a special device. The *advlock* operator allows a process to acquire or release an advisory lock on an object. None of these operators modifies the object in the filestore. They are simply using the object for naming or directing the desired operation.

There are four operations for management of the objects. The *inactive* and *reclaim* operators were described in Section 6.6. The *lock* and *unlock* operators allow the callers of the vnode interface to provide hints to the code that implement operations on the underlying objects. Stateless filesystems such as NFS ignore these hints. Stateful filesystems, however, can use hints to avoid doing extra work. For example, an *open* system call requesting that a new file be created requires two steps. First, a *lookup* call is done to see if the file already exists. Before the lookup is started, a *lock* request is made on the directory being searched. While scanning through the directory checking for the name, the lookup code also identifies a location within the directory that contains enough space to hold the new name. If the lookup returns successfully (meaning that the name does not already exist), the *open* code verifies that the user has permission to create the file. If the caller is not eligible to create the new file, then they are expected to call *unlock* to release the lock that they acquired during the lookup. Otherwise, the *create* operation is called. If the filesystem is stateful and has been able to lock the directory, then it can simply create the name in the previously identified space, because it knows that no other processes will have had access to the directory. Once the name is created, an *unlock* request is made on the directory. If the filesystem is stateless, then it cannot lock the directory, so the *create* operator must rescan the directory to find space and to verify that the name has not been created since the lookup.

8.2 Structure of an Inode

To allow files to be allocated concurrently and to provide random access within files, FreeBSD uses the concept of an *index node*, or *inode*. The inode contains information about the contents of the file; see Figure 8.1 (on page 298). This information includes the following:

- The type and access mode for the file
- The file's owner and group-access identifiers
- The time that the file was created, when it was most recently read and written, and when its inode was most recently updated by the system
- The size of the file in bytes
- The number of physical blocks used by the file (including blocks used to hold indirect pointers and extended attributes)
- The number of directory entries that reference the file
- The kernel and user setable flags that describe characteristics of the file
- The generation number of the file (a randomly selected number assigned to the inode each time that the latter is allocated to a new file; the generation number is used by NFS to detect references to deleted files)
- The blocksize of the data blocks referenced by the inode (typically the same as, but sometimes larger than, the filesystem blocksize)
- The size of the extended attribute information

Notably missing in the inode is the filename. Filenames are maintained in directories rather than in inodes because a file may have many names, or links, and the name of a file can be large (up to 255 bytes in length). Directories are described in Section 8.3.

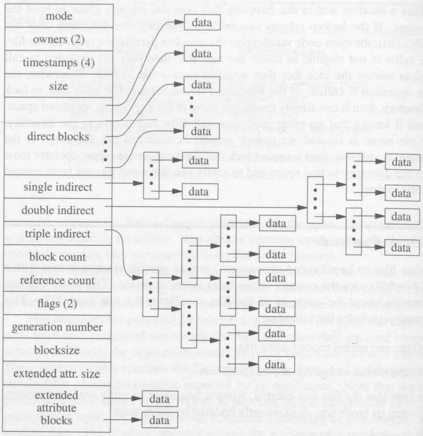

Figure 8.1 The structure of an inode.

To create a new name for a file, the system increments the count of the number of names referring to that inode. Then the new name is entered in a directory, along with the number of the inode. Conversely, when a name is deleted, the entry is deleted from a directory, and the name count for the inode is then decremented. When the name count reaches zero, the system deallocates the inode by putting all the inode's blocks back on a list of free blocks.

The inode also contains an array of pointers to the blocks in the file. The system can convert from a logical block number to a physical sector number by indexing into the array using the logical block number. A null array entry shows that no block has been allocated and will cause a block of zeros to be returned on a read. On a write of such an entry, a new block is allocated, the array entry is updated with the new block number, and the data are written to the disk.

Inodes are fixed in size, and most files are small, so the array of pointers must be small for efficient use of space. The first 12 array entries are allocated in the inode itself. For typical filesystems, this implementation allows the first 96 or 192 Kbyte of data to be located directly via a simple indexed lookup.

For somewhat larger files, Figure 8.1 shows that the inode contains a *single indirect pointer* that points to a *single indirect block* of pointers to data blocks. To find the 100th logical block of a file, the system first fetches the block identified by the indirect pointer and then indexes into the 88th block (100 minus 12 direct pointers) and fetches that data block.

For files that are bigger than a few megabytes, the single indirect block is eventually exhausted; these files must resort to using a *double indirect block*, which is a pointer to a block of pointers to pointers to data blocks. For files of multiple Gbyte, the system uses a *triple indirect block*, which contains three levels of pointer before reaching the data block.

Although indirect blocks appear to increase the number of disk accesses required to get a block of data, the overhead of the transfer is typically much lower. In Section 6.6, we discussed the management of the cache that holds recently used disk blocks. The first time that a block of indirect pointers is needed, it is brought into the cache. Further accesses to the indirect pointers find the block already resident in memory; thus, they require only a single disk access to get the data.

Changes to the Inode Format

Traditionally, the FreeBSD fast filesystem (which we shall refer to in this book as UFS1) [McKusick et al., 1984] and its derivatives have used 32-bit pointers to reference the blocks used by a file on the disk. The UFS1 filesystem was designed in the early 1980s when the largest disks were 330 Mbytes. There was debate at the time whether it was worth squandering 32 bits per block pointer rather than using the 24-bit block pointers of the filesystem that it replaced. Luckily the futurist view prevailed, and the design used 32-bit block pointers. Over the 20 years since it has been deployed, storage systems have grown to hold over a terabyte of data. Depending on the block size configuration, the 32-bit block pointers of UFS1 run out of space in the 1 to 4 terabyte range. While some stopgap measures can be used to extend the maximum-size storage systems supported by UFS1, by 2002 it became clear the only long-term solution was to use 64-bit block pointers. Thus, we decided to build a new filesystem, UFS2, that would use 64-bit block pointers.

We considered the alternatives between trying to make incremental changes to the existing UFS1 filesystem versus importing another existing filesystem such as XFS [Sweeney et al., 1996], or ReiserFS [Reiser, 2001]. We also considered writing a new filesystem from scratch so that we could take advantage of recent filesystem research and experience. We chose to extend the UFS1 filesystem because this approach allowed us to reuse most of the existing UFS1 code base. The benefits of this decision were that UFS2 was developed and deployed quickly, it became stable and reliable rapidly, and the same code base could be used to support both UFS1 and UFS2 filesystem formats. Over 90 percent of the code base is shared, so bug fixes and feature or performance enhancements usually apply to both filesystem formats.

The on-disk inodes used by UFS1 are 128 bytes in size and have only two unused 32-bit fields. It would not be possible to convert to 64-bit block pointers without reducing the number of direct block pointers from 12 to 5. Doing so would dramatically increase the amount of wasted space, since only direct block pointers can reference fragments, so the only alternative is to increase the size of the on-disk inode to 256 bytes.

Once one is committed to changing to a new on-disk format for the inodes, it is possible to include other inode-related changes that were not possible within the constraints of the old inodes. While it was tempting to throw in everything that has ever been suggested over the last 20 years, we felt that it was best to limit the addition of new capabilities to those that were likely to have a clear benefit. Every new addition adds complexity that has a cost both in maintainability and performance. Obscure or little-used features may add conditional checks in frequently executed code paths such as read and write, slowing down the overall performance of the filesystem even if they are not used.

Extended Attributes

A major addition in UFS2 is support for extended attributes. Extended attributes are a piece of auxiliary data storage associated with an inode that can be used to store auxiliary data that is separate from the contents of the file. The idea is similar to the concept of data forks used in the Apple filesystem [Apple, 2003]. By integrating the extended attributes into the inode itself, it is possible to provide the same integrity guarantees as are made for the contents of the file itself. Specifically, the successful completion of an *fsync* system call ensures that the file data, the extended attributes, and all names and paths leading to the names of the file are in stable store.

The current implementation has space in the inode to store up to two blocks of extended attributes. The new UFS2 inode format had room for up to five additional 64-bit pointers. Thus, the number of extended attribute blocks could have been between one to five blocks. We chose to allocate two blocks to the extended attributes and to leave the other three as spares for future use. By having two, all the code had to be prepared to deal with an array of pointers, so if the number got expanded into the remaining spares in the future, the existing implementation will work without changes to the source code. By saving three spares, we provided a reasonable amount of space for future needs. And if the decision to allow only two blocks proves to be too little space, one or more of the spares can be used to expand the size of the extended attributes in the future. If vastly more extended attribute space is needed, a spare could be used as an indirect pointer to extended attribute data blocks.

Figure 8.2 shows the format used for the extended attributes. The first field of the header of each attribute is its length. Applications that do not understand the name space or name can simply skip over the unknown attribute by adding the

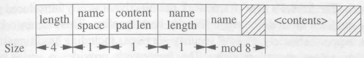

Figure 8.2 Format of extended attributes. The header of each attribute has a 4-byte length, 1-byte name-space class, 1-byte content pad length, 1-byte name length, and name. The name is padded so that the contents start on an 8-byte boundary. The contents are padded to the size shown by the "content pad length" field. The size of the contents can be calculated by subtracting from the length the size of the header (including the name) and the content pad length.

length to their current position to get to the next attribute. Thus, many different applications can share the usage of the extended attribute space, even if they do not understand each other's data types.

The first of two initial uses for extended attributes is to support an *access control list*, generally referred to as an ACL. An ACL replaces the group permissions for a file with a more specific list of the users that are permitted to access the files. The ACL also includes a list of the permissions that each user is granted. These permissions include the traditional read, write, and execute permissions along with other properties such as the right to rename or delete the file [Rhodes, 2003].

Earlier implementations of ACLs were done with a single auxiliary file per filesystem that was indexed by the inode number and had a small fixed-sized area to store the ACL permissions. The small size was to keep the size of the auxiliary file reasonable, since it had to have space for every possible inode in the filesystem. There were two problems with this implementation. The fixed size of the space per inode to store the ACL information meant that it was not possible to give access to long lists of users. The second problem was that it was difficult to atomically commit changes to the ACL list for a file, since an update required that both the file inode and the ACL file be written to have the update take effect [Watson, 2000].

Both problems with the auxiliary file implementation of ACLs are fixed by storing the ACL information directly in the extended-attribute data area of the inode. Because of the large size of the extended attribute data area (a minimum of 8 Kbytes and typically 32 Kbytes), long lists of ACL information can be easily stored. Space used to store extended attribute information is proportional to the number of inodes with extended attributes and the size of the ACL lists that they use. Atomic update of the information is much easier, since writing the inode will update the inode attributes and the set of data that it references including the extended attributes in one disk operation. While it would be possible to update the old auxiliary file on every *fsync* system call done on the filesystem, the cost of doing so would be prohibitive. Here, the kernel knows if the extended attribute data block for an inode is dirty and can write just that data block during an *fsync* call on the inode.

The second use for extended attributes is for data labeling. Data labels provide permissions for a *mandatory access control* (MAC) framework enforced by a copy is made; the modification is made to the copy rather than to the original. In virtual-memory management, copy-on-write is a common scheme that the kernel uses to manage pages shared by multiple processes. All the page-table entries mapping a shared page are set such that the first write reference to the page causes a page fault. When the page fault is serviced, the faulted page is replaced with a private copy, which is writable.

core file A file (named **procname.core**) that is created by the system when certain signals are delivered to a process. The file contains a record of the state of the process at the time the signal occurred. This record includes the contents of the process's virtual address space and, on most systems, the user structure.

CPU See *central processing unit.*

crash Among computer scientists, an unexpected system failure.

crash dump A record of the state of a machine at the time of a crash. This record is usually written to a place on secondary storage that is thought to be safe so that it can be saved until the information can be recovered.

credential A structure that identifies a user. It contains the real, effective, and saved user and group identifiers. See also *real user identifier; real group identifier; effective user identifier; effective group identifier; saved UID; saved GID.*

current working directory The directory from which relative pathnames are interpreted for a process. The current working directory for a process is set with the *chdir* or *fchdir* system call.

cylinder group In the Fast Filesystem, a collection of blocks on a disk drive that is grouped together to use localizing information. That is, the filesystem allocates inodes and data blocks on a per-cylinder-group basis. Cylinder group is a historic name from the days when the geometry of disks was known.

daemon A long-lived process that provides a system-related service. There are daemon processes that execute in kernel mode (e.g., the *pagedaemon*) and daemon processes that execute in user mode (e.g., the *routing daemon*). The Old English term *daemon* means "a deified being," as distinguished from the term *demon*, which means "an evil spirit."

DARPA Defense Advanced Research Projects Agency. An agency of the U.S. Department of Defense that is responsible for managing defense-sponsored research in the United States.

datagram socket A type of socket that supports an unreliable message transport that preserves message boundaries.

data segment The segment of a process's address space that contains the initialized and uninitialized data portions of a program. See also *bss segment; stack segment; text segment.*

decapsulation In network communication, the removal of the outermost header information from a message. See also *encapsulation.*

Page 42 of 43

demand paging A memory-management technique in which memory is divided into pages and the pages are provided to processes as needed—that is, *on demand*. See also *pure demand paging*.

demon See *daemon*.

denial of service attack Any attempt to overload a system such that it is unable to do work for legitimate users of the system. For example, sending a system so many packets that it runs out of mbufs and so cannot process any other network traffic.

descriptor An integer assigned by the system when a file is referenced by the *open* system call or when a socket is created with the *socket*, *pipe*, or *socketpair* system calls. The integer uniquely identifies an access path to the file or socket from a given process or from any of that process's children. Descriptors can also be duplicated with the *dup* and *fcntl* system calls.

descriptor table A per-process table that holds references to objects on which I/O may be done. I/O descriptors are indices into this table.

device In UNIX, a peripheral connected to the CPU.

device driver A software module that is part of the kernel and that supports access to a peripheral device.

device flags Data specified in a system configuration file and passed to a device driver. The use of these flags varies across device drivers. Device drivers for terminal devices use the flags to indicate the terminal lines on which the driver should ignore modem-control signals on input.

device number A number that uniquely identifies a device within the character-device class. Historically, a device number comprises two parts: a major device number and a minor device number. In FreeBSD 5.2, device numbers are assigned dynamically and are used only for backward compatibility with older applications.

device special file A file through which processes can access hardware devices on a machine. For example, a sound card is accessed through such a file.

direct memory access (DMA) A facility whereby a peripheral device can access main memory without the assistance of the CPU. DMA is typically used to transfer contiguous blocks of data between main memory and a peripheral device.

directory In UNIX, a special type of file that contains entries that are references to other files. By convention, a directory contains at least two entries: dot (.) and dot-dot (..). Dot refers to the directory itself; dot-dot refers to the parent directory.

directory entry An entry that is represented by a variable-length record structure in the directory file. Each structure holds an ASCII string that represents the filename, the number of bytes of space provided for the string, the number of bytes of space provided for the entry, the type of the file referenced by the entry, and the number of the inode associated with the filename. By convention, a directory entry with a zero inode number is treated as unallocated, and the space held by the entry is available for use.