

Google Exhibit 1120
Google v. VirtaMove

Oracle Ex. 1120
Oracle v. VirtaMove, IPR2025-00965



THE DESIGN OF THE UNIX® OPERATING SYSTEM

Maurice J. Bach

PRENTICE-HALL, INC., Englewood Cliffs, New Jersey 07632

Copyright © 1986 by Bell Telephone Laboratories, Incorporated.

Published by Prentice-Hall, Inc.
A division of Simon & Schuster
Englewood Cliffs, New Jersey 07632

Prentice-Hall Software Series
Brian W. Kernighan, Advisor

Cover Design Consultant: Sarah Bach

UNIX® is a registered trademark of AT&T.
DEC, PDP, and VAX are trademarks of Digital Equipment Corp.
Series 32000 is a trademark of National Semiconductor Corp.
®Ada is a registered trademark of the U.S. Government (Ada Joint Program Office).
UNIVAC is a trademark of Sperry Corp.
This document was set on an AUTOLOGIC, Inc. APS-5 phototypesetter driven by the TROFF formatter operating under the UNIX system on an AT&T 3B20 computer.

The Publisher offers discounts on this book when ordered in bulk quantities. For more information write:

Special Sales/College Marketing
Prentice-Hall, Inc.
College Technical and Reference Division
Englewood Cliffs, New Jersey 07632

The author and publisher of this book have used their best efforts in preparing this book. These efforts include the development, research, and testing of the theories and programs to determine their effectiveness. The author and publisher make no warranty of any kind, expressed or implied, with regard to these programs or the documentation contained in this book. The author and publisher shall not be liable in any event for incidental or consequential damages in connection with, or arising out of, the furnishing, performance, or use of these programs.

All rights reserved. No part of this book may be reproduced, in any form or by any means, without permission in writing from the publisher.

Printed in the United States of America

10

ISBN 0-13-201799-7 025

Prentice-Hall International (UK) Limited, *London*
Prentice-Hall of Australia Pty. Limited, *Sydney*
Prentice-Hall Canada Inc., *Toronto*
Prentice-Hall Hispanoamericana, S.A., *Mexico*
Prentice-Hall of India Private Limited, *New Delhi*
Prentice-Hall of Japan, Inc., *Tokyo*
Prentice-Hall of Southeast Asia Pte. Ltd., *Singapore*
Editora Prentice-Hall do Brasil, Ltda., *Rio de Janeiro*

To my parents, for their patience and devotion,
to my daughters, Sarah and Rachel, for their laughter,
to my son, Joseph, who arrived after the first printing,
and to my wife, Debby, for her love and understanding.

CONTENTS

PREFACE xi

CHAPTER 1 GENERAL OVERVIEW OF THE SYSTEM 1

1.1 HISTORY 1

1.2 SYSTEM STRUCTURE 4

1.3 USER PERSPECTIVE 6

1.4 OPERATING SYSTEM SERVICES 14

1.5 ASSUMPTIONS ABOUT HARDWARE 15

1.6 SUMMARY 18

CHAPTER 2 INTRODUCTION TO THE KERNEL 19
2.1 ARCHITECTURE OF THE UNIX OPERATING SYSTEM 19
2.2 INTRODUCTION TO SYSTEM CONCEPTS 22
2.3 KERNEL DATA STRUCTURES 34
2.4 SYSTEM ADMINISTRATION 34
2.5 SUMMARY AND PREVIEW 36
2.6 EXERCISES 37

CHAPTER 3 THE BUFFER CACHE 38
3.1 BUFFER HEADERS 39
3.2 STRUCTURE OF THE BUFFER POOL 40
3.3 SCENARIOS FOR RETRIEVAL OF A BUFFER 42
3.4 READING AND WRITING DISK BLOCKS 53
3.5 ADVANTAGES AND DISADVANTAGES OF THE BUFFER CACHE 56
3.6 SUMMARY 57
3.7 EXERCISES 58

CHAPTER 4 INTERNAL REPRESENTATION OF FILES 60
4.1 INODES 61
4.2 STRUCTURE OF A REGULAR FILE 67
4.3 DIRECTORIES 73
4.4 CONVERSION OF A PATH NAME TO AN INODE 74
4.5 SUPER BLOCK 76
4.6 INODE ASSIGNMENT TO A NEW FILE 77
4.7 ALLOCATION OF DISK BLOCKS 84
4.8 OTHER FILE TYPES 88
4.9 SUMMARY 88
4.10 EXERCISES 89

CHAPTER 5 SYSTEM CALLS FOR THE FILE SYSTEM 91
5.1 OPEN 92
5.2 READ 96
5.3 WRITE 101
5.4 FILE AND RECORD LOCKING 103
5.5 ADJUSTING THE POSITION OF FILE I/O — LSEEK . . 103
5.6 CLOSE 103
5.7 FILE CREATION 105
5.8 CREATION OF SPECIAL FILES 107
5.9 CHANGE DIRECTORY AND CHANGE ROOT 109
5.10 CHANGE OWNER AND CHANGE MODE 110
5.11 STAT AND FSTAT 110
5.12 PIPES 111
5.13 DUP 117
5.14 MOUNTING AND UNMOUNTING FILE SYSTEMS . . 119
5.15 LINK 128
5.16 UNLINK 132
5.17 FILE SYSTEM ABSTRACTIONS 138
5.18 FILE SYSTEM MAINTENANCE 139
5.19 SUMMARY 140
5.20 EXERCISES 140

CHAPTER 6 THE STRUCTURE OF PROCESSES 146
6.1 PROCESS STATES AND TRANSITIONS 147
6.2 LAYOUT OF SYSTEM MEMORY 151
6.3 THE CONTEXT OF A PROCESS 159
6.4 SAVING THE CONTEXT OF A PROCESS 162
6.5 MANIPULATION OF THE PROCESS ADDRESS SPACE 171
6.6 SLEEP 182

6.7 SUMMARY 188
6.8 EXERCISES 189

CHAPTER 7 PROCESS CONTROL 191
7.1 PROCESS CREATION 192
7.2 SIGNALS 200
7.3 PROCESS TERMINATION 212
7.4 AWAITING PROCESS TERMINATION 213
7.5 INVOKING OTHER PROGRAMS 217
7.6 THE USER ID OF A PROCESS 227
7.7 CHANGING THE SIZE OF A PROCESS 229
7.8 THE SHELL 232
7.9 SYSTEM BOOT AND THE INIT PROCESS 235
7.10 SUMMARY 238
7.11 EXERCISES 239

CHAPTER 8 PROCESS SCHEDULING AND TIME 247
8.1 PROCESS SCHEDULING 248
8.2 SYSTEM CALLS FOR TIME 258
8.3 CLOCK 260
8.4 SUMMARY 268
8.5 EXERCISES 268

CHAPTER 9 MEMORY MANAGEMENT POLICIES 271
9.1 SWAPPING 272
9.2 DEMAND PAGING 285
9.3 A HYBRID SYSTEM WITH SWAPPING AND DEMAND PAGING 307
9.4 SUMMARY 307
9.5 EXERCISES 308

CHAPTER 10 THE I/O SUBSYSTEM 312
10.1 DRIVER INTERFACES 313
10.2 DISK DRIVERS 325
10.3 TERMINAL DRIVERS 329
10.4 STREAMS 344
10.5 SUMMARY 351
10.6 EXERCISES 352

CHAPTER 11 INTERPROCESS COMMUNICATION 355
11.1 PROCESS TRACING 356
11.2 SYSTEM V IPC 359
11.3 NETWORK COMMUNICATIONS 382
11.4 SOCKETS 383
11.5 SUMMARY 388
11.6 EXERCISES 389

CHAPTER 12 MULTIPROCESSOR SYSTEMS 391
12.1 PROBLEM OF MULTIPROCESSOR SYSTEMS 392
12.2 SOLUTION WITH MASTER AND SLAVE PROCESSORS 393
12.3 SOLUTION WITH SEMAPHORES 395
12.4 THE TUNIS SYSTEM 410
12.5 PERFORMANCE LIMITATIONS 410
12.6 EXERCISES 410

CHAPTER 13 DISTRIBUTED UNIX SYSTEMS 412
13.1 SATELLITE PROCESSORS 414
13.2 THE NEWCASTLE CONNECTION 422
13.3 TRANSPARENT DISTRIBUTED FILE SYSTEMS . . . 426
13.4 A TRANSPARENT DISTRIBUTED MODEL WITHOUT STUB PROCESSES 429

13.5 SUMMARY 430
13.6 EXERCISES 431

APPENDIX – SYSTEM CALLS 434
BIBLIOGRAPHY 454
INDEX 458

PREFACE

The UNIX system was first described in a 1974 paper in the Communications of the ACM [Thompson 74] by Ken Thompson and Dennis Ritchie. Since that time, it has become increasingly widespread and popular throughout the computer industry where more and more vendors are offering support for it on their machines. It is especially popular in universities where it is frequently used for operating systems research and case studies.

Many books and papers have described parts of the system, among them, two special issues of the Bell System Technical Journal in 1978 [BSTJ 78] and 1984 [BLTJ 84]. Many books describe the user level interface, particularly how to use electronic mail, how to prepare documents, or how to use the command interpreter called the shell; some books such as *The UNIX Programming Environment* [Kernighan 84] and *Advanced UNIX Programming* [Rochkind 85] describe the programming interface. This book describes the internal algorithms and structures that form the basis of the operating system (called the kernel) and their relationship to the programmer interface. It is thus applicable to several environments. First, it can be used as a textbook for an operating systems course at either the advanced undergraduate or first-year graduate level. It is most beneficial to reference the system source code when using the book, but the book can be read independently, too. Second, system programmers can use the book as a reference to gain better understanding of how the kernel works and to compare algorithms used in the UNIX system to algorithms used in other operating systems.

Finally, programmers on UNIX systems can gain a deeper understanding of how their programs interact with the system and thereby code more-efficient, sophisticated programs.

The material and organization for the book grew out of a course that I prepared and taught at AT&T Bell Laboratories during 1983 and 1984. While the course centered on reading the source code for the system, I found that understanding the code was easier once the concepts of the algorithms had been mastered. I have attempted to keep the descriptions of algorithms in this book as simple as possible, reflecting in a small way the simplicity and elegance of the system it describes. Thus, the book is not a line-by-line rendition of the system written in English; it is a description of the general flow of the various algorithms, and most important, a description of how they interact with each other. Algorithms are presented in a C-like pseudo-code to aid the reader in understanding the natural language description, and their names correspond to the procedure names in the kernel. Figures depict the relationship between various data structures as the system manipulates them. In later chapters, small C programs illustrate many system concepts as they manifest themselves to users. In the interests of space and clarity, these examples do not usually check for error conditions, something that should always be done when writing programs. I have run them on System V; except for programs that exercise features specific to System V, they should run on other versions of the system, too.

Many exercises originally prepared for the course have been included at the end of each chapter, and they are a key part of the book. Some exercises are straightforward, designed to illustrate concepts brought out in the text. Others are more difficult, designed to help the reader understand the system at a deeper level. Finally, some are exploratory in nature, designed for investigation as a research problem. Difficult exercises are marked with asterisks.

The system description is based on UNIX System V Release 2 supported by AT&T, with some new features from Release 3. This is the system with which I am most familiar, but I have tried to portray interesting contributions of other variations to the operating system, particularly those of Berkeley Software Distribution (BSD). I have avoided issues that assume particular hardware characteristics, trying to cover the kernel-hardware interface in general terms and ignoring particular machine idiosyncrasies. Where machine-specific issues are important to understand implementation of the kernel, however, I delve into the relevant detail. At the very least, examination of these topics will highlight the parts of the operating system that are the most machine dependent.

The reader must have programming experience with a high-level language and, preferably, with an assembly language as a prerequisite for understanding this book. It is recommended that the reader have experience working with the UNIX system and that the reader knows the C language [Kernighan 78]. However, I have attempted to write this book in such a way that the reader should still be able to absorb the material without such background. The appendix contains a simplified description of the system calls, sufficient to understand the presentation

in the book, but not a complete reference manual.

The book is organized as follows. Chapter 1 is the introduction, giving a brief, general description of system features as perceived by the user and describing the system structure. Chapter 2 describes the general outline of the kernel architecture and presents some basic concepts. The remainder of the book follows the outline presented by the system architecture, describing the various components in a building block fashion. It can be divided into three parts: the file system, process control, and advanced topics. The file system is presented first, because its concepts are easier than those for process control. Thus, Chapter 3 describes the system buffer cache mechanism that is the foundation of the file system. Chapter 4 describes the data structures and algorithms used internally by the file system. These algorithms use the algorithms explained in Chapter 3 and take care of the internal bookkeeping needed for managing user files. Chapter 5 describes the system calls that provide the user interface to the file system; they use the algorithms in Chapter 4 to access user files.

Chapter 6 turns to the control of processes. It defines the context of a process and investigates the internal kernel primitives that manipulate the process context. In particular, it considers the system call interface, interrupt handling, and the context switch. Chapter 7 presents the system calls that control the process context. Chapter 8 deals with process scheduling, and Chapter 9 covers memory management, including swapping and paging systems.

Chapter 10 outlines general driver interfaces, with specific discussion of disk drivers and terminal drivers. Although devices are logically part of the file system, their discussion is deferred until here because of issues in process control that arise in terminal drivers. This chapter also acts as a bridge to the more advanced topics presented in the rest of the book. Chapter 11 covers interprocess communication and networking, including System V messages, shared memory and semaphores, and BSD sockets. Chapter 12 explains tightly coupled multiprocessor UNIX systems, and Chapter 13 investigates loosely coupled distributed systems.

The material in the first nine chapters could be covered in a one-semester course on operating systems, and the material in the remaining chapters could be covered in advanced seminars with various projects being done in parallel.

A few caveats must be made at this time. No attempt has been made to describe system performance in absolute terms, nor is there any attempt to suggest configuration parameters for a system installation. Such data is likely to vary according to machine type, hardware configuration, system version and implementation, and application mix. Similarly, I have made a conscious effort to avoid predicting future development of UNIX operating system features. Discussion of advanced topics does not imply a commitment by AT&T to provide particular features, nor should it even imply that particular areas are under investigation.

It is my pleasure to acknowledge the assistance of many friends and colleagues who encouraged me while I wrote this book and provided constructive criticism of the manuscript. My deepest appreciation goes to Ian Johnstone, who suggested

that I write this book, gave me early encouragement, and reviewed the earliest draft of the first chapters. Ian taught me many tricks of the trade, and I will always be indebted to him. Doris Ryan also had a hand in encouraging me from the very beginning, and I will always appreciate her kindness and thoughtfulness. Dennis Ritchie freely answered numerous questions on the historical and technical background of the system. Many people gave freely of their time and energy to review drafts of the manuscript, and this book owes a lot to their detailed comments. They are Debby Bach, Doug Bayer, Lenny Brandwein, Steve Buroff, Tom Butler, Ron Gomes, Mesut Gunduc, Laura Israel, Dean Jagels, Keith Kelleman, Brian Kernighan, Bob Martin, Bob Mitze, Dave Nowitz, Michael Poppers, Marilyn Safran, Curt Schimmel, Zvi Spitz, Tom Vaden, Bill Weber, Larry Wehr, and Bob Zarrow. Mary Fruhstuck provided help in preparing the manuscript for typesetting. I would like to thank my management for their continued support throughout this project and my colleagues, for providing such a stimulating atmosphere and wonderful work environment at AT&T Bell Laboratories. John Wait and the staff at Prentice-Hall provided much valuable assitance and advice to get the book into its final form. Last, but not least, my wife, Debby, gave me lots of emotional support, without which I could never have succeeded.

1

GENERAL OVERVIEW OF THE SYSTEM

The UNIX system has become quite popular since its inception in 1969, running on machines of varying processing power from microprocessors to mainframes and providing a common execution environment across them. The system is divided into two parts. The first part consists of programs and services that have made the UNIX system environment so popular; it is the part readily apparent to users, including such programs as the shell, mail, text processing packages, and source code control systems. The second part consists of the operating system that supports these programs and services. This book gives a detailed description of the operating system. It concentrates on a description of UNIX System V produced by AT&T but considers interesting features provided by other versions too. It examines the major data structures and algorithms used in the operating system that ultimately provide users with the standard user interface.

This chapter provides an introduction to the UNIX system. It reviews its history and outlines the overall system structure. The next chapter gives a more detailed introduction to the operating system.

1.1 HISTORY

In 1965, Bell Telephone Laboratories joined an effort with the General Electric Company and Project MAC of the Massachusetts Institute of Technology to

develop a new operating system called Multics [Organick 72]. The goals of the Multics system were to provide simultaneous computer access to a large community of users, to supply ample computation power and data storage, and to allow users to share their data easily, if desired. Many people who later took part in the early development of the UNIX system participated in the Multics work at Bell Laboratories. Although a primitive version of the Multics system was running on a GE 645 computer by 1969, it did not provide the general service computing for which it was intended, nor was it clear when its development goals would be met. Consequently, Bell Laboratories ended its participation in the project.

With the end of their work on the Multics project, members of the Computing Science Research Center at Bell Laboratories were left without a "convenient interactive computing service" [Ritchie 84a]. In an attempt to improve their programming environment, Ken Thompson, Dennis Ritchie, and others sketched a paper design of a file system that later evolved into an early version of the UNIX file system. Thompson wrote programs that simulated the behavior of the proposed file system and of programs in a demand-paging environment, and he even encoded a simple kernel for the GE 645 computer. At the same time, he wrote a game program, "Space Travel," in Fortran for a GECOS system (the Honeywell 635), but the program was unsatisfactory because it was difficult to control the "space ship" and the program was expensive to run. Thompson later found a little-used PDP-7 computer that provided good graphic display and cheap executing power. Programming "Space Travel" for the PDP-7 enabled Thompson to learn about the machine, but its environment for program development required cross-assembly of the program on the GECOS machine and carrying paper tape for input to the PDP-7. To create a better development environment, Thompson and Ritchie implemented their system design on the PDP-7, including an early version of the UNIX file system, the process subsystem, and a small set of utility programs. Eventually, the new system no longer needed the GECOS system as a development environment but could support itself. The new system was given the name UNIX, a pun on the name Multics coined by another member of the Computing Science Research Center, Brian Kernighan.

Although this early version of the UNIX system held much promise, it could not realize its potential until it was used in a real project. Thus, while providing a text processing system for the patent department at Bell Laboratories, the UNIX system was moved to a PDP-11 in 1971. The system was characterized by its small size: 16K bytes for the system, 8K bytes for user programs, a disk of 512K bytes, and a limit of 64K bytes per file. After its early success, Thompson set out to implement a Fortran compiler for the new system, but instead came up with the language B, influenced by BCPL [Richards 69]. B was an interpretive language with the performance drawbacks implied by such languages, so Ritchie developed it into one he called C, allowing generation of machine code, declaration of data types, and definition of data structures. In 1973, the operating system was rewritten in C, an unheard of step at the time, but one that was to have tremendous impact on its acceptance among outside users. The number of installations at Bell

Laboratories grew to about 25, and a UNIX Systems Group was formed to provide internal support.

At this time, AT&T could not market computer products because of a 1956 Consent Decree it had signed with the Federal government, but it provided the UNIX system to universities who requested it for educational purposes. AT&T neither advertised, marketed, nor supported the system, in adherence to the terms of the Consent Decree. Nevertheless, the system's popularity steadily increased. In 1974, Thompson and Ritchie published a paper describing the UNIX system in the Communications of the ACM [Thompson 74], giving further impetus to its acceptance. By 1977, the number of UNIX system sites had grown to about 500, of which 125 were in universities. UNIX systems became popular in the operating telephone companies, providing a good environment for program development, network transaction operations services, and real-time services (via MERT [Lycklama 78a]). Licenses of UNIX systems were provided to commercial institutions as well as universities. In 1977, Interactive Systems Corporation became the first Value Added Reseller (VAR)[1] of a UNIX system, enhancing it for use in office automation environments. 1977 also marked the year that the UNIX system was first "ported" to a non-PDP machine (that is, made to run on another machine with few or no changes), the Interdata 8/32.

With the growing popularity of microprocessors, other companies ported the UNIX system to new machines, but its simplicity and clarity tempted many developers to enhance it in their own way, resulting in several variants of the basic system. In the period from 1977 to 1982, Bell Laboratories combined several AT&T variants into a single system, known commercially as UNIX System III. Bell Laboratories later added several features to UNIX System III, calling the new product UNIX System V,[2] and AT&T announced official support for System V in January 1983. However, people at the University of California at Berkeley had developed a variant to the UNIX system, the most recent version of which is called 4.3 BSD for VAX machines, providing some new, interesting features. This book will concentrate on the description of UNIX System V and will occasionally talk about features provided in the BSD system.

By the beginning of 1984, there were about 100,000 UNIX system installations in the world, running on machines with a wide range of computing power from microprocessors to mainframes and on machines across different manufacturers' product lines. No other operating system can make that claim. Several reasons have been suggested for the popularity and success of the UNIX system.

---

1. Value Added Resellers add specific applications to a computer system to satisfy a particular market. They market the applications rather than the operating system upon which they run.

2. What happened to System IV? An internal version of the system evolved into System V.

- The system is written in a high-level language, making it easy to read, understand, change, and move to other machines. Ritchie estimates that the first system in C was 20 to 40 percent larger and slower because it was not written in assembly language, but the advantages of using a higher-level language far outweigh the disadvantages (see page 1965 of [Ritchie 78b]).
- It has a simple user interface that has the power to provide the services that users want.
- It provides primitives that permit complex programs to be built from simpler programs.
- It uses a hierarchical file system that allows easy maintenance and efficient implementation.
- It uses a consistent format for files, the byte stream, making application programs easier to write.
- It provides a simple, consistent interface to peripheral devices.
- It is a multi-user, multiprocess system; each user can execute several processes simultaneously.
- It hides the machine architecture from the user, making it easier to write programs that run on different hardware implementations.

The philosophy of simplicity and consistency underscores the UNIX system and accounts for many of the reasons cited above.

Although the operating system and many of the command programs are written in C, UNIX systems support other languages, including Fortran, Basic, Pascal, Ada, Cobol, Lisp, and Prolog. The UNIX system can support any language that has a compiler or interpreter and a system interface that maps user requests for operating system services to the standard set of requests used on UNIX systems.

1.2 SYSTEM STRUCTURE

Figure 1.1 depicts the high-level architecture of the UNIX system. The hardware at the center of the diagram provides the operating system with basic services that will be described in Section 1.5. The operating system interacts directly[3] with the hardware, providing common services to programs and insulating them from hardware idiosyncrasies. Viewing the system as a set of layers, the operating system is commonly called the *system kernel*, or just the kernel, emphasizing its

3. In some implementations of the UNIX system, the operating system interacts with a native operating system that, in turn, interacts with the underlying hardware and provides necessary services to the system. Such configurations allow installations to run other operating systems and their applications in parallel to the UNIX system. The classic example of such a configuration is the MERT system [Lycklama 78a]. More recent configurations include implementations for IBM System/370 computers [Felton 84] and for UNIVAC 1100 Series computers [Bodenstab 84].

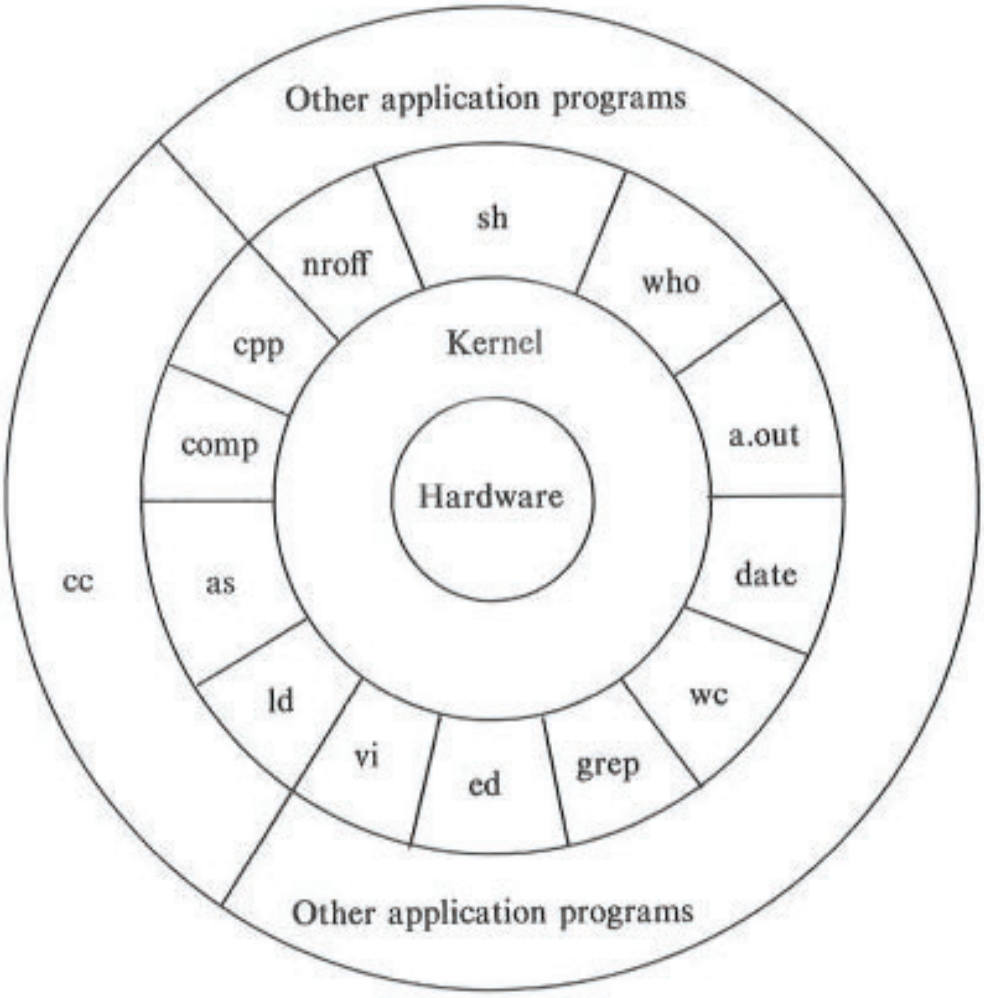


Figure 1.1. Architecture of UNIX Systems

isolation from user programs. Because programs are independent of the underlying hardware, it is easy to move them between UNIX systems running on different hardware if the programs do not make assumptions about the underlying hardware. For instance, programs that assume the size of a machine word are more difficult to move to other machines than programs that do not make this assumption.

Programs such as the shell and editors (*ed* and *vi*) shown in the outer layers interact with the kernel by invoking a well defined set of *system calls*. The system calls instruct the kernel to do various operations for the calling program and exchange data between the kernel and the program. Several programs shown in the figure are in standard system configurations and are known as *commands*, but private user programs may also exist in this layer as indicated by the program whose name is *a.out*, the standard name for executable files produced by the C compiler. Other application programs can build on top of lower-level programs, hence the existence of the outermost layer in the figure. For example, the standard C compiler, *cc*, is in the outermost layer of the figure: it invokes a C preprocessor,

two-pass compiler, assembler, and loader (link-editor), all separate lower-level programs. Although the figure depicts a two-level hierarchy of application programs, users can extend the hierarchy to whatever levels are appropriate. Indeed, the style of programming favored by the UNIX system encourages the combination of existing programs to accomplish a task.

Many application subsystems and programs that provide a high-level view of the system such as the shell, editors, SCCS (Source Code Control System), and document preparation packages, have gradually become synonymous with the name "UNIX system." However, they all use lower-level services ultimately provided by the kernel, and they avail themselves of these services via the set of system calls. There are about 64 system calls in System V, of which fewer than 32 are used frequently. They have simple options that make them easy to use but provide the user with a lot of power. The set of system calls and the internal algorithms that implement them form the body of the kernel, and the study of the UNIX operating system presented in this book reduces to a detailed study and analysis of the system calls and their interaction with one another. In short, the kernel provides the services upon which all application programs in the UNIX system rely, and it defines those services. This book will frequently use the terms "UNIX system," "kernel," or "system," but the intent is to refer to the kernel of the UNIX operating system and should be clear in context.

1.3 USER PERSPECTIVE

This section briefly reviews high-level features of the UNIX system such as the file system, the processing environment, and building block primitives (for example, *pipes*). Later chapters will explore kernel support of these features in detail.

1.3.1 The File System

The UNIX file system is characterized by

- a hierarchical structure,
- consistent treatment of file data,
- the ability to create and delete files,
- dynamic growth of files,
- the protection of file data,
- the treatment of peripheral devices (such as terminals and tape units) as files.

The file system is organized as a tree with a single root node called *root* (written "/"); every non-leaf node of the file system structure is a *directory* of files, and files at the leaf nodes of the tree are either directories, *regular files*, or *special* device files. The name of a file is given by a *path name* that describes how to locate the file in the file system hierarchy. A path name is a sequence of component names separated by slash characters; a component is a sequence of characters that

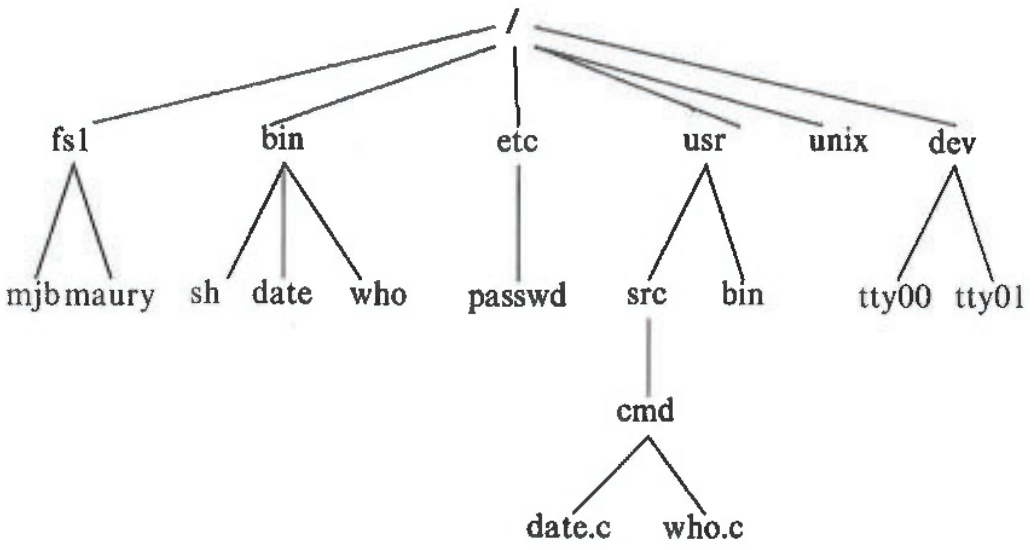


Figure 1.2. Sample File System Tree

designates a file name that is uniquely contained in the previous (directory) component. A full path name starts with a slash character and specifies a file that can be found by starting at the file system root and traversing the file tree, following the branches that lead to successive component names of the path name. Thus, the path names "/etc/passwd", "/bin/who", and "/usr/src/cmd/who.c" designate files in the tree shown in Figure 1.2, but "/bin/passwd" and "/usr/src/date.c" do not. A path name does not have to start from root but can be designated relative to the *current directory* of an executing process, by omitting the initial slash in the path name. Thus, starting from directory "/dev", the path name "tty01" designates the file whose full path name is "/dev/tty01".

Programs in the UNIX system have no knowledge of the internal format in which the kernel stores file data, treating the data as an unformatted stream of bytes. Programs may interpret the byte stream as they wish, but the interpretation has no bearing on how the operating system stores the data. Thus, the syntax of accessing the data in a file is defined by the system and is identical for all programs, but the semantics of the data are imposed by the program. For example, the text formatting program *troff* expects to find "new-line" characters at the end of each line of text, and the system accounting program *acctcom* expects to find fixed length records. Both programs use the same system services to access the data in the file as a byte stream, and internally, they parse the stream into a suitable format. If either program discovers that the format is incorrect, it is responsible for taking the appropriate action.

Directories are like regular files in this respect; the system treats the data in a directory as a byte stream, but the data contains the names of the files in the directory in a predictable format so that the operating system and programs such as

ls (list the names and attributes of files) can discover the files in a directory.

Permission to access a file is controlled by *access permissions* associated with the file. Access permissions can be set independently to control read, write, and execute permission for three classes of users: the file owner, a file group, and everyone else. Users may create files if directory access permissions allow it. The newly created files are leaf nodes of the file system directory structure.

To the user, the UNIX system treats devices as if they were files. Devices, designated by special device files, occupy node positions in the file system directory structure. Programs access devices with the same syntax they use when accessing regular files; the semantics of reading and writing devices are to a large degree the same as reading and writing regular files. Devices are protected in the same way that regular files are protected: by proper setting of their (file) access permissions. Because device names look like the names of regular files and because the same operations work for devices and regular files, most programs do not have to know internally the types of files they manipulate.

For example, consider the C program in Figure 1.3, which makes a new copy of an existing file. Suppose the name of the executable version of the program is *copy*. A user at a terminal invokes the program by typing

```
copy oldfile newfile
```

where *oldfile* is the name of the existing file and *newfile* is the name of the new file. The system invokes *main*, supplying *argc* as the number of parameters in the list *argv*, and initializing each member of the array *argv* to point to a user-supplied parameter. In the example above, *argc* is 3, *argv[0]* points to the character string *copy* (the program name is conventionally the 0th parameter), *argv[1]* points to the character string *oldfile*, and *argv[2]* points to the character string *newfile*. The program then checks that it has been invoked with the proper number of parameters. If so, it invokes the *open* system call "read-only" for the file *oldfile*, and if the system call succeeds, invokes the *creat* system call to create *newfile*. The permission modes on the newly created file will be 0666 (octal), allowing all users access to the file for reading and writing. All system calls return −1 on failure; if the *open* or *creat* calls fail, the program prints a message and calls the *exit* system call with return status 1, terminating its execution and indicating that something went wrong.

The *open* and *creat* system calls return an integer called a *file descriptor*, which the program uses for subsequent references to the files. The program then calls the subroutine *copy*, which goes into a loop, invoking the *read* system call to read a buffer's worth of characters from the existing file, and invoking the *write* system call to write the data to the new file. The *read* system call returns the number of bytes read, returning 0 when it reaches the end of file. The program finishes the loop when it encounters the end of file, or when there is some error on the *read* system call (it does not check for *write* errors). Then it returns from *copy* and *exit*s with return status 0, indicating that the program completed successfully.

```
#include <fcntl.h>
char buffer[2048];
int version = 1;        /* Chapter 2 explains this */

main(argc, argv)
        int argc;
        char *argv[];
{
        int fdold, fdnew;

        if (argc != 3)
        {
                printf("need 2 arguments for copy program\n");
                exit(1);
        }
        fdold = open(argv[1], O_RDONLY);    /* open source file read only */
        if (fdold == -1)
        {
                printf("cannot open file %s\n", argv[1]);
                exit(1);
        }
        fdnew = creat(argv[2], 0666);    /* create target file rw for all */
        if (fdnew == -1)
        {
                printf("cannot create file %s\n", argv[2]);
                exit(1);
        }
        copy(fdold, fdnew);
        exit(0);
}

copy(old, new)
        int old, new;
{
        int count;

        while ((count = read(old, buffer, sizeof(buffer))) > 0)
                write(new, buffer, count);
}
```

Figure 1.3. Program to Copy a File

The program copies any files supplied to it as arguments, provided it has permission to *open* the existing file and permission to create the new file. The file can be a file of printable characters, such as the source code for the program, or it can contain unprintable characters, even the program itself. Thus, the two

invocations

```
copy copy.c newcopy.c
copy copy newcopy
```

both work. The old file can also be a directory. For instance,

```
copy . dircontents
```

copies the contents of the current directory, denoted by the name ".", to a regular file, "dircontents"; the data in the new file is identical, byte for byte, to the contents of the directory, but the file is a regular file. (The system call *mknod* creates a new directory.) Finally, either file can be a device special file. For example,

```
copy /dev/tty terminalread
```

reads the characters typed at the terminal (the special file */dev/tty* is the user's terminal) and copies them to the file *terminalread*, terminating only when the user types the character control-d. Similarly,

```
copy /dev/tty /dev/tty
```

reads characters typed at the terminal and copies them back.

1.3.2 Processing Environment

A *program* is an executable file, and a *process* is an instance of the program in execution. Many processes can execute simultaneously on UNIX systems (this feature is sometimes called multiprogramming or multitasking) with no logical limit to their number, and many instances of a program (such as *copy*) can exist simultaneously in the system. Various system calls allow processes to create new processes, terminate processes, synchronize stages of process execution, and control reaction to various events. Subject to their use of system calls, processes execute independently of each other.

For example, a process executing the program in Figure 1.4 executes the *fork* system call to create a new process. The new process, called the *child* process, gets a 0 return value from *fork* and invokes *execl* to execute the program *copy* (the program in Figure 1.3). The *execl* call overlays the address space of the child process with the file "copy", assumed to be in the current directory, and runs the program with the user-supplied parameters. If the *execl* call succeeds, it never returns because the process executes in a new address space, as will be seen in Chapter 7. Meanwhile, the process that had invoked *fork* (the parent) receives a non-0 return from the call, calls *wait*, suspending its execution until *copy* finishes, prints the message "copy done," and *exit*s (every program *exit*s at the end of its *main* function, as arranged by standard C program libraries that are linked during the compilation process). For example, if the name of the executable program is *run*, and a user invokes the program by

```
main(argc, argv)
        int argc;
        char *argv[];
{
        /* assume 2 args:  source file and target file */
        if (fork() == 0)
                execl("copy", "copy", argv[1], argv[2], 0);
        wait((int *) 0);
        printf("copy done\n");
}
```

Figure 1.4. Program that Creates a New Process to Copy Files

run oldfile newfile

the process copies "oldfile" to "newfile" and prints out the message. Although this program adds little to the "copy" program, it exhibits four major system calls used for process control: *fork*, *exec*, *wait*, and, discreetly, *exit*.

Generally, the system calls allow users to write programs that do sophisticated operations, and as a result, the kernel of the UNIX system does not contain many functions that are part of the "kernel" in other systems. Such functions, including compilers and editors, are user-level programs in the UNIX system. The prime example of such a program is the *shell*, the command interpreter program that users typically execute after logging into the system. The shell interprets the first word of a *command line* as a *command* name: for many commands, the shell *fork*s and the child process *exec*s the command associated with the name, treating the remaining words on the command line as parameters to the command.

The shell allows three types of commands. First, a command can be an executable file that contains object code produced by compilation of source code (a C program for example). Second, a command can be an executable file that contains a sequence of shell command lines. Finally, a command can be an internal shell command (instead of an executable file). The internal commands make the shell a programming language in addition to a command interpreter and include commands for looping (*for-in-do-done* and *while-do-done*), commands for conditional execution (*if-then-else-fi*), a "case" statement command, a command to change the current directory of a process (*cd*), and several others. The shell syntax allows for pattern matching and parameter processing. Users execute commands without having to know their types.

The shell searches for commands in a given sequence of directories, changeable by user request per invocation of the shell. The shell usually executes a command synchronously, waiting for the command to terminate before reading the next command line. However, it also allows asynchronous execution, where it reads the next command line and executes it without waiting for the prior command to terminate. Commands executed asynchronously are said to execute in the

background. For example, typing the command

```
who
```

causes the system to execute the program stored in the file */bin/who*,[4] which prints a list of people who are currently logged in to the system. While *who* executes, the shell waits for it to finish and then prompts the user for another command. By typing

```
who &
```

the system executes the program *who* in the background, and the shell is ready to accept another command immediately.

Every process executing in the UNIX system has an execution environment that includes a current directory. The current directory of a process is the start directory used for all path names that do not begin with the slash character. The user may execute the shell command *cd*, change directory, to move around the file system tree and change the current directory. The command line

```
cd /usr/src/uts
```

changes the shell's current directory to the directory "/usr/src/uts". The command line

```
cd ../..
```

changes the shell's current directory to the directory that is two nodes "closer" to the root node: the component ".." refers to the *parent directory* of the current directory.

Because the shell is a user program and not part of the kernel, it is easy to modify it and tailor it to a particular environment. For instance, users can use the C shell to provide a history mechanism and avoid retyping recently used commands, instead of the Bourne shell (named after its inventor, Steve Bourne), provided as part of the standard System V release. Or some users may be granted use only of a restricted shell, providing a scaled down version of the regular shell. The system can execute the various shells simultaneously. Users have the capability to execute many processes simultaneously, and processes can create other processes dynamically and synchronize their execution, if desired. These features provide users with a powerful execution environment. Although much of the power of the shell derives from its capabilities as a programming language and from its capabilities for pattern matching of arguments, this section concentrates on the process environment provided by the system via the shell. Other important shell

4. The directory "/bin" contains many useful commands and is usually included in the sequence of directories the shell searches.

features are beyond the scope of this book (see [Bourne 78] for a detailed description of the shell).

1.3.3 Building Block Primitives

As described earlier, the philosophy of the UNIX system is to provide operating system primitives that enable users to write small, modular programs that can be used as building blocks to build more complex programs. One such primitive visible to shell users is the capability to *redirect I/O*. Processes conventionally have access to three files: they read from their *standard input* file, write to their *standard output* file, and write error messages to their *standard error* file. Processes executing at a terminal typically use the terminal for these three files, but each may be "redirected" independently. For instance, the command line

```
ls
```

lists all files in the current directory on the standard output, but the command line

```
ls > output
```

redirects the standard output to the file called "output" in the current directory, using the *creat* system call mentioned above. Similarly, the command line

```
mail mjb < letter
```

opens the file "letter" for its standard intput and *mails* its contents to the user named "mjb." Processes can redirect input and output simultaneously, as in

```
nroff -mm < doc1 > doc1.out 2> errors
```

where the text formatter *nroff* reads the input file *doc1*, redirects its standard output to the file *doc1.out*, and redirects error messages to the file *errors* (the notation "2>" means to redirect the output for file descriptor 2, conventionally the standard error). The programs *ls*, *mail*, and *nroff* do not know what file their standard input, standard output, or standard error will be; the shell recognizes the symbols "<", ">", and "2>" and sets up the standard input, standard output, and standard error appropriately before executing the processes.

The second building block primitive is the *pipe*, a mechanism that allows a stream of data to be passed between reader and writer processes. Processes can redirect their standard output to a pipe to be read by other processes that have redirected their standard input to come from the pipe. The data that the first processes write into the pipe is the input for the second processes. The second processes could also redirect their output, and so on, depending on programming need. Again, the processes need not know what type of file their standard output is; they work regardless of whether their standard output is a regular file, a pipe, or a device. When using the smaller programs as building blocks for a larger, more complex program, the programmer uses the pipe primitive and redirection of I/O to integrate the piece parts. Indeed, the system tacitly encourages such programming

style so that new programs can work with existing programs.

For example, the program *grep* searches a set of files (parameters to grep) for a given pattern:

```
grep main a.c b.c c.c
```

searches the three files a.c, b.c, and c.c for lines containing the string "main" and prints the lines that it finds onto standard output. Sample output may be:

```
a.c: main(argc, argv)
c.c: /* here is the main loop in the program */
c.c: main()
```

The program *wc* with the option −l counts the number of lines in the standard input file. The command line

```
grep main a.c b.c c.c | wc −l
```

counts the number of lines in the files that contain the string "main"; the output from *grep* is "piped" directly into the *wc* command. For the previous sample output from *grep*, the output from the piped command is

```
3
```

The use of pipes frequently makes it unnecessary to create temporary files.

1.4 OPERATING SYSTEM SERVICES

Figure 1.1 depicts the kernel layer immediately below the layer of user application programs. The kernel performs various primitive operations on behalf of user processes to support the user interface described above. Among the services provided by the kernel are

- Controlling the execution of processes by allowing their creation, termination or suspension, and communication
- Scheduling processes fairly for execution on the CPU. Processes share the CPU in a *time-shared* manner: the CPU[5] executes a process, the kernel suspends it when its time quantum elapses, and the kernel schedules another process to execute. The kernel later reschedules the suspended process.
- Allocating main memory for an executing process. The kernel allows processes to share portions of their address space under certain conditions, but protects the private address space of a process from outside tampering. If the system runs low on free memory, the kernel frees memory by writing a process

5. Chapter 12 will consider multiprocessor systems; until then, assume a single processor model.

temporarily to secondary memory, called a *swap* device. If the kernel writes entire processes to a swap device, the implementation of the UNIX system is called a *swapping* system; if it writes pages of memory to a swap device, it is called a *paging* system.

- Allocating secondary memory for efficient storage and retrieval of user data. This service constitutes the file system. The kernel allocates secondary storage for user files, reclaims unused storage, structures the file system in a well understood manner, and protects user files from illegal access.
- Allowing processes controlled access to peripheral devices such as terminals, tape drives, disk drives, and network devices.

The kernel provides its services transparently. For example, it recognizes that a given file is a regular file or a device, but hides the distinction from user processes. Similarly, it formats data in a file for internal storage, but hides the internal format from user processes, returning an unformatted byte stream. Finally, it offers necessary services so that user-level processes can support the services they must provide, while omitting services that can be implemented at the user level. For example, the kernel supports the services that the shell needs to act as a command interpreter: It allows the shell to read terminal input, to spawn processes dynamically, to synchronize process execution, to create pipes, and to redirect I/O. Users can construct private versions of the shell to tailor their environments to their specifications without affecting other users. These programs use the same kernel services as the standard shell.

1.5 ASSUMPTIONS ABOUT HARDWARE

The execution of user processes on UNIX systems is divided into two levels: user and kernel. When a process executes a system call, the *execution mode* of the process changes from *user mode* to *kernel mode*: the operating system executes and attempts to service the user request, returning an error code if it fails. Even if the user makes no explicit requests for operating system services, the operating system still does bookkeeping operations that relate to the user process, handling interrupts, scheduling processes, managing memory, and so on. Many machine architectures (and their operating systems) support more levels than the two outlined here, but the two modes, user and kernel, are sufficient for UNIX systems.

The differences between the two modes are

- Processes in user mode can access their own instructions and data but not kernel instructions and data (or those of other processes). Processes in kernel mode, however, can access kernel and user addresses. For example, the virtual address space of a process may be divided between addresses that are accessible only in kernel mode and addresses that are accessible in either mode.
- Some machine instructions are privileged and result in an error when executed in user mode. For example, a machine may contain an instruction that manipulates the processor status register; processes executing in user mode

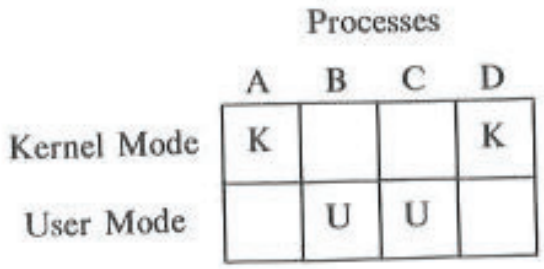


Figure 1.5. Multiple Processes and Modes of Execution

should not have this capability.

Put simply, the hardware views the world in terms of kernel mode and user mode and does not distinguish among the many users executing programs in those modes. The operating system keeps internal records to distinguish the many processes executing on the system. Figure 1.5 shows the distinction: the kernel distinguishes between processes A, B, C, and D on the horizontal axis, and the hardware distinguishes the mode of execution on the vertical axis.

Although the system executes in one of two modes, the kernel runs on behalf of a user process. The kernel is not a separate set of processes that run in parallel to user processes, but it is part of each user process. The ensuing text will frequently refer to "the kernel" allocating resources or "the kernel" doing various operations, but what is meant is that a process executing in kernel mode allocates the resources or does the various operations. For example, the shell reads user terminal input via a system call: The kernel, executing on behalf of the shell process, controls the operation of the terminal and returns the typed characters to the shell. The shell then executes in user mode, interprets the character stream typed by the user, and does the specified set of actions, which may require invocation of other system calls.

1.5.1 Interrupts and Exceptions

The UNIX system allows devices such as I/O peripherals or the system clock to interrupt the CPU asynchronously. On receipt of the interrupt, the kernel saves its current *context* (a frozen image of what the process was doing), determines the cause of the interrupt, and services the interrupt. After the kernel services the interrupt, it restores its interrupted context and proceeds as if nothing had happened. The hardware usually prioritizes devices according to the order that interrupts should be handled: When the kernel services an interrupt, it *blocks* out lower priority interrupts but services higher priority interrupts.

An exception condition refers to unexpected events caused by a process, such as addressing illegal memory, executing privileged instructions, dividing by zero, and so on. They are distinct from interrupts, which are caused by events that are

external to a process. Exceptions happen "in the middle" of the execution of an instruction, and the system attempts to restart the instruction after handling the exception; interrupts are considered to happen between the execution of two instructions, and the system continues with the next instruction after servicing the interrupt. The UNIX system uses one mechanism to handle interrupts and exception conditions.

1.5.2 Processor Execution Levels

The kernel must sometimes prevent the occurrence of interrupts during critical activity, which could result in corrupt data if interrupts were allowed. For instance, the kernel may not want to receive a disk interrupt while manipulating linked lists, because handling the interrupt could corrupt the pointers, as will be seen in the next chapter. Computers typically have a set of privileged instructions that set the processor execution level in the processor status word. Setting the processor execution level to certain values masks off interrupts from that level and lower levels, allowing only higher-level interrupts. Figure 1.6 shows a sample set of execution levels. If the kernel masks out disk interrupts, all interrupts except for clock interrupts and machine error interrupts are prevented. If it masks out software interrupts, all other interrupts may occur.

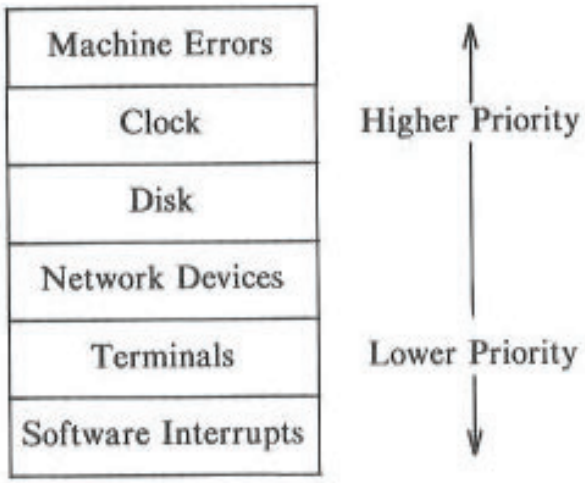


Figure 1.6. Typical Interrupt Levels

1.5.3 Memory Management

The kernel permanently resides in main memory as does the currently executing process (or parts of it, at least). When compiling a program, the compiler generates a set of addresses in the program that represent addresses of variables

and data structures or the addresses of instructions such as functions. The compiler generates the addresses for a *virtual machine* as if no other program will execute simultaneously on the physical machine.

When the program is to run on the machine, the kernel allocates space in main memory for it, but the *virtual addresses* generated by the compiler need not be identical to the physical addresses that they occupy in the machine. The kernel coordinates with the machine hardware to set up a *virtual to physical* address translation that maps the compiler-generated addresses to the physical machine addresses. The mapping depends on the capabilities of the machine hardware, and the parts of UNIX systems that deal with them are therefore machine dependent. For example, some machines have special hardware to support demand paging. Chapters 6 and 9 will discuss issues of memory management and how they relate to hardware in more detail.

1.6 SUMMARY

This chapter has described the overall structure of the UNIX system, the relationship between processes running in user mode versus kernel mode, and the assumptions the kernel makes about the hardware. Processes execute in user mode or kernel mode, where they avail themselves of system services using a well-defined set of system calls. The system design encourages programmers to write small programs that do only a few operations but do them well, and then to combine the programs using *pipe*s and I/O redirection to do more sophisticated processing.

The system calls allow processes to do operations that are otherwise forbidden to them. In addition to servicing system calls, the kernel does general bookkeeping for the user community, controlling process scheduling, managing the storage and protection of processes in main memory, fielding interrupts, managing files and devices, and taking care of system error conditions. The UNIX system kernel purposely omits many functions that are part of other operating systems, providing a small set of system calls that allow processes to do necessary functions at user level. The next chapter gives a more detailed introduction to the kernel, describing its architecture and some basic concepts used in its implementation.

2

INTRODUCTION TO THE KERNEL

The last chapter gave a high-level perspective of the UNIX system environment. This chapter focuses on the kernel, providing an overview of its architecture and outlining basic concepts and structures essential for understanding the rest of the book.

2.1 ARCHITECTURE OF THE UNIX OPERATING SYSTEM

It has been noted (see page 239 of [Christian 83]) that the UNIX system supports the illusions that the file system has "places" and that processes have "life." The two entities, files and processes, are the two central concepts in the UNIX system model. Figure 2.1 gives a block diagram of the kernel, showing various modules and their relationships to each other. In particular, it shows the file subsystem on the left and the process control subsystem on the right, the two major components of the kernel. The diagram serves as a useful logical view of the kernel, although in practice the kernel deviates from the model because some modules interact with the internal operations of others.

Figure 2.1 shows three levels: user, kernel, and hardware. The system call and library interface represent the border between user programs and the kernel depicted in Figure 1.1. System calls look like ordinary function calls in C programs, and libraries map these function calls to the primitives needed to enter

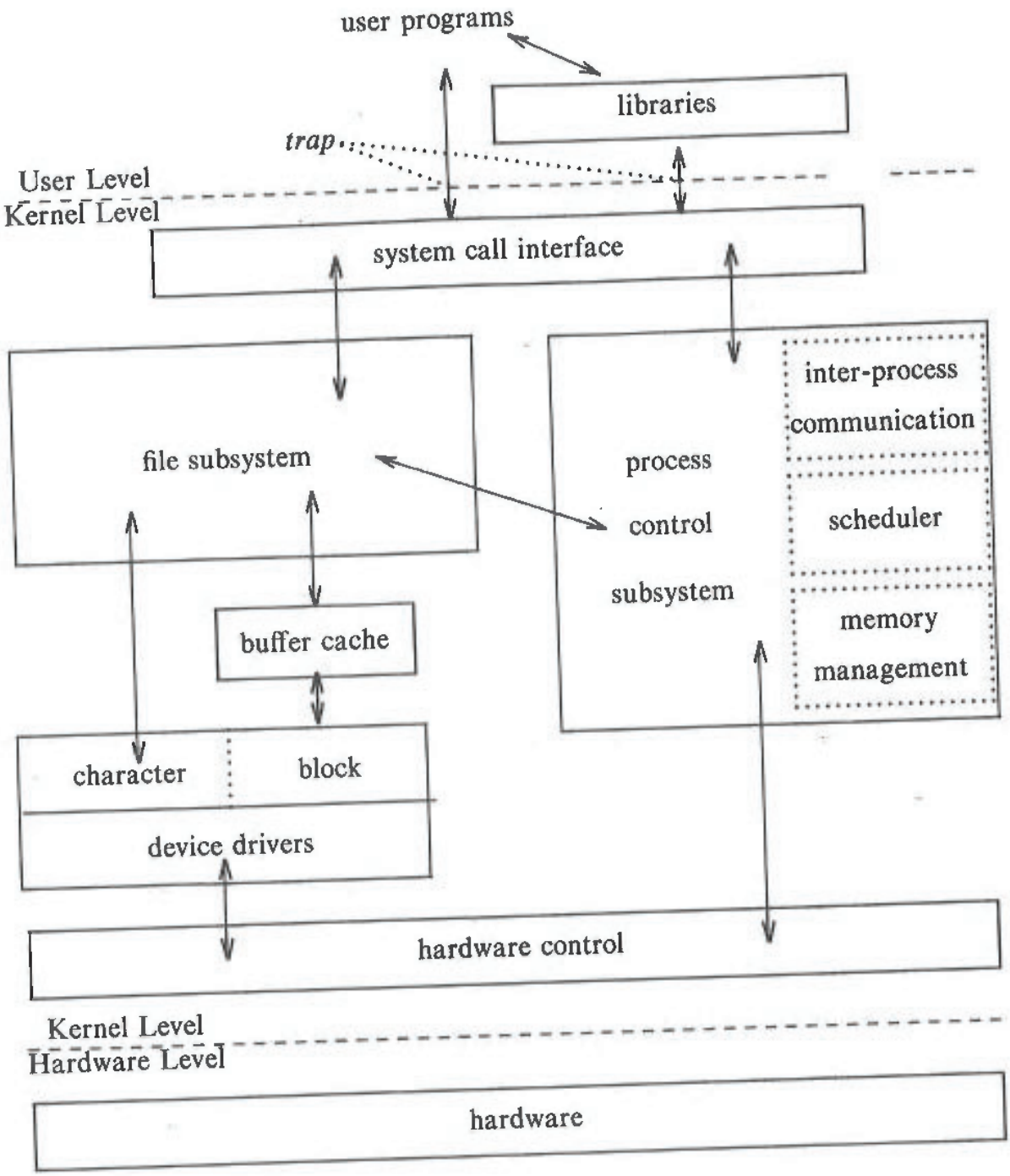


Figure 2.1. Block Diagram of the System Kernel

the operating system, as covered in more detail in Chapter 6. Assembly language programs may invoke system calls directly without a system call library, however. Programs frequently use other libraries such as the standard I/O library to provide a more sophisticated use of the system calls. The libraries are linked with the programs at compile time and are thus part of the user program for purposes of

this discussion. An example later on will illustrate these points.

The figure partitions the set of system calls into those that interact with the file subsystem and those that interact with the process control subsystem. The file subsystem manages files, allocating file space, administering free space, controlling access to files, and retrieving data for users. Processes interact with the file subsystem via a specific set of system calls, such as *open* (to open a file for reading or writing), *close*, *read*, *write*, *stat* (query the attributes of a file), *chown* (change the record of who owns the file), and *chmod* (change the access permissions of a file). These and others will be examined in Chapter 5.

The file subsystem accesses file data using a buffering mechanism that regulates data flow between the kernel and secondary storage devices. The buffering mechanism interacts with block I/O device drivers to initiate data transfer to and from the kernel. Device drivers are the kernel modules that control the operation of peripheral devices. Block I/O devices are random access storage devices; alternatively, their device drivers make them appear to be random access storage devices to the rest of the system. For example, a tape driver may allow the kernel to treat a tape unit as a random access storage device. The file subsystem also interacts directly with "raw" I/O device drivers without the intervention of a buffering mechanism. Raw devices, sometimes called character devices, include all devices that are not block devices.

The process control subsystem is responsible for process synchronization, interprocess communication, memory management, and process scheduling. The file subsystem and the process control subsystem interact when loading a file into memory for execution, as will be seen in Chapter 7: the process subsystem reads executable files into memory before executing them.

Some of the system calls for controlling processes are *fork* (create a new process), *exec* (overlay the image of a program onto the running process), *exit* (finish executing a process), *wait* (synchronize process execution with the *exit* of a previously *fork*ed process), *brk* (control the size of memory allocated to a process), and *signal* (control process response to extraordinary events). Chapter 7 will examine these system calls and others.

The memory management module controls the allocation of memory. If at any time the system does not have enough physical memory for all processes, the kernel moves them between main memory and secondary memory so that all processes get a fair chance to execute. Chapter 9 will describe two policies for managing memory: swapping and demand paging. The swapper process is sometimes called the scheduler, because it "schedules" the allocation of memory for processes and influences the operation of the CPU scheduler. However, this text will refer to it as the swapper to avoid confusion with the CPU scheduler.

The *scheduler* module allocates the CPU to processes. It schedules them to run in turn until they voluntarily relinquish the CPU while awaiting a resource or until the kernel preempts them when their recent run time exceeds a time quantum. The scheduler then chooses the highest priority eligible process to run; the original process will run again when it is the highest priority eligible process available.

There are several forms of interprocess communication, ranging from asynchronous signaling of events to synchronous transmission of messages between processes.

Finally, the hardware control is responsible for handling interrupts and for communicating with the machine. Devices such as disks or terminals may interrupt the CPU while a process is executing. If so, the kernel may resume execution of the interrupted process after servicing the interrupt: Interrupts are *not* serviced by special processes but by special functions in the kernel, called in the context of the currently running process.

2.2 INTRODUCTION TO SYSTEM CONCEPTS

This section gives an overview of some major kernel data structures and describes the function of modules shown in Figure 2.1 in more detail.

2.2.1 An Overview of the File Subsystem

The internal representation of a file is given by an *inode*, which contains a description of the disk layout of the file data and other information such as the file owner, access permissions, and access times. The term inode is a contraction of the term *index node* and is commonly used in literature on the UNIX system. Every file has one inode, but it may have several names, all of which map into the inode. Each name is called a *link*. When a process refers to a file by name, the kernel parses the file name one component at a time, checks that the process has permission to search the directories in the path, and eventually retrieves the inode for the file. For example, if a process calls

```
open("/fs2/mjb/rje/sourcefile", 1);
```

the kernel retrieves the inode for "/fs2/mjb/rje/sourcefile". When a process creates a new file, the kernel assigns it an unused inode. Inodes are stored in the file system, as will be seen shortly, but the kernel reads them into an in-core[1] inode table when manipulating files.

The kernel contains two other data structures, the *file table* and the *user file descriptor table*. The file table is a global kernel structure, but the user file descriptor table is allocated per process. When a process *opens* or *creats* a file, the kernel allocates an entry from each table, corresponding to the file's inode. Entries in the three structures — user file descriptor table, file table, and inode table — maintain the state of the file and the user's access to it. The file table keeps track of the byte offset in the file where the user's next *read* or *write* will start, and the

1. The term *core* refers to primary memory of a machine, not to hardware technology.

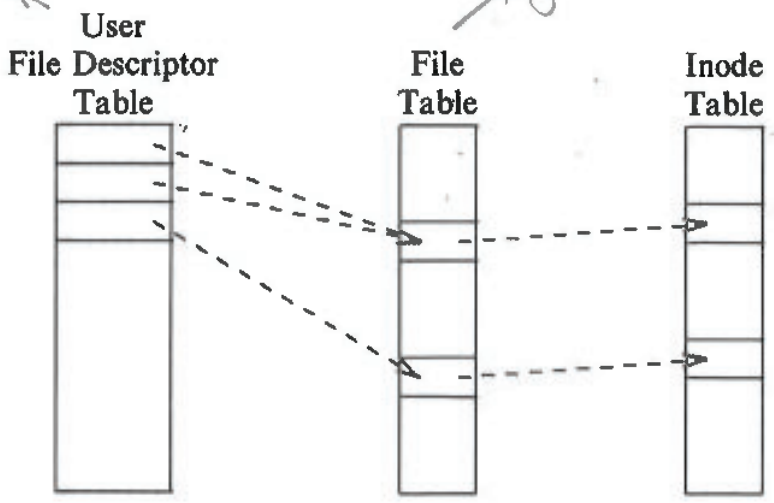


Figure 2.2. File Descriptors, File Table, and Inode Table

access rights allowed to the *open*ing process. The user file descriptor table identifies all open files for a process. Figure 2.2 shows the tables and their relationship to each other. The kernel returns a *file descriptor* for the *open* and *creat* system calls, which is an index into the user file descriptor table. When executing *read* and *write* system calls, the kernel uses the file descriptor to access the user file descriptor table, follows pointers to the file table and inode table entries, and, from the inode, finds the data in the file. Chapters 4 and 5 describe these data structures in great detail. For now, suffice it to say that use of three tables allows various degrees of sharing access to a file.

The UNIX system keeps regular files and directories on block devices such as tapes or disks. Because of the difference in access time between the two, few, if any, UNIX system installations use tapes for their file systems. In coming years, diskless work stations will be common, where files are located on a remote system and accessed via a network (see Chapter 13). For simplicity, however, the ensuing text assumes the use of disks. An installation may have several physical disk units, each containing one or more *file systems*. Partitioning a disk into several file systems makes it easier for administrators to manage the data stored there. The kernel deals on a logical level with file systems rather than with disks, treating each one as a *logical device* identified by a logical *device number*. The conversion between logical device (file system) addresses and physical device (disk) addresses is done by the disk driver. This book will use the term device to mean a logical device unless explicitly stated otherwise.

A file system consists of a sequence of logical blocks, each containing 512, 1024, 2048, or any convenient multiple of 512 bytes, depending on the system implementation. The size of a logical block is homogeneous within a file system but may vary between different file systems in a system configuration. Using large logical blocks increases the effective data transfer rate between disk and memory,

because the kernel can transfer more data per disk operation and therefore make fewer time-consuming operations. For example, reading 1K bytes from a disk in one read operation is faster than reading 512 bytes twice. However, if a logical block is too large, effective storage capacity may drop, as will be shown in Chapter 5. For simplicity, this book will use the term "block" to mean a logical block, and it will assume that a logical block contains 1K bytes of data unless explicitly stated otherwise.

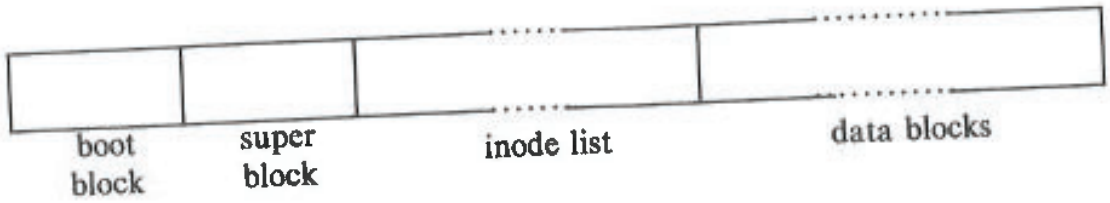


Figure 2.3. File System Layout

A file system has the following structure (Figure 2.3).

- The *boot block* occupies the beginning of a file system, typically the first sector, and may contain the *bootstrap* code that is read into the machine to *boot*, or initialize, the operating system. Although only one boot block is needed to boot the system, every file system has a (possibly empty) boot block.
- The *super block* describes the state of a file system — how large it is, how many files it can store, where to find free space on the file system, and other information.
- The *inode list* is a list of inodes that follows the super block in the file system. Administrators specify the size of the inode list when configuring a file system. The kernel references inodes by index into the inode list. One inode is the *root inode* of the file system: it is the inode by which the directory structure of the file system is accessible after execution of the *mount* system call (Section 5.14).
- The data blocks start at the end of the inode list and contain file data and administrative data. An allocated data block can belong to one and only one file in the file system.

2.2.2 Processes

This section examines the process subsystem more closely. It describes the structure of a process and some process data structures used for memory management. Then it gives a preliminary view of the process state diagram and considers various issues involved in some state transitions.

A process is the execution of a program and consists of a pattern of bytes that the CPU interprets as machine instructions (called "text"), data, and stack. Many processes appear to execute simultaneously as the kernel schedules them for execution, and several processes may be instances of one program. A process

executes by following a strict sequence of instructions that is self-contained and does not jump to that of another process; it reads and writes its data and stack sections, but it cannot read or write the data and stack of other processes. Processes communicate with other processes and with the rest of the world via system calls.

In practical terms, a process on a UNIX system is the entity that is created by the *fork* system call. Every process except *process 0* is created when another process executes the *fork* system call. The process that invoked the *fork* system call is the *parent* process, and the newly created process is the *child* process. Every process has one parent process, but a process can have many child processes. The kernel identifies each process by its process number, called the *process ID* (PID). Process 0 is a special process that is created "by hand" when the system boots; after *fork*ing a child process (process 1), process 0 becomes the *swapper* process. Process 1, known as *init*, is the ancestor of every other process in the system and enjoys a special relationship with them, as explained in Chapter 7.

A user compiles the source code of a program to create an executable file, which consists of several parts:

- a set of "headers" that describe the attributes of the file,
- the program text,
- a machine language representation of data that has initial values when the program starts execution, and an indication of how much space the kernel should allocate for uninitialized data, called *bss*[2] (the kernel initializes it to 0 at run time),
- other sections, such as symbol table information.

For the program in Figure 1.3, the text of the executable file is the generated code for the functions *main* and *copy*, the initialized data is the variable *version* (put into the program just so that it should have some initialized data), and the uninitialized data is the array *buffer*. System V versions of the C compiler create a separate text section by default but support an option that allows inclusion of program instructions in the data section, used in older versions of the system.

The kernel loads an executable file into memory during an *exec* system call, and the loaded process consists of at least three parts, called *region*s: text, data, and the stack. The text and data regions correspond to the text and data-bss sections of the executable file, but the stack region is automatically created and its size is dynamically adjusted by the kernel at run time. The stack consists of logical *stack frames* that are *push*ed when calling a function and *pop*ped when returning; a special register called the *stack pointer* indicates the current stack depth. A stack

2. The name bss comes from an assembly pseudo-operator on the IBM 7090 machine, which stood for "block started by symbol."

frame contains the parameters to a function, its local variables, and the data necessary to recover the previous stack frame, including the value of the program counter and stack pointer at the time of the function call. The program code contains instruction sequences that manage stack growth, and the kernel allocates space for the stack, as needed. In the program in Figure 1.3, parameters *argc* and *argv* and variables *fdold* and *fdnew* in the function *main* appear on the stack when *main* is called (once in every program, by convention), and parameters *old* and *new* and the variable *count* in the function *copy* appear on the stack whenever *copy* is called.

Because a process in the UNIX system can execute in two modes, kernel or user, it uses a separate stack for each mode. The user stack contains the arguments, local variables, and other data for functions executing in user mode. The left side of Figure 2.4 shows the user stack for a process when it makes the *write* system call in the *copy* program. The process startup procedure (included in a library) had called the function *main* with two parameters, pushing frame 1 onto the user stack; frame 1 contains space for the two local variables of *main*. *Main* then called *copy* with two parameters, *old* and *new*, and pushed frame 2 onto the user stack; frame 2 contains space for the local variable *count*. Finally, the process invoked the system call *write* by invoking the library function *write*. Each system call has an entry point in a system call library; the system call library is encoded in assembly language and contains special *trap* instructions, which, when executed, cause an "interrupt" that results in a hardware switch to kernel mode. A process calls the library entry point for a particular system call just as it calls any function, creating a stack frame for the library function. When the process executes the special instruction, it switches mode to the kernel, executes kernel code, and uses the kernel stack.

The kernel stack contains the stack frames for functions executing in kernel mode. The function and data entries on the kernel stack refer to functions and data in the kernel, not the user program, but its construction is the same as that of the user stack. The kernel stack of a process is null when the process executes in user mode. The right side of Figure 2.4 depicts the kernel stack representation for a process executing the *write* system call in the *copy* program. The names of the algorithms are described during the detailed discussion of the *write* system call in later chapters.

Every process has an entry in the kernel *process table*, and each process is allocated a *u area*[3] that contains private data manipulated only by the kernel. The process table contains (or points to) a *per process region table*, whose entries point to entries in a *region* table. A region is a contiguous area of a process's address

3. The *u* in *u area* stands for "user." Another name for the *u area* is *u block*; this book will always refer to it as the *u area*.

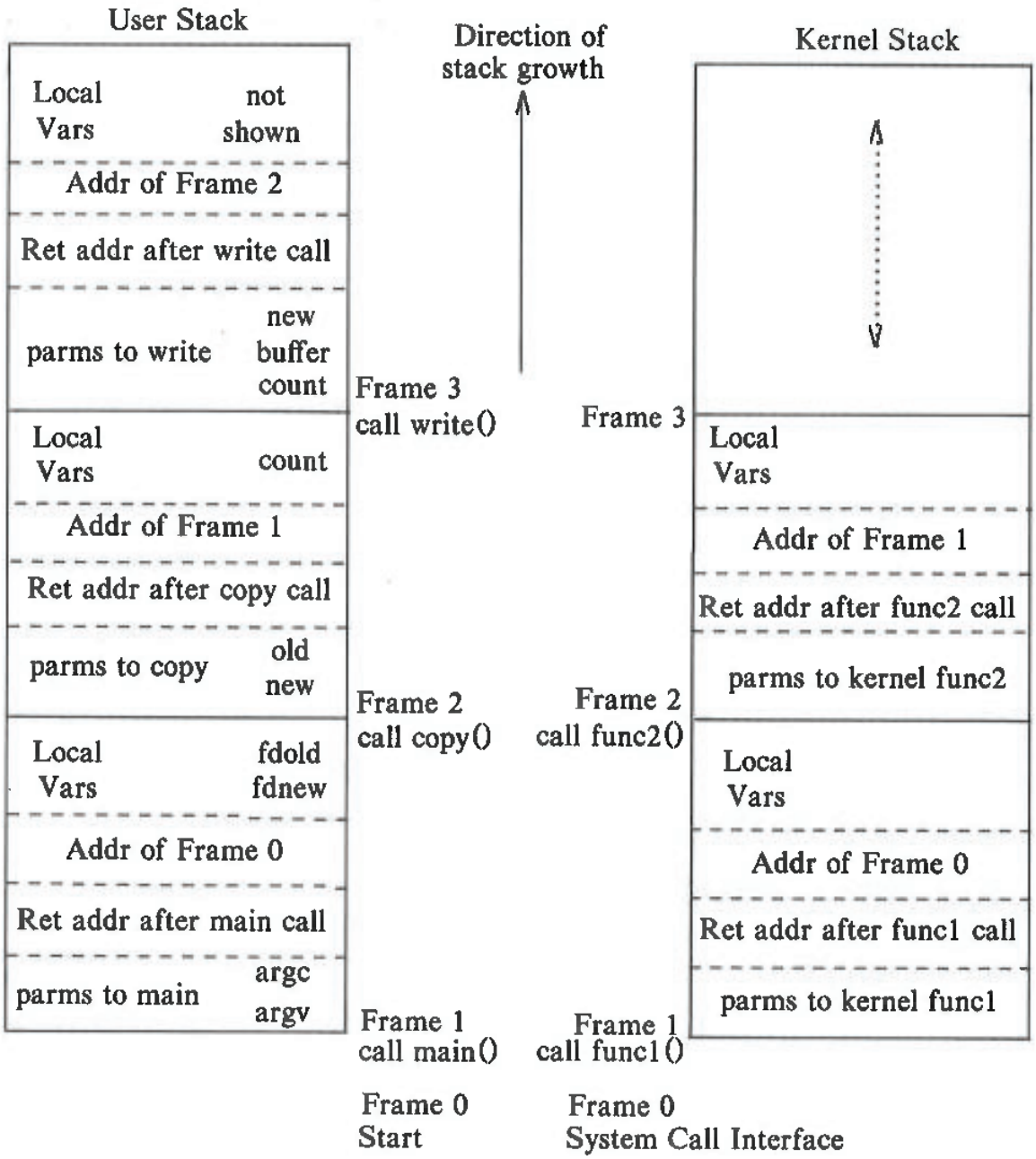


Figure 2.4. User and Kernel Stack for Copy Program

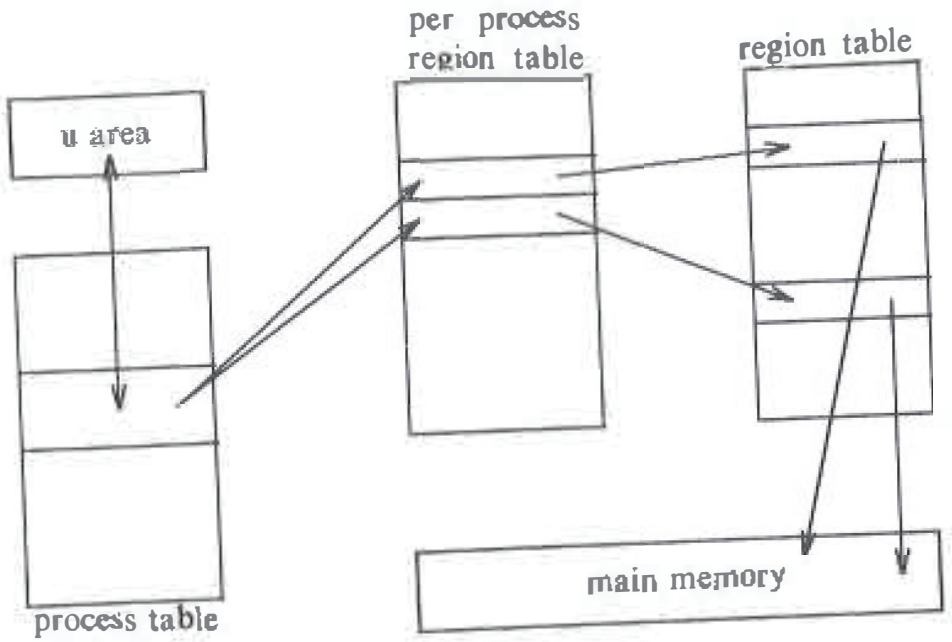


Figure 2.5. Data Structures for Processes

space, such as text, data, and stack. Region table entries describe the attributes of the region, such as whether it contains text or data, whether it is shared or private, and where the "data" of the region is located in memory. The extra level of indirection (from the per process region table to the region table) allows independent processes to share regions. When a process invokes the *exec* system call, the kernel allocates regions for its text, data, and stack after freeing the old regions the process had been using. When a process invokes *fork*, the kernel duplicates the address space of the old process, allowing processes to share regions when possible and making a physical copy otherwise. When a process invokes *exit*, the kernel frees the regions the process had used. Figure 2.5 shows the relevant data structures of a running process: The process table points to a per process region table with pointers to the region table entries for the text, data, and stack regions of the process.

The process table entry and the *u area* contain control and status information about the process. The *u area* is an extension of the process table entry, and Chapter 6 will examine the distinction between the two tables. Fields in the process table discussed in the following chapters are

- a state field,
- identifiers indicating the user who owns the process (user IDs, or UIDs),
- an event descriptor set when a process is suspended (in the *sleep* state).

The *u area* contains information describing the process that needs to be accessible only when the process is executing. The important fields are

- a pointer to the process table slot of the currently executing process,
- parameters of the current system call, return values and error codes,
- file descriptors for all open files,
- internal I/O parameters,
- current directory and current root (see Chapter 5),
- process and file size limits.

The kernel can directly access fields of the *u area* of the executing process but not of the *u area* of other processes. Internally, the kernel references the structure variable *u* to access the *u area* of the currently running process, and when another process executes, the kernel rearranges its virtual address space so that the structure *u* refers to the *u area* of the new process. The implementation gives the kernel an easy way to identify the current process by following the pointer from the *u area* to its process table entry.

2.2.2.1 Context of a process

The *context* of a process is its state, as defined by its text, the values of its global user variables and data structures, the values of machine registers it uses, the values stored in its process table slot and *u area*, and the contents of its user and kernel stacks. The text of the operating system and its global data structures are shared by all processes but do not constitute part of the context of a process.

When executing a process, the system is said to be executing in the context of the process. When the kernel decides that it should execute another process, it does a *context switch*, so that the system executes in the context of the other process. The kernel allows a context switch only under specific conditions, as will be seen. When doing a context switch, the kernel saves enough information so that it can later switch back to the first process and resume its execution. Similarly, when moving from user to kernel mode, the kernel saves enough information so that it can later return to user mode and continue execution from where it left off. Moving between user and kernel mode is a change in mode, not a context switch. Recalling Figure 1.5, the kernel does a context switch when it changes context from process A to process B; it changes execution mode from user to kernel or from kernel to user, still executing in the context of one process, such as process A.

The kernel services interrupts in the context of the interrupted process even though it may not have caused the interrupt. The interrupted process may have been executing in user mode or in kernel mode. The kernel saves enough information so that it can later resume execution of the interrupted process and services the interrupt in kernel mode. The kernel does not spawn or schedule a special process to handle interrupts.

2.2.2.2 Process states

The lifetime of a process can be divided into a set of *states*, each with certain characteristics that describe the process. Chapter 6 will describe all process states, but it is essential to understand the following states now:

1. The process is currently executing in user mode.
2. The process is currently executing in kernel mode.
3. The process is not executing, but it is ready to run as soon as the scheduler chooses it. Many processes may be in this state, and the scheduling algorithm determines which one will execute next.
4. The process is *sleeping*. A process puts itself to sleep when it can no longer continue executing, such as when it is waiting for I/O to complete.

Because a processor can execute only one process at a time, at most one process may be in states 1 and 2. The two states correspond to the two modes of execution, user and kernel.

2.2.2.3 State transitions

The process states described above give a static view of a process, but processes move continuously between the states according to well-defined rules. A *state transition* diagram is a directed graph whose *nodes* represent the states a process can enter and whose *edges* represent the events that cause a process to move from one state to another. State transitions are legal between two states if there exists an edge from the first state to the second. Several transitions may emanate from a state, but a process will follow one and only one transition depending on the system event that occurs. Figure 2.6 shows the state transition diagram for the process states defined above.

Several processes can execute simultaneously in a time-shared manner, as stated earlier, and they may all run simultaneously in kernel mode. If they were allowed to run in kernel mode without constraint, they could corrupt global kernel data structures. By prohibiting arbitrary context switches and controlling the occurrence of interrupts, the kernel protects its consistency.

The kernel allows a context switch only when a process moves from the state "kernel running" to the state "asleep in memory." Processes running in kernel mode cannot be preempted by other processes; therefore the kernel is sometimes said to be *non-preemptive*, although the system does preempt processes that are in user mode. The kernel maintains consistency of its data structures because it is non-preemptive, thereby solving the *mutual exclusion* problem — making sure that critical sections of code are executed by at most one process at a time.

For instance, consider the sample code in Figure 2.7 to put a data structure, whose address is in the pointer *bp1*, onto a doubly linked list after the structure whose address is in *bp*. If the system allowed a context switch while the kernel executed the code fragment, the following situation could occur. Suppose the

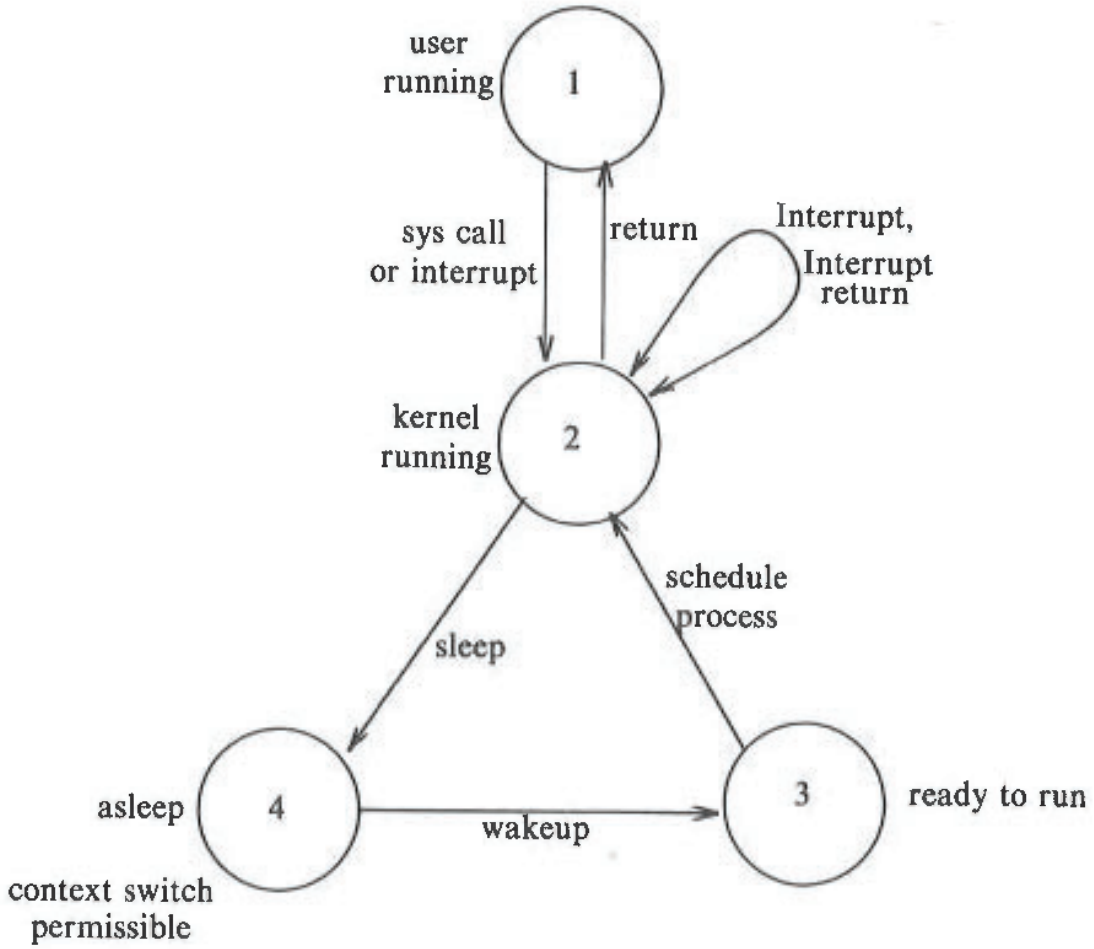


Figure 2.6. Process States and Transitions

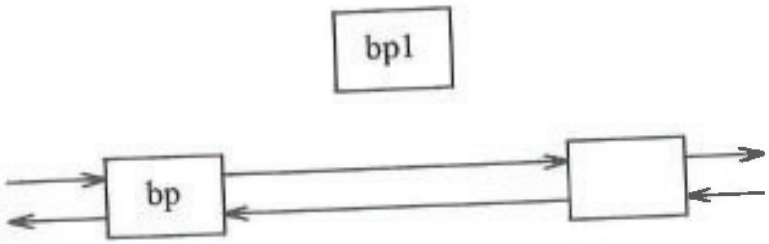
kernel executes the code until the comment and then does a context switch. The doubly linked list is in an inconsistent state: the structure *bp1* is half on and half off the linked list. If a process were to follow the forward pointers, it would find *bp1* on the linked list, but if it were to follow the back pointers, it would not find *bp1* (Figure 2.8). If other processes were to manipulate the pointers on the linked list before the original process ran again, the structure of the doubly linked list could be permanently destroyed. The UNIX system prevents such situations by disallowing context switches when a process executes in kernel mode. If a process goes to sleep, thereby permitting a context switch, kernel algorithms are encoded to make sure that system data structures are in a safe, consistent state.

A related problem that can cause inconsistency in kernel data is the handling of interrupts, which can change kernel state information. For example, if the kernel was executing the code in Figure 2.7 and received an interrupt when it reached the

```
struct queue {


} *bp, *bp1;
bp1->forp = bp->forp;
bp1->backp = bp;
bp->forp = bp1;
/* consider possible context switch here */
bp1->forp->backp = bp1;
```

Figure 2.7. Sample Code Creating Doubly Linked List



Placing bp1 on doubly linked list

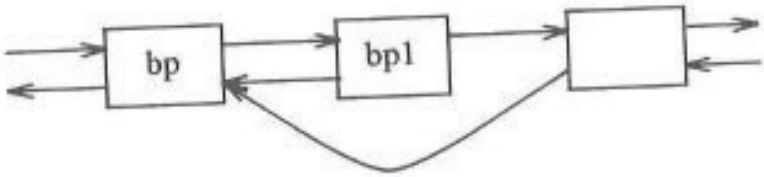


Figure 2.8. Incorrect Linked List because of Context Switch

comment, the interrupt handler could corrupt the links if it manipulates the pointers, as illustrated earlier. To solve this problem, the system could prevent all interrupts while executing in kernel mode, but that would delay servicing of the interrupt, possibly hurting system throughput. Instead, the kernel raises the processor execution level to prevent interrupts when entering *critical* regions of code. A section of code is critical if execution of arbitrary interrupt handlers could result in consistency problems. For example, if a disk interrupt handler manipulates the buffer queues in the figure, the section of code where the kernel manipulates the buffer queues is a critical region of code with respect to the disk interrupt handler. Critical regions are small and infrequent so that system throughput is largely unaffected by their existence. Other operating systems solve this problem by preventing all interrupts when executing in system states or by using elaborate locking schemes to ensure consistency. Chapter 12 will return to

this issue for multiprocessor systems, where the solution outlined here is insufficient.

To review, the kernel protects its consistency by allowing a context switch only when a process puts itself to sleep and by preventing one process from changing the state of another process. It also raises the processor execution level around critical regions of code to prevent interrupts that could otherwise cause inconsistencies. The process scheduler periodically preempts processes executing in user mode so that processes cannot monopolize use of the CPU.

2.2.2.4 Sleep and wakeup

A process executing in kernel mode has great autonomy in deciding what it is going to do in reaction to system events. Processes can communicate with each other and "suggest" various alternatives, but they make the final decision by themselves. As will be seen, there is a set of rules that processes obey when confronted with various circumstances, but each process ultimately follows these rules under its own initiative. For instance, when a process must temporarily suspend its execution ("go to sleep"), it does so of its own free will. Consequently, an interrupt handler cannot go to sleep, because if it could, the interrupted process would be put to sleep by default.

Processes go to sleep because they are awaiting the occurrence of some event, such as waiting for I/O completion from a peripheral device, waiting for a process to exit, waiting for system resources to become available, and so on. Processes are said to *sleep on an event*, meaning that they are in the sleep state until the event occurs, at which time they wake up and enter the state "ready to run." Many processes can simultaneously sleep on an event; when an event occurs, *all* processes sleeping on the event wake up because the event condition is no longer true. When a process wakes up, it follows the state transition from the "sleep" state to the "ready-to-run" state, where it is eligible for later scheduling; it does *not* execute immediately. Sleeping processes do not consume CPU resources: The kernel does not constantly check to see that a process is still sleeping but waits for the event to occur and awakens the process then.

For example, a process executing in kernel mode must sometimes lock a data structure in case it goes to sleep at a later stage; processes attempting to manipulate the locked structure must check the lock and sleep if another process owns the lock. The kernel implements such locks in the following manner:

```
while (condition is true)
        sleep (event: the condition becomes false);
set condition true;
```

It unlocks the lock and awakens all processes asleep on the lock in the following manner:

```
set condition false;
wakeup (event: the condition is false);
```

Figure 2.9 depicts a scenario where three processes, A, B, and C, contend for a locked buffer. The sleep condition is that the buffer is locked. The processes execute one at a time, find the buffer locked, and sleep on the event that the buffer becomes unlocked. Eventually, the buffer is unlocked, and all processes wake up and enter the state "ready to run." The kernel eventually chooses one process, say B, to execute. Process B executes the "while" loop, finds that the buffer is unlocked, sets the buffer lock, and proceeds. If process B later goes to sleep again before unlocking the buffer (waiting for completion of an I/O operation, for example), the kernel can schedule other processes to run. If it chooses process A, process A executes the "while" loop, finds that the buffer is locked, and goes to sleep again; process C may do the same thing. Eventually, process B awakens and unlocks the buffer, allowing either process A or C to gain access to the buffer. Thus, the "while-sleep" loop insures that at most one process can gain access to a resource.

Chapter 6 will present the algorithms for sleep and wakeup in greater detail. In the meantime, they should be considered "atomic": A process enters the sleep state instantaneously and stays there until it wakes up. After it goes to sleep, the kernel schedules another process to run and switches context to it.

2.3 KERNEL DATA STRUCTURES

Most kernel data structures occupy fixed-size tables rather than dynamically allocated space. The advantage of this approach is that the kernel code is simple, but it limits the number of entries for a data structure to the number that was originally configured when generating the system: If, during operation of the system, the kernel should run out of entries for a data structure, it cannot allocate space for new entries dynamically but must report an error to the requesting user. If, on the other hand, the kernel is configured so that it it is unlikely to run out of table space, the extra table space may be wasted because it cannot be used for other purposes. Nevertheless, the simplicity of the kernel algorithms has generally been considered more important than the need to squeeze out every last byte of main memory. Algorithms typically use simple loops to find free table entries, a method that is easier to understand and sometimes more efficient than more complicated allocation schemes.

2.4 SYSTEM ADMINISTRATION

Administrative processes are loosely classified as those processes that do various functions for the general welfare of the user community. Such functions include disk formatting, creation of new file systems, repair of damaged file systems, kernel debugging, and others. Conceptually, there is no difference between administrative

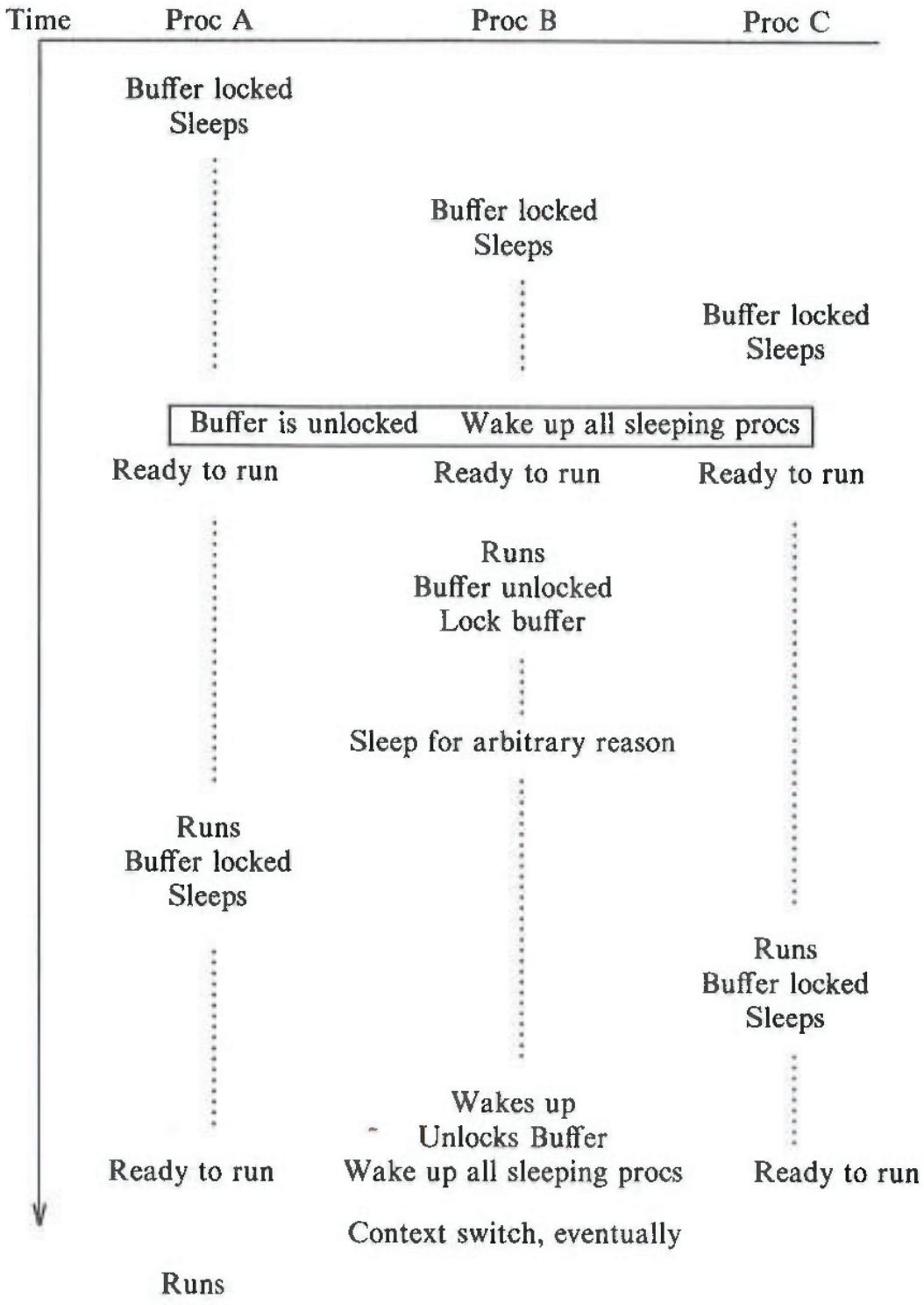


Figure 2.9. Multiple Processes Sleeping on a Lock

processes and user processes: They use the same set of system calls available to the general community. They are distinguished from general user processes only in the rights and privileges they are allowed. For example, file permission modes may allow administrative processes to manipulate files otherwise off-limits to general users. Internally, the kernel distinguishes a special user called the *superuser*, endowing it with special privileges, as will be seen. A user may become a superuser by going through a login-password sequence or by executing special programs. Other uses of superuser privileges will be studied in later chapters. In short, the kernel does not recognize a separate class of administrative processes.

2.5 SUMMARY AND PREVIEW

This chapter has described the architecture of the kernel; its two major components are the file subsystem and the process subsystem. The file subsystem controls the storage and retrieval of data in user files. Files are organized into file systems, which are treated as logical devices; a physical device such as a disk can contain several logical devices (file systems). Each file system has a super block that describes the structure and contents of the file system, and each file in a file system is described by an inode that gives the attributes of the file. System calls that manipulate files do so via inodes.

Processes exist in various states and move between them according to well-defined transition rules. In particular, processes executing in kernel mode can suspend their execution and enter the sleep state, but no process can put another process to sleep. The kernel is non-preemptive, meaning that a process executing in kernel mode will continue to execute until it enters the sleep state or until it returns to execute in user mode. The kernel maintains the consistency of its data structures by enforcing the policy of non-preemption and by blocking interrupts when executing critical regions of code.

The remainder of this text describes the subsystems shown in Figure 2.1 and their interactions in detail, starting with the file subsystem and continuing with the process subsystem. The next chapter covers the buffer cache and describes buffer allocation algorithms, used in the algorithms presented in Chapters 4, 5, and 7. Chapter 4 examines internal algorithms of the file system, including the manipulation of inodes, the structure of files, and the conversion of path names to inodes. Chapter 5 explains the system calls that use the algorithms in Chapter 4 to access the file system, such as *open*, *close*, *read*, and *write*. Chapter 6 deals with the basic ideas of the context of a process and its address space, and Chapter 7 covers system calls that deal with process management and use the algorithms in Chapter 6. Chapter 8 examines process scheduling, and Chapter 9 discusses memory management algorithms. Chapter 10 covers device drivers, postponed to this point so that the relationship between the terminal driver and process management can be explained. Chapter 11 presents several forms of interprocess communication. Finally, the last two chapters cover advanced topics, including multiprocessor systems and distributed systems.

2.6 EXERCISES

1. Consider the following sequence of commands:

```
grep main a.c b.c c.c > grepout &
wc -l < grepout &
rm grepout &
```

The ampersand ("&") at the end of each command line informs the shell to run the command in the background, and it can execute each command line in parallel. Why is this not equivalent to the following command line?

```
grep main a.c b.c c.c | wc -l
```

2. Consider the sample kernel code in Figure 2.7. Suppose a context switch happens when the code reaches the comment, and suppose another process removes a buffer from the linked list by executing the following code:

```
remove(qp)
        struct queue *qp;
{
        qp->forp->backp = qp->backp;
        qp->backp->forp = qp->forp;
        qp->forp = qp->backp = NULL;
}
```

Consider three cases:
- The process removes the structure *bp1* from the linked list.
- The process removes the structure that currently follows *bp1* on the linked list.
- The process removes the structure that originally followed *bp1* before *bp* was half placed on the linked list.

What is the status of the linked list after the original process completes executing the code after the comment?

3. What should happen if the kernel attempts to awaken all processes sleeping on an event, but no processes are asleep on the event at the time of the wakeup?

8. UNIX System V allows a maximum of 14 characters for a path name component. *Namei* truncates extra characters in a component. How should the file system and respective algorithms be redesigned to allow arbitrary length component names?
9. Suppose a user has a private version of the UNIX system but changes it so that a path name component can consist of 30 characters; the private version of the operating system stores the directory entries the same way that the standard operating system does, except that the directory entries are 32 bytes long instead of 16. If the user mounts the private file system on a standard system, what would happen in algorithm *namei* when a process accesses a file on the private file system?

* 10. Consider the algorithm *namei* for converting a path name into an inode. As the search progresses, the kernel checks that the current working inode is that of a directory. Is it possible for another process to remove (*unlink*) the directory? How can the kernel prevent this? The next chapter will come back to this problem.

* 11. Design a directory structure that improves the efficiency of searching for path names by avoiding the linear search. Consider two techniques: hashing and n-ary trees.

* 12. Design a scheme that reduces the number of directory searches for file names by caching frequently used names.

* 13. Ideally, a file system should never contain a free inode whose inode number is less than the "remembered" inode used by *ialloc*. How is it possible for this assertion to be false?

14. The super block is a disk block and contains other information besides the free block list, as described in this chapter. Therefore, the super block free list cannot contain as many free block numbers as can be potentially stored in a disk block on the linked list of free disk blocks. What is the optimal number of free block numbers that should be stored in a block on the linked list?

* 15. Discuss a system implementation that keeps track of free disk blocks with a bit map instead of a linked list of blocks. What are the advantages and disadvantages of this scheme?

5

SYSTEM CALLS FOR THE FILE SYSTEM

The last chapter described the internal data structures for the file system and the algorithms that manipulate them. This chapter deals with system calls for the file system, using the concepts explored in the previous chapter. It starts with system calls for accessing existing files, such as *open*, *read*, *write*, *lseek*, and *close*, then presents system calls to create new files, namely, *creat* and *mknod*, and then examines the system calls that manipulate the inode or that maneuver through the file system: *chdir*, *chroot*, *chown*, *chmod*, *stat*, and *fstat*. It investigates more advanced system calls: *pipe* and *dup* are important for the implementation of pipes in the shell; *mount* and *umount* extend the file system tree visible to users; *link* and *unlink* change the structure of the file system hierarchy. Then, it presents the notion of file system abstractions, allowing the support of various file systems as long as they conform to standard interfaces. The last section in the chapter covers file system maintenance. The chapter introduces three kernel data structures: the file table, with one entry allocated for every opened file in the system, the user file descriptor table, with one entry allocated for every file descriptor known to a process, and the mount table, containing information for every active file system.

Figure 5.1 shows the relationship between the system calls and the algorithms described previously. It classifies the system calls into several categories, although some system calls appear in more than one category:

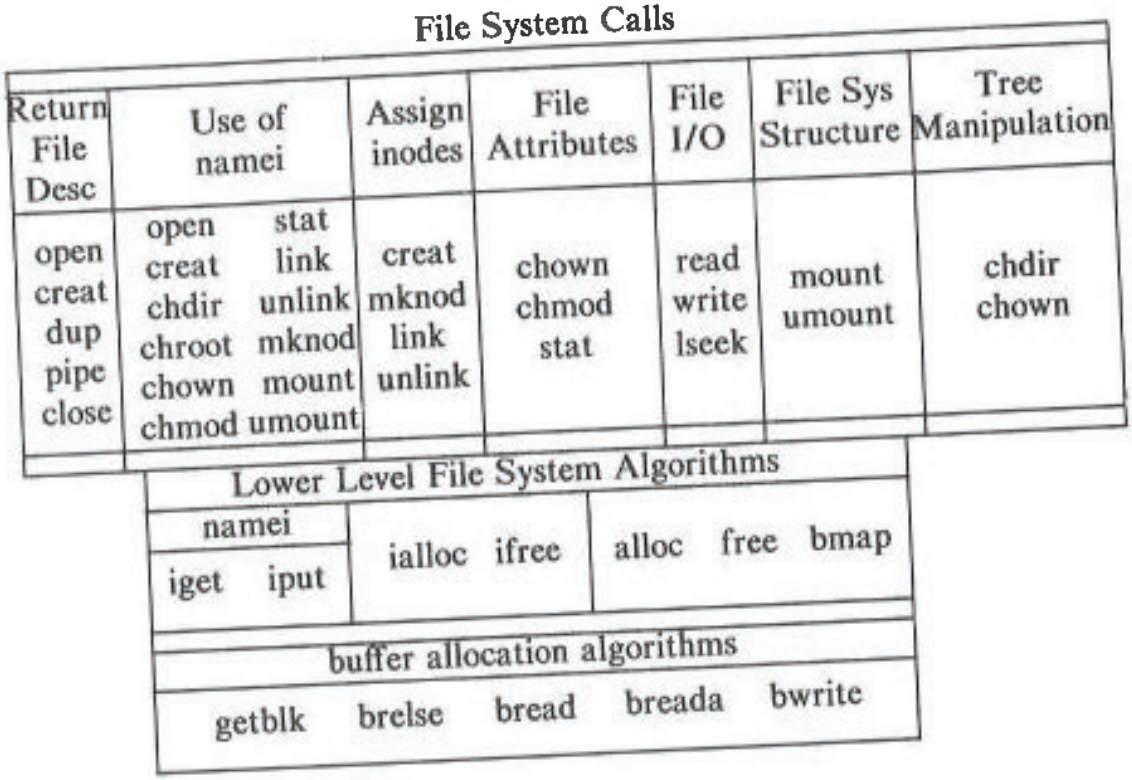


Figure 5.1. File System Calls and Relation to Other Algorithms

- System calls that return file descriptors for use in other system calls;
- System calls that use the *namei* algorithm to parse a path name;
- System calls that assign and free inodes, using algorithms *ialloc* and *ifree*;
- System calls that set or change the attributes of a file;
- System calls that do I/O to and from a process, using algorithms *alloc*, *free*, and the buffer allocation algorithms;
- System calls that change the structure of the file system;
- System calls that allow a process to change its view of the file system tree.

5.1 OPEN

The *open* system call is the first step a process must take to access the data in a file. The syntax for the *open* system call is

```
fd = open(pathname, flags, modes);
```

where *pathname* is a file name, *flags* indicate the type of open (such as for reading or writing), and *modes* give the file permissions if the file is being created. The *open* system call returns an integer[1] called the user *file descriptor*. Other file

operations, such as reading, writing, seeking, duplicating the file descriptor, setting file I/O parameters, determining file status, and closing the file, use the file descriptor that the *open* system call returns.

The kernel searches the file system for the file name parameter using algorithm *namei* (see Figure 5.2). It checks permissions for opening the file after it finds the in-core inode and allocates an entry in the file table for the open file. The file table entry contains a pointer to the inode of the open file and a field that indicates the byte offset in the file where the kernel expects the next *read* or *write* to begin. The kernel initializes the offset to 0 during the *open* call, meaning that the initial *read* or *write* starts at the beginning of a file by default. Alternatively, a process can *open* a file in *write-append* mode, in which case the kernel initializes the offset to the size of the file. The kernel allocates an entry in a private table in the process *u area*, called the user file descriptor table, and notes the index of this entry. The index is the file descriptor that is returned to the user. The entry in the user file table points to the entry in the global file table.

```
algorithm open
inputs:  file name
         type of open
         file permissions (for creation type of open)
output: file descriptor
{
        convert file name to inode (algorithm namei);
        if (file does not exist or not permitted access)
                return(error);
        allocate file table entry for inode, initialize count, offset;
        allocate user file descriptor entry, set pointer to file table entry;
        if (type of open specifies truncate file)
                free all file blocks (algorithm free);
        unlock(inode);                  /* locked above in namei */
        return(user file descriptor);
}
```

Figure 5.2. Algorithm for Opening a File

Suppose a process executes the following code, opening the file "/etc/passwd" twice, once read-only and once write-only, and the file "local" once, for reading and writing.[2]

1. All system calls return the value −1 if they fail. The return value −1 will not be explicitly mentioned when discussing the syntax of the system calls.

2. The definition of the *open* system call specifies three parameters (the third is used for the *create* mode of open), but programmers usually use only the first two. The C compiler does not check that the number of parameters is correct. System implementations typically pass the first two parameters and a third "garbage" parameter (whatever happens to be on the stack) to the kernel. The kernel

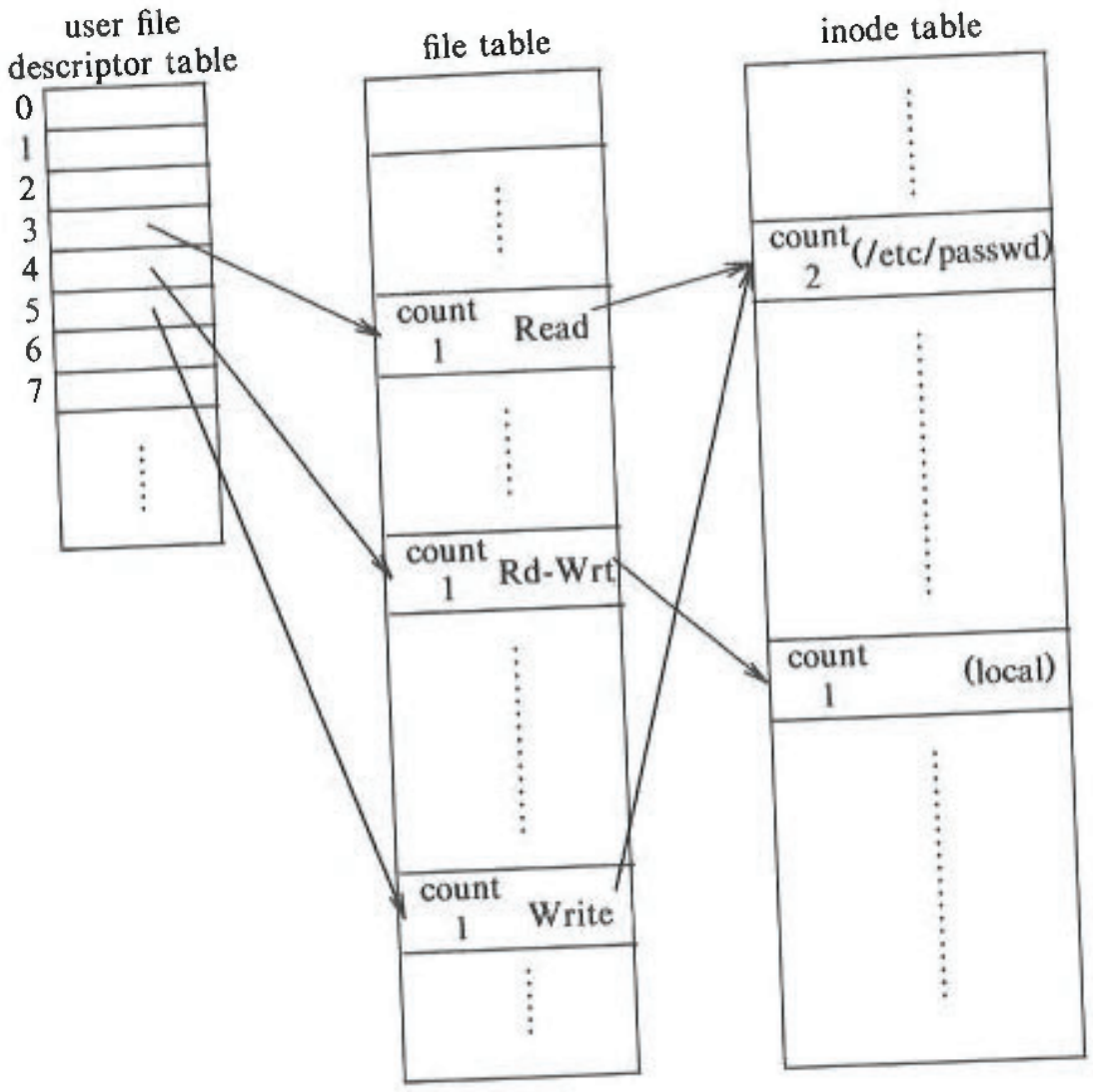


Figure 5.3. Data Structures after Open

```
fd1 = open("/etc/passwd", O_RDONLY);
fd2 = open("local", O_RDWR);
fd3 = open("/etc/passwd", O_WRONLY);
```

Figure 5.3 shows the relationship between the inode table, file table, and user file descriptor data structures. Each *open* returns a file descriptor to the process, and the corresponding entry in the user file descriptor table points to a unique entry in

---

does not check the third parameter unless the second parameter indicates that it must, allowing programmers to encode only two parameters.

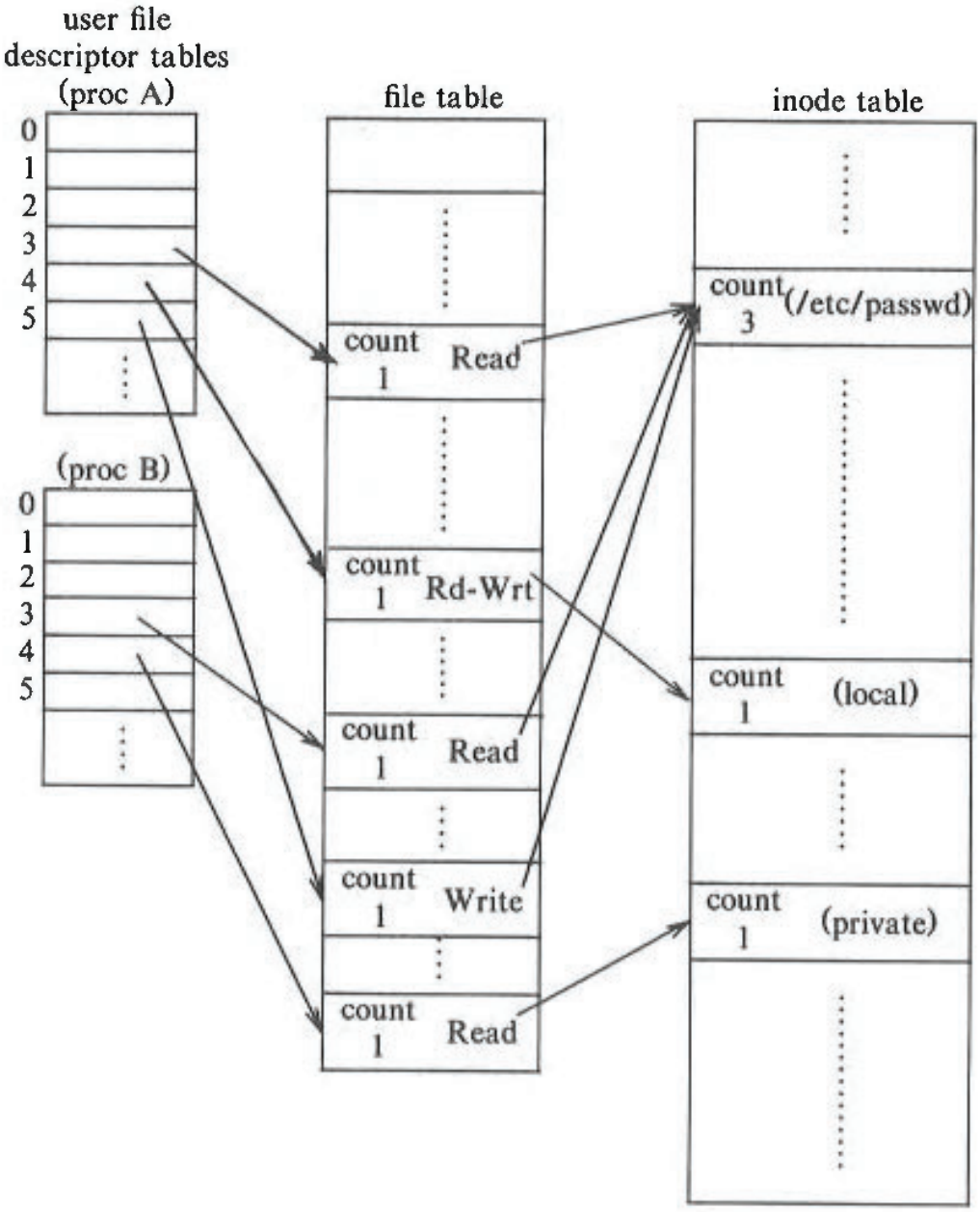


Figure 5.4. Data Structures after Two Processes Open Files

the kernel file table even though one file ("/etc/passwd") is opened twice. The file table entries of all instances of an open file point to one entry in the in-core inode table. The process can *read* or *write* the file "/etc/passwd" but only through file descriptors 3 and 5 in the figure. The kernel notes the capability to read or write the file in the file table entry allocated during the *open* call. Suppose a second process executes the following code.

```
fd1 = open("/etc/passwd", O_RDONLY);
fd2 = open("private", O_RDONLY);
```

Figure 5.4 shows the relationship between the appropriate data structures while both processes (and no others) have the files open. Again, each *open* call results in allocation of a unique entry in the user file descriptor table and in the kernel file table, but the kernel contains at most one entry per file in the in-core inode table.

The user file descriptor table entry could conceivably contain the file offset for the position of the next I/O operation and point directly to the in-core inode entry for the file, eliminating the need for a separate kernel file table. The examples above show a one-to-one relationship between user file descriptor entries and kernel file table entries. Thompson notes, however, that he implemented the file table as a separate structure to allow sharing of the offset pointer between several user file descriptors (see page 1943 of [Thompson 78]). The *dup* and *fork* system calls, explained in Sections 5.13 and 7.1, manipulate the data structures to allow such sharing.

The first three user file descriptors (0, 1, and 2) are called the *standard input*, *standard output*, and *standard error* file descriptors. Processes on UNIX systems conventionally use the standard input descriptor to read input data, the standard output descriptor to write output data, and the standard error descriptor to write error data (messages). Nothing in the operating system assumes that these file descriptors are special. A group of users could adopt the convention that file descriptors 4, 6, and 11 are special file descriptors, but counting from 0 (in C) is much more natural. Adoption of the convention by all user programs makes it easy for them to communicate via *pipes*, as will be seen in Chapter 7. Normally, the control terminal (see Chapter 10) serves as standard input, standard output and standard error.

5.2 READ

The syntax of the *read* system call is

```
number = read(fd, buffer, count)
```

where *fd* is the file descriptor returned by *open*, *buffer* is the address of a data structure in the user process that will contain the read data on successful completion of the call, *count* is the number of bytes the user wants to read, and *number* is the number of bytes actually read. Figure 5.5 depicts the algorithm *read* for reading a regular file. The kernel gets the file table entry that corresponds to

```
algorithm read
input:   user file descriptor
         address of buffer in user process
         number of bytes to read
output: count of bytes copied into user space
{
        get file table entry from user file descriptor;
        check file accessibility;
        set parameters in u area for user address, byte count, I/O to user;
        get inode from file table;
        lock inode;
        set byte offset in u area from file table offset;
        while (count not satisfied)
        {
                convert file offset to disk block (algorithm bmap);
                calculate offset into block, number of bytes to read;
                if (number of bytes to read is 0)
                        /* trying to read end of file */
                        break;                  /* out of loop */
                read block (algorithm breada if with read ahead, algorithm
                                bread otherwise);
                copy data from system buffer to user address;
                update u area fields for file byte offset, read count,
                                address to write into user space;
                release buffer;                 /* locked in bread */
        }
        unlock inode;
        update file table offset for next read;
        return(total number of bytes read);
}
```

Figure 5.5. Algorithm for Reading a File

the user file descriptor, following the pointer in Figure 5.3. It now sets several I/O parameters in the *u area* (Figure 5.6), eliminating the need to pass them as function parameters. Specifically, it sets the I/O mode to indicate that a read is being done, a flag to indicate that the I/O will go to user address space, a count field to indicate the number of bytes to read, the target address of the user data buffer, and finally, an offset field (from the file table) to indicate the byte offset into the file where the I/O should begin. After the kernel sets the I/O parameters in the *u area*, it follows the pointer from the file table entry to the inode, locking the inode before it reads the file.

The algorithm now goes into a loop until the *read* is satisfied. The kernel converts the file byte offset into a block number, using algorithm *bmap*, and it notes the byte offset in the block where the I/O should begin and how many bytes

mode	indicates read or write
count	count of bytes to read or write
offset	byte offset in file
address	target address to copy data, in user or kernel memory
flag	indicates if address is in user or kernel memory

Figure 5.6. I/O Parameters Saved in U Area

in the block it should read. After reading the block into a buffer, possibly using block read ahead (algorithms *bread* and *breada*) as will be described, it copies the data from the block to the target address in the user process. It updates the I/O parameters in the *u area* according to the number of bytes it read, incrementing the file byte offset and the address in the user process where the next data should be delivered, and decrementing the count of bytes it needs to read to satisfy the user read request. If the user request is not satisfied, the kernel repeats the entire cycle, converting the file byte offset to a block number, reading the block from disk to a system buffer, copying data from the buffer to the user process, releasing the buffer, and updating I/O parameters in the *u area*. The cycle completes either when the kernel completely satisfies the user request, when the file contains no more data, or if the kernel encounters an error in reading the data from disk or in copying the data to user space. The kernel updates the offset in the file table according to the number of bytes it actually read; consequently, successive *read*s of a file deliver the file data in sequence. The *lseek* system call (Section 5.6) adjusts the value of the file table offset and changes the order in which a process *read*s or *write*s a file.

```
#include <fcntl.h>
main()
{
        int fd;
        char lilbuf[20], bigbuf[1024];

        fd = open("/etc/passwd", O_RDONLY);
        read(fd, lilbuf, 20);
        read(fd, bigbuf, 1024);
        read(fd, lilbuf, 20);
}
```

Figure 5.7. Sample Program for Reading a File

Consider the program in Figure 5.7. The *open* returns a file descriptor that the user assigns to the variable *fd* and uses in the subsequent *read* calls. In the *read* system call, the kernel verifies that the file descriptor parameter is legal, and that

the process had previously *open*ed the file for reading. It stores the values *lilbuf*, 20, and 0 in the *u area*, corresponding to the address of the user buffer, the byte count, and the starting byte offset in the file. It calculates that byte offset 0 is in the 0th block of the file and retrieves the entry for the 0th block in the inode. Assuming such a block exists, the kernel reads the entire block of 1024 bytes into a buffer but copies only 20 bytes to the user address *lilbuf*. It increments the *u area* byte offset to 20 and decrements the count of data to read to 0. Since the *read* has been satisfied, the kernel resets the file table offset to 20, so that subsequent *read*s on the file descriptor will begin at byte 20 in the file, and the system call returns the number of bytes actually read, 20.

For the second *read* call, the kernel again verifies that the descriptor is legal and that the process had *open*ed the file for reading, because it has no way of knowing that the user *read* request is for the same file that was determined to be legal during the last *read*. It stores in the *u area* the user address *bigbuf*, the number of bytes the process wants to read, 1024, and the starting offset in the file, 20, taken from the file table. It converts the file byte offset to the correct disk block, as above, and reads the block. If the time between *read* calls is small, chances are good that the block will be in the buffer cache. But the kernel cannot satisfy the *read* request entirely from the buffer, because only 1004 out of the 1024 bytes for this request are in the buffer. So it copies the last 1004 bytes from the buffer into the user data structure *bigbuf* and updates the parameters in the *u area* to indicate that the next iteration of the read loop starts at byte 1024 in the file, that the data should be copied to byte position 1004 in *bigbuf*, and that the number of bytes to to satisfy the *read* request is 20.

The kernel now cycles to the beginning of the loop in the *read* algorithm. It converts byte offset 1024 to logical block offset 1, looks up the second direct block number in the inode, and finds the correct disk block to read. It reads the block from the buffer cache, reading the block from disk if it is not in the cache. Finally, it copies 20 bytes from the buffer to the correct address in the user process. Before leaving the system call, the kernel sets the offset field in the file table entry to 1044, the byte offset that should be accessed next. For the last *read* call in the example, the kernel proceeds as in the first *read* call, except that it starts reading at byte 1044 in the file, finding that value in the offset field in the file table entry for the descriptor.

The example shows how advantageous it is for I/O requests to start on file system block boundaries and to be multiples of the block size. Doing so allows the kernel to avoid an extra iteration in the *read* algorithm loop, with the consequent expense of accessing the inode to find the correct block number for the data and competing with other processes for access to the buffer pool. The standard I/O library was written to hide knowledge of the kernel buffer size from users; its use avoids the performance penalties inherent in processes that nibble at the file system inefficiently (see exercise 5.4).

As the kernel goes through the *read* loop, it determines whether a file is subject to read-ahead: if a process *read*s two blocks sequentially, the kernel assumes that

all subsequent *read*s will be sequential until proven otherwise. During each iteration through the loop, the kernel saves the next logical block number in the in-core inode and, during the next iteration, compares the current logical block number to the value previously saved. If they are equal, the kernel calculates the physical block number for read-ahead and saves its value in the *u area* for use in the *breada* algorithm. Of course, if a process does not *read* to the end of a block, the kernel does not invoke read-ahead for the next block.

Recall from Figure 4.9 that it is possible for some block numbers in an inode or in indirect blocks to have the value 0, even though later blocks have nonzero value. If a process attempts to *read* data from such a block, the kernel satisfies the request by allocating an arbitrary buffer in the *read* loop, clearing its contents to 0, and copying it to the user address. This case is different from the case where a process encounters the end of a file, meaning that no data was ever written to any location beyond the current point. When encountering end of file, the kernel returns no data to the process (see exercise 5.1).

When a process invokes the *read* system call, the kernel locks the inode for the duration of the call. Afterwards, it could go to sleep reading a buffer associated with data or with indirect blocks of the inode. If another process were allowed to change the file while the first process was sleeping, *read* could return inconsistent data. For example, a process may *read* several blocks of a file; if it slept while reading the first block and a second process were to *write* the other blocks, the returned data would contain a mixture of old and new data. Hence, the inode is left locked for the duration of the *read* call, affording the process a consistent view of the file as it existed at the start of the call.

The kernel can preempt a *read*ing process between system calls in user mode and schedule other processes to run. Since the inode is unlocked at the end of a system call, nothing prevents other processes from accessing the file and changing its contents. It would be unfair for the system to keep an inode locked from the time a process *open*ed the file until it *close*d the file, because one process could keep a file open and thus prevent other processes from ever accessing it. If the file was "/etc/passwd", used by the login process to check a user's password, then one malicious (or, perhaps, just errant) user could prevent all other users from logging in. To avoid such problems, the kernel unlocks the inode at the end of each system call that uses it. If another process changes the file between the two *read* system calls by the first process, the first process may *read* unexpected data, but the kernel data structures are consistent.

For example, suppose the kernel executes the two processes in Figure 5.8 concurrently. Assuming both processes complete their *open* calls before either one starts its *read* or *write* calls, the kernel could execute the *read* and *write* calls in any of six sequences: *read1, read2, write1, write2*, or *read1, write1, read2, write2*, or *read1, write1, write2, read2*, and so on. The data that process A *read*s depends on the order that the system executes the system calls of the two processes; the system does not guarantee that the data in the file remains the same after *open*ing the file. Use of the *file and record locking* feature (Section 5.4) allows a process to

```
#include <fcntl.h>
/* process A */
main()
{
        int fd;
        char buf[512];
        fd = open("/etc/passwd", O_RDONLY);
        read(fd, buf, sizeof(buf));        /* read1 */
        read(fd, buf, sizeof(buf));        /* read2 */
}

/* process B */
main()
{
        int fd, i;
        char buf[512];
        for (i = 0;  i < sizeof(buf);  i++)
                buf[i] = 'a';
        fd = open("/etc/passwd", O_WRONLY);
        write(fd, buf, sizeof(buf));       /* write1 */
        write(fd, buf, sizeof(buf));       /* write2 */
}
```

Figure 5.8. A Reader and a Writer Process

guarantee file consistency while it has a file *open*.

Finally, the program in Figure 5.9 shows how a process can *open* a file more than once and *read* it via different file descriptors. The kernel manipulates the file table offsets associated with the two file descriptors independently, and hence, the arrays *buf1* and *buf2* should be identical when the process completes, assuming no other process *writes* "/etc/passwd" in the meantime.

5.3 WRITE

The syntax for the *write* system call is

```
number = write(fd, buffer, count);
```

where the meaning of the variables *fd*, *buffer*, *count*, and *number* are the same as they are for the *read* system call. The algorithm for *writ*ing a regular file is similar to that for *read*ing a regular file. However, if the file does not contain a block that corresponds to the byte offset to be written, the kernel allocates a new block using algorithm *alloc* and assigns the block number to the correct position in the inode's table of contents. If the byte offset is that of an indirect block, the kernel may

```
#include <fcntl.h>
main()
{
        int fd1, fd2;
        char buf1[512], buf2[512];

        fd1 = open("/etc/passwd", O_RDONLY);
        fd2 = open("/etc/passwd", O_RDONLY);
        read(fd1, buf1, sizeof(buf1));
        read(fd2, buf2, sizeof(buf2));
}
```

Figure 5.9. Reading a File via Two File Descriptors

have to allocate several blocks for use as indirect blocks and data blocks. The inode is locked for the duration of the *write*, because the kernel may change the inode when allocating new blocks; allowing other processes access to the file could corrupt the inode if several processes allocate blocks simultaneously for the same byte offsets. When the write is complete, the kernel updates the file size entry in the inode if the file has grown larger.

For example, suppose a process writes byte number 10,240 to a file, the highest-numbered byte yet written to the file. When accessing the byte in the file using algorithm *bmap*, the kernel will find not only that the file does not contain a block for that byte but also that it does not contain the necessary indirect block. It assigns a disk block for the indirect block and writes the block number in the in-core inode. Then it assigns a disk block for the data block and writes its block number into the first position in the newly assigned indirect block.

The kernel goes through an internal loop, as in the *read* algorithm, writing one block to disk during each iteration. During each iteration, it determines whether it will write the entire block or only part of it. If it writes only part of a block, it must first read the block from disk so as not to overwrite the parts that will remain the same, but if it writes the whole block, it need not read the block, since it will overwrite its previous contents anyway. The write proceeds block by block, but the kernel uses a *delayed write* (Section 3.4) to write the data to disk, caching it in case another process should *read* or *write* it soon and avoiding extra disk operations. Delayed write is probably most effective for pipes, because another process is reading the pipe and removing its data (Section 5.12). But even for regular files, delayed write is effective if the file is created temporarily and will be read soon. For example, many programs, such as editors and mail, create temporary files in the directory "/tmp" and quickly remove them. Use of delayed write can reduce

the number of disk writes for temporary files.

5.4 FILE AND RECORD LOCKING

The original UNIX system developed by Thompson and Ritchie did not have an internal mechanism by which a process could insure exclusive access to a file. A locking mechanism was considered unnecessary because, as Ritchie notes, "we are not faced with large, single-file databases maintained by independent processes" (see [Ritchie 81]). To make the UNIX system more attractive to commercial users with database applications, System V now contains file and record locking mechanisms. File locking is the capability to prevent other processes from *read*ing or *writ*ing any part of an entire file, and record locking is the capability to prevent other processes from *read*ing or *writ*ing particular records (parts of a file between particular byte offsets). Exercise 5.9 explores the implementation of file and record locking.

5.5 ADJUSTING THE POSITION OF FILE I/O — LSEEK

The ordinary use of *read* and *write* system calls provides sequential access to a file, but processes can use the *lseek* system call to position the I/O and allow random access to a file. The syntax for the system call is

```
position = lseek(fd, offset, reference);
```

where *fd* is the file descriptor identifying the file, *offset* is a byte offset, and *reference* indicates whether *offset* should be considered from the beginning of the file, from the current position of the read/write offset, or from the end of the file. The return value, *position*, is the byte offset where the next *read* or *write* will start. In the program in Figure 5.10, for example, a process *open*s a file, *read*s a byte, then invokes *lseek* to advance the file table offset value by 1023 (with *reference* 1), and loops. Thus, the program *read*s every 1024th byte of the file. If the value of *reference* is 0, the kernel seeks from the beginning of the file, and if its value is 2, the kernel seeks beyond the end of the file. The *lseek* system call has nothing to do with the seek operation that positions a disk arm over a particular disk sector. To implement *lseek*, the kernel simply adjusts the offset value in the file table; subsequent *read* or *write* system calls use the file table offset as their starting byte offset.

5.6 CLOSE

A process *close*s an *open* file when it no longer wants to access it. The syntax for the *close* system call is

```
#include <fcntl.h>
main(argc, argv)
        int argc;
        char *argv[];
{
        int fd, skval;
        char c;

        if (argc != 2)
                exit();
        fd = open(argv[1], O_RDONLY);
        if (fd == -1)
                exit();
        while ((skval = read(fd, &c, 1)) == 1)
        {
                printf("char %c\n", c);
                skval = lseek(fd, 1023L, 1);
                printf("new seek val %d\n", skval);
        }
}
```

Figure 5.10. Program with Lseek System Call

close(fd);

where *fd* is the file descriptor for the *open* file. The kernel does the *close* operation by manipulating the file descriptor and the corresponding file table and inode table entries. If the reference count of the file table entry is greater than 1 because of *dup* or *fork* calls, then other user file descriptors reference the file table entry, as will be seen; the kernel decrements the count and the *close* completes. If the file table reference count is 1, the kernel frees the entry and releases the in-core inode originally allocated in the *open* system call (algorithm *iput*). If other processes still reference the inode, the kernel decrements the inode reference count but leaves it allocated; otherwise, the inode is free for reallocation because its reference count is 0. When the *close* system call completes, the user file descriptor table entry is empty. Attempts by the process to use that file descriptor result in an error until the file descriptor is reassigned as a result of another system call. When a process *exit*s, the kernel examines its active user file descriptors and internally *close*s each one. Hence, no process can keep a file open after it terminates.

For example, Figure 5.11 shows the relevant table entries of Figure 5.4, after the second process *close*s its files. The entries for file descriptors 3 and 4 in the user file descriptor table are empty. The count fields of the file table entries are now 0, and the entries are empty. The inode reference count for the files "/etc/passwd" and "private" are also decremented. The inode entry for "private" is on the free list because its reference count is 0, but its entry is not empty. If

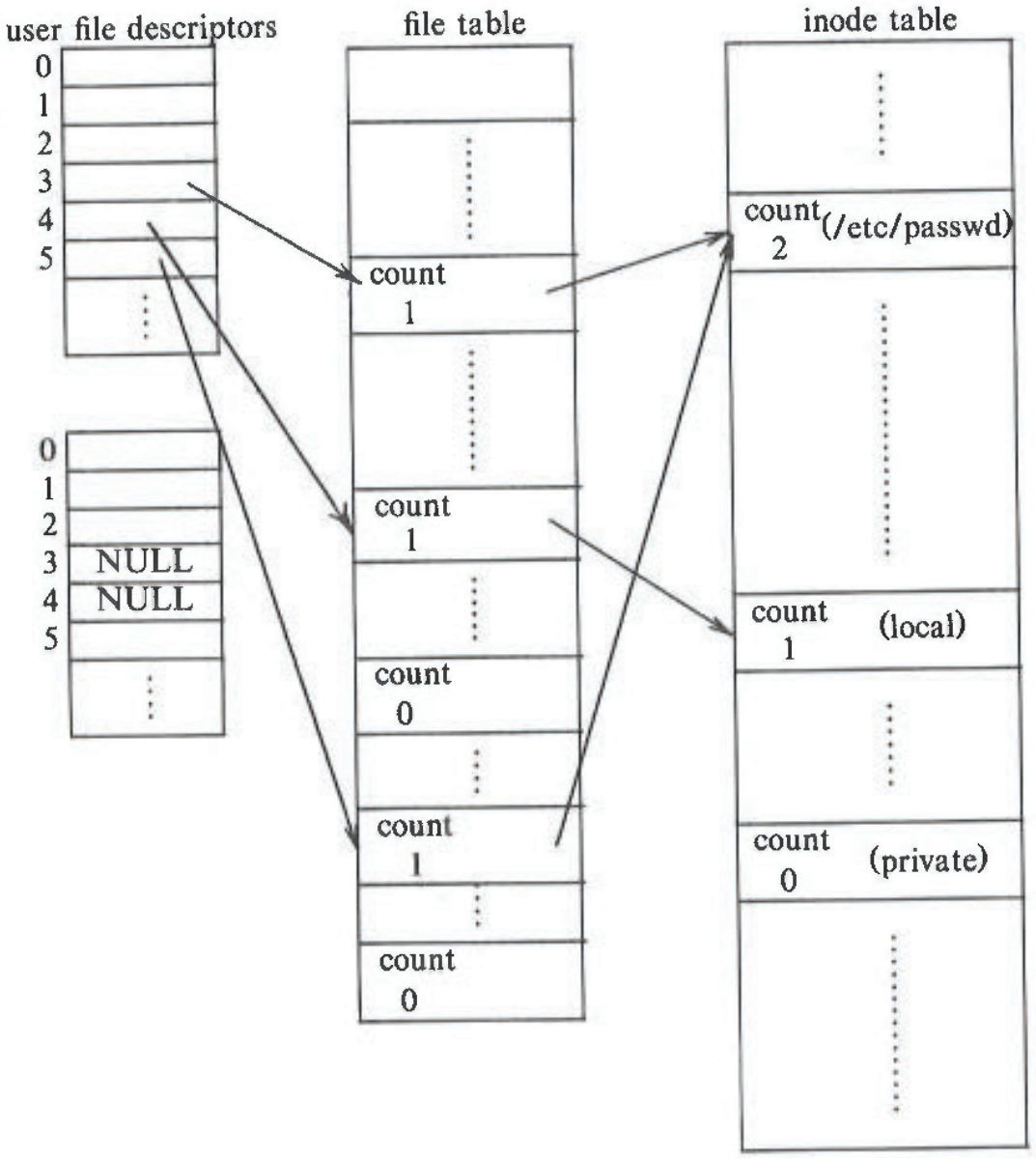


Figure 5.11. Tables after Closing a File

another process accesses the file "private" while the inode is still on the free list, the kernel will reclaim the inode, as explained in Section 4.1.2.

5.7 FILE CREATION

The *open* system call gives a process access to an existing file, but the *creat* system call creates a new file in the system. The syntax for the *creat* system call is

```
fd = creat(pathname, modes);
```

where the variables *pathname*, *modes*, and *fd* mean the same as they do in the *open* system call. If no such file previously existed, the kernel creates a new file with the specified name and permission modes; if the file already existed, the kernel truncates the file (releases all existing data blocks and sets the file size to 0) subject to suitable file access permissions.[3] Figure 5.12 shows the algorithm for file creation.

```
algorithm creat
input:  file name
        permission settings
output: file descriptor
{
        get inode for file name (algorithm namei);
        if (file already exists)
        {
                if (not permitted access)
                {
                        release inode (algorithm iput);
                        return(error);
                }
        }
        else        /* file does not exist yet */
        {
                assign free inode from file system (algorithm ialloc);
                create new directory entry in parent directory: include
                        new file name and newly assigned inode number;
        }
        allocate file table entry for inode, initialize count;
        if (file did exist at time of create)
                free all file blocks (algorithm free);
        unlock(inode);
        return(user file descriptor);
}
```

Figure 5.12. Algorithm for Creating a File

The kernel parses the path name using algorithm *namei*, following the algorithm literally while parsing directory names. However, when it arrives at the last component of the path name, namely, the file name that it will create, *namei*

3. The *open* system call specifies two flags, *O_CREAT* (create) and *O_TRUNC* (truncate): If a process specifies the *O_CREAT* flag on an *open* and the file does not exist, the kernel will create the file. If the file already exists, it will not be truncated unless the *O_TRUNC* flag is also set.

notes the byte offset of the first empty directory slot in the directory and saves the offset in the *u area*. If the kernel does not find the path name component in the directory, it will eventually write the name into the empty slot just found. If the directory has no empty slots, the kernel remembers the offset of the end of the directory and creates a new slot there. It also remembers the inode of the directory being searched in its *u area* and keeps the inode locked; the directory will become the parent directory of the new file. The kernel does not write the new file name into the directory yet, so that it has less to undo in event of later errors. It checks that the directory allows the process write permission: Because a process will write the directory as a result of the *creat* call, write permission for a directory means that processes are allowed to create files in the directory.

Assuming no file by the given name previously existed, the kernel assigns an inode for the new file, using algorithm *ialloc* (Section 4.6). It then writes the new file name component and the inode number of the newly allocated inode in the parent directory, at the byte offset saved in the *u area*. Afterwards, it releases the inode of the parent directory, having held it from the time it searched the directory for the file name. The parent directory now contains the name of the new file and its inode number. The kernel writes the newly allocated inode to disk (algorithm *bwrite*) before it writes the directory with the new name to disk. If the system crashes between the write operations for the inode and the directory, there will be an allocated inode that is not referenced by any path name in the system but the system will function normally. If, on the other hand, the directory were written before the newly allocated inode and the system crashed in the middle, the file system would contain a path name that referred to a bad inode. (See Section 5.16.1 for more detail.)

If the given file already existed before the *creat*, the kernel finds its inode while searching for the file name. The old file must allow write permission for a process to create a "new" file by the same name, because the kernel changes the file contents during the *creat* call: It truncates the file, freeing all its data blocks using algorithm *free*, so that the file looks like a newly created file. However, the owner and permission modes of the file are the same as they were for the original file: The kernel does not reassign ownership to the owner of the process, and it ignores the permission modes specified by the process. Finally, the kernel does not check that the parent directory of the existing file allows write permission, because it will not change the directory contents.

The *creat* system call proceeds according to the same algorithm as the *open* system call. The kernel allocates an entry in the file table for the created file so that the process can *write* the file, allocates an entry in the user file descriptor table, and eventually returns the index to the latter entry as the user file descriptor.

5.8 CREATION OF SPECIAL FILES

The system call *mknod* creates special files in the system, including named pipes, device files, and directories. It is similar to *creat* in that the kernel allocates an

inode for the file. The syntax of the *mknod* system call is

mknod(pathname, type and permissions, dev)

where *pathname* is the name of the node to be created, *type and permissions* give the node type (directory, for example) and access permissions for the new file to be created, and *dev* specifies the major and minor device numbers for block and character special files (Chapter 10). Figure 5.13 depicts the algorithm *mknod* for making a new node.

```
algorithm make new node
inputs:  node (file name)
         file type
         permissions
         major, minor device number (for block, character special files)
output: none
{
        if (new node not named pipe and user not super user)
                return(error);
        get inode of parent of new node (algorithm namei);
        if (new node already exists)
        {
                release parent inode (algorithm iput);
                return(error);
        }
        assign free inode from file system for new node (algorithm ialloc);
        create new directory entry in parent directory: include new node
                name and newly assigned inode number;
        release parent directory inode (algorithm iput);
        if (new node is block or character special file)
                write major, minor numbers into inode structure;
        release new node inode (algorithm iput);
}
```

Figure 5.13. Algorithm for Making New Node

The kernel searches the file system for the file name it is about to create. If the file does not yet exist, the kernel assigns a new inode on the disk and writes the new file name and inode number into the parent directory. It sets the file type field in the inode to indicate that the file type is a pipe, directory or special file. Finally, if the file is a *character special* or *block special* device file, it writes the major and minor device numbers into the inode. If the *mknod* call is creating a directory node, the node will exist after the system call completes but its contents will be in the wrong format (there are no directory entries for "." and ".."). Exercise 5.33 considers the other steps needed to put a directory into the correct format.

```
algorithm change directory
input:   new directory name
output: none
{
        get inode for new directory name (algorithm namei);
        if (inode not that of directory or process not permitted access to file)
        {
                release inode (algorithm iput);
                return(error);
        }
        unlock inode;
        release "old" current directory inode (algorithm iput);
        place new inode into current directory slot in u area;
}
```

Figure 5.14. Algorithm for Changing Current Directory

5.9 CHANGE DIRECTORY AND CHANGE ROOT

When the system is first booted, process 0 makes the file system root its current directory during initialization. It executes the algorithm *iget* on the root inode, saves it in the *u area* as its current directory, and releases the inode lock. When a new process is created via the *fork* system call, the new process inherits the current directory of the old process in its *u area*, and the kernel increments the inode reference count accordingly.

The algorithm *chdir* (Figure 5.14) changes the current directory of a process. The syntax for the *chdir* system call is

```
chdir(pathname);
```

where *pathname* is the directory that becomes the new current directory of the process. The kernel parses the name of the target directory using algorithm *namei* and checks that the target file is a directory and that the process owner has access permission to the directory. It releases the lock to the new inode but keeps the inode allocated and its reference count incremented, releases the inode of the old current directory (algorithm *iput*) stored in the *u area*, and stores the new inode in the *u area*. After a process changes its current directory, algorithm *namei* uses the inode for the start directory to search for all path names that do not begin from root. After execution of the *chdir* system call, the inode reference count of the new directory is at least one, and the inode reference count of the previous current directory may be 0. In this respect, *chdir* is similar to the *open* system call, because both system calls access a file and leave its inode allocated. The inode allocated during the *chdir* system call is released only when the process executes another *chdir* call or when it *exit*s.

A process usually uses the global file system root for all path names starting with "/". The kernel contains a global variable that points to the inode of the global root, allocated by *iget* when the system is booted. Processes can change their notion of the file system root via the *chroot* system call. This is useful if a user wants to simulate the usual file system hierarchy and run processes there. Its syntax is

```
chroot(pathname);
```

where *pathname* is the directory that the kernel subsequently treats as the process's root directory. When executing the *chroot* system call, the kernel follows the same algorithm as for changing the current directory. It stores the new root inode in the process *u area*, unlocking the inode on completion of the system call. However, since the default root for the kernel is stored in a global variable, it does not release the inode of the old root automatically, but only if it or an ancestor process had executed the *chroot* system call. The new inode is now the logical root of the file system for the process (and all its children), meaning that all path name searches in algorithm *namei* that start from root ("/") start from this inode, and that all attempts to use ".." over the root will leave the working directory of the process in the new root. A process bestows new child processes with its changed root, just as it bestows them with its current directory.

5.10 CHANGE OWNER AND CHANGE MODE

Changing the owner or mode (access permissions) of a file are operations on the inode, not on the file per se. The syntax of the calls is

```
chown(pathname, owner, group)
chmod(pathname, mode)
```

To change the owner of a file, the kernel converts the file name to an inode using algorithm *namei*. The process owner must be superuser or match that of the file owner (a process cannot give away something that does not belong to it). The kernel then assigns the new owner and group to the file, clears the set user and set group flags (see Section 7.5), and releases the inode via algorithm *iput*. After the change of ownership, the old owner loses "owner" access rights to the file. To change the mode of a file, the kernel follows a similar procedure, changing the mode flags in the inode instead of the owner numbers.

5.11 STAT AND FSTAT

The system calls *stat* and *fstat* allow processes to query the status of files, returning information such as the file type, file owner, access permissions, file size, number of links, inode number, and file access times. The syntax for the system calls is

```
stat(pathname, statbuffer);
fstat(fd, statbuffer);
```

where *pathname* is a file name, *fd* is a file descriptor returned by a previous *open* call, and *statbuffer* is the address of a data structure in the user process that will contain the status information of the file on completion of the call. The system calls simply write the fields of the inode into *statbuffer*. The program in Figure 5.33 will illustrate the use of *stat* and *fstat*.

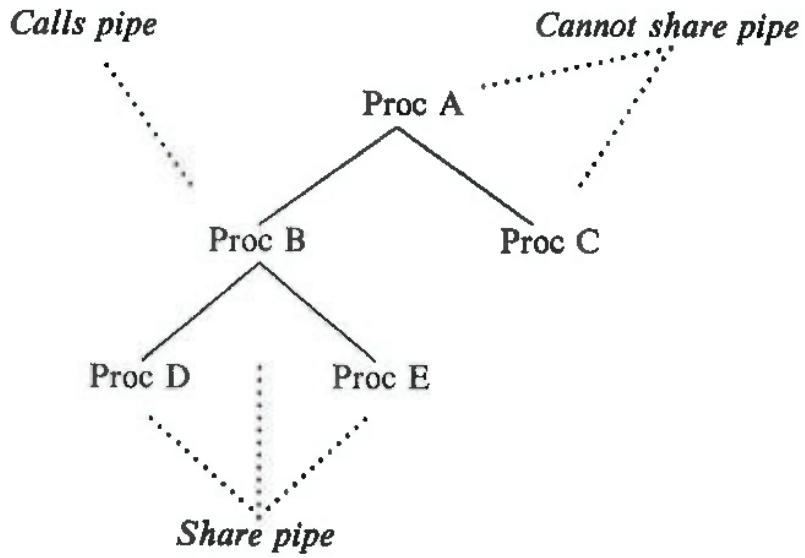


Figure 5.15. Process Tree and Sharing Pipes

5.12 PIPES

Pipes allow transfer of data between processes in a first-in-first-out manner (*FIFO*), and they also allow synchronization of process execution. Their implementation allows processes to communicate even though they do not know what processes are on the other end of the pipe. The traditional implementation of pipes uses the file system for data storage. There are two kinds of pipes: *named pipes* and, for lack of a better term, *unnamed pipes*, which are identical except for the way that a process initially accesses them. Processes use the *open* system call for named pipes, but the *pipe* system call to create an unnamed pipe. Afterwards, processes use the regular system calls for files, such as *read*, *write*, and *close* when manipulating pipes. Only related processes, descendants of a process that issued the *pipe* call, can share access to unnamed pipes. In Figure 5.15 for example, if process B creates a pipe and then spawns processes D and E, the three processes share access to the pipe, but processes A and C do not. However, all processes can access a named pipe regardless of their relationship, subject to the usual file permissions.

Because unnamed pipes are more common, they will be presented first.

5.12.1 The Pipe System Call

The syntax for creation of a pipe is

```
pipe(fdptr);
```

where *fdptr* is the pointer to an integer array that will contain the two file descriptors for *read*ing and *writ*ing the pipe. Because the kernel implements pipes in the file system and because a pipe does not exist before its use, the kernel must assign an inode for it on creation. It also allocates a pair of user file descriptors and corresponding file table entries for the pipe: one file descriptor for *read*ing from the pipe and the other for *writ*ing to the pipe. It uses the file table so that the interface for the *read*, *write* and other system calls is consistent with the interface for regular files. As a result, processes do not have to know whether they are *read*ing or *writ*ing a regular file or a pipe.

```
algorithm pipe
input:  none
output: read file descriptor
        write file descriptor
{
    assign new inode from pipe device (algorithm ialloc);
    allocate file table entry for reading, another for writing;
    initialize file table entries to point to new inode;
    allocate user file descriptor for reading, another for writing,
        initialize to point to respective file table entries;
    set inode reference count to 2;
    initialize count of inode readers, writers to 1;
}
```

Figure 5.16. Algorithm for Creation of (Unnamed) Pipes

Figure 5.16 shows the algorithm for creating unnamed pipes. The kernel assigns an inode for a pipe from a file system designated the *pipe device* using algorithm *ialloc*. A pipe device is just a file system from which the kernel can assign inodes and data blocks for pipes. System administrators specify a pipe device during system configuration, and it may be identical to another file system. While a pipe is active, the kernel cannot reassign the pipe inode and data blocks to another file.

The kernel then allocates two file table entries for the read and write descriptors, respectively, and updates the bookkeeping information in the in-core inode. Each file table entry records how many instances of the pipe are open for reading or writing, initially 1 for each file table entry, and the inode reference

count indicates how many times the pipe was "opened," initially two — one for each file table entry. Finally, the inode records byte offsets in the pipe where the next read or write of the pipe will start. Maintaining the byte offsets in the inode allows convenient FIFO access to the pipe data and differs from regular files where the offset is maintained in the file table. Processes cannot adjust them via the *lseek* system call and so random access I/O to a pipe is not possible.

5.12.2 Opening a Named Pipe

A named pipe is a file whose semantics are the same as those of an unnamed pipe, except that it has a directory entry and is accessed by a path name. Processes *open* named pipes in the same way that they open regular files and, hence, processes that are not closely related can communicate. Named pipes permanently exist in the file system hierarchy (subject to their removal by the *unlink* system call), but unnamed pipes are transient: When all processes finish using the pipe, the kernel reclaims its inode.

The algorithm for opening a named pipe is identical to the algorithm for opening a regular file. However, before completing the system call, the kernel increments the read or write counts in the inode, indicating the number of processes that have the named pipe open for reading or writing. A process that *open*s the named pipe for reading will sleep until another process opens the named pipe for writing, and vice versa. It makes no sense for a pipe to be open for reading if there is no hope for it to receive data; the same is true for writing. Depending on whether the process *open*s the named pipe for reading or writing, the kernel awakens other processes that were asleep, waiting for a writer or reader process (respectively) on the named pipe.

If a process *open*s a named pipe for reading and a writing process exists, the *open* call completes. Or if a process *open*s a named pipe with the *no delay* option, the *open* returns immediately, even if there are no writing processes. But if neither condition is true, the process sleeps until a writer process *open*s the pipe. Similar rules hold for a process that *open*s a pipe for writing.

5.12.3 Reading and Writing Pipes

A pipe should be viewed as if processes *write* into one end of the pipe and *read* from the other end. As mentioned above, processes access data from a pipe in FIFO manner, meaning that the order that data is written into a pipe is the order that it is read from the pipe. The number of processes *read*ing from a pipe do not necessarily equal the number of processes *writ*ing the pipe; if the number of readers or writers is greater than 1, they must coordinate use of the pipe with other mechanisms. The kernel accesses the data for a pipe exactly as it accesses data for a regular file: It stores data on the pipe device and assigns blocks to the pipe as needed during *write* calls. The difference between storage allocation for a pipe and

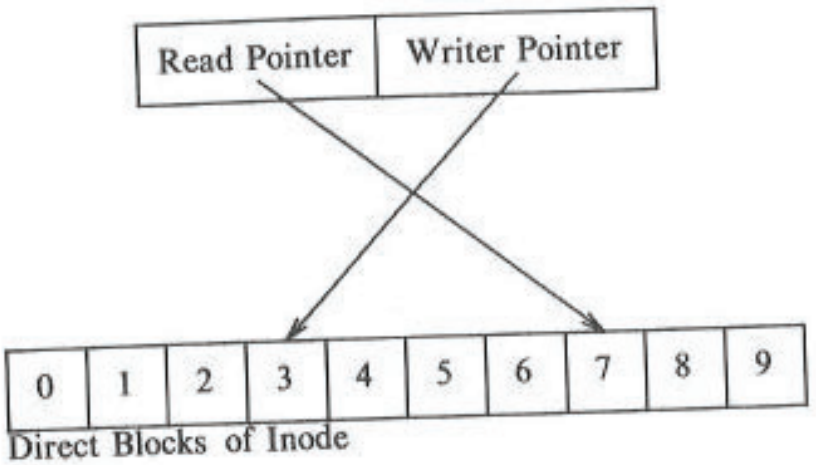


Figure 5.17. Logical View of Reading and Writing a Pipe

a regular file is that a pipe uses only the direct blocks of the inode for greater efficiency, although this places a limit on how much data a pipe can hold at a time. The kernel manipulates the direct blocks of the inode as a circular queue, maintaining read and write pointers internally to preserve the FIFO order (Figure 5.17).

Consider four cases for *read*ing and *writ*ing pipes: *writ*ing a pipe that has room for the data being written, *read*ing from a pipe that contains enough data to satisfy the *read*, *read*ing from a pipe that does not contain enough data to satisfy the *read*, and finally, *writ*ing a pipe that does not have room for the data being written.

Consider first the case that a process is writing a pipe and assume that the pipe has room for the data being written: The sum of the number of bytes being written and the number of bytes already in the pipe is less than or equal to the pipe's capacity. The kernel follows the algorithm for writing a regular file, except that it increments the pipe size automatically after every *write*, since by definition the amount of data in the pipe grows with every *write*. This differs from the growth of a regular file where the process increments the file size only when it *write*s data beyond the current end of file. If the next byte offset in the pipe were to require use of an indirect block, the kernel adjusts the file offset value in the *u area* to point to the beginning of the pipe (byte offset 0). The kernel never overwrites data in the pipe; it can reset the byte offset to 0 because it has already determined that the data will not overflow the pipe's capacity. When the writer process has written all its data into the pipe, the kernel updates the pipe's (inode) write pointer so that the next process to *write* the pipe will proceed from where the last *write* stopped. The kernel then awakens all other processes that fell asleep waiting to read data from the pipe.

When a process *read*s a pipe, it checks if the pipe is empty or not. If the pipe contains data, the kernel *read*s the data from the pipe as if the pipe were a regular file, following the regular algorithm for *read*. However, its initial offset is the pipe

read pointer stored in the inode, indicating the extent of the previous *read*. After *read*ing each block, the kernel decrements the size of the pipe according to the number of bytes it read, and it adjusts the *u area* offset value to wrap around to the beginning of the pipe, if necessary. When the *read* system call completes, the kernel awakens all sleeping writer processes and saves the current read offset in the inode (not in the file table entry).

If a process attempts to *read* more data than is in the pipe, the *read* will complete successfully after returning all data currently in the pipe, even though it does not satisfy the user count. If the pipe is empty, the process will typically sleep until another process *write*s data into the pipe, at which time all sleeping processes that were waiting for data wake up and race to *read* the pipe. If, however, a process *open*s a named pipe with the *no delay* option, it will return immediately from a *read* if the pipe contains no data. The semantics of reading and writing pipes are similar to the semantics of reading and writing terminal devices (Chapter 10), allowing programs to ignore the type of file they are dealing with.

If a process *write*s a pipe and the pipe cannot hold all the data, the kernel marks the inode and goes to sleep waiting for data to drain from the pipe. When another process subsequently *read*s from the pipe, the kernel will notice that processes are asleep waiting for data to drain from the pipe, and it will awaken them, as explained above. The exception to this statement is when a process *write*s an amount of data greater than the pipe capacity (that is, the amount of data that can be stored in the inode direct blocks); here, the kernel *write*s as much data as possible to the pipe and puts the process to sleep until more room becomes available. Thus, it is possible that written data will not be contiguous in the pipe if other processes write their data to the pipe before this process resumes its write.

Analyzing the implementation of pipes, the process interface is consistent with that of regular files, but the implementation differs because the kernel stores the read and write offsets in the inode instead of in the file table. The kernel must store the offsets in the inode for named pipes so that processes can share their values: They cannot share values stored in file table entries because a process gets a new file table entry for each *open* call. However, the sharing of read and write offsets in the inode predates the implementation of named pipes. Processes with access to unnamed pipes share access to the pipe through common file table entries, so they could conceivably store the read and write offsets in the file table entry, as is done for regular files. This was not done, because the low-level routines in the kernel no longer have access to the file table entry: The code is simpler because the processes share offsets stored in the inode.

5.12.4 Closing Pipes

When closing a pipe, a process follows the same procedure it would follow for closing a regular file, except that the kernel does special processing before releasing the pipe's inode. The kernel decrements the number of pipe readers or writers, according to the type of the file descriptor. If the count of writer processes drops to

0 and there are processes asleep waiting to read data from the pipe, the kernel awakens them, and they return from their *read* calls without reading any data. If the count of reader processes drops to 0 and there are processes asleep waiting to write data to the pipe, the kernel awakens them and sends them a signal (Chapter 7) to indicate an error condition. In both cases, it makes no sense to allow the processes to continue sleeping when there is no hope that the state of the pipe will ever change. For example, if a process is waiting to read an unnamed pipe and there are no more writer processes, there will never be a writer process. Although it *is* possible to get new reader or writer processes for named pipes, the kernel treats them consistently with unnamed pipes. If no reader or writer processes access the pipe, the kernel frees all its data blocks and adjusts the inode to indicate that the pipe is empty. When it releases the inode of an ordinary pipe, it frees the disk copy for reassignment.

```
char string[] = "hello";
main()
{
        char buf[1024];
        char *cp1, *cp2;
        int fds[2];

        cp1 = string;
        cp2 = buf;
        while (*cp1)
                *cp2++ = *cp1++;
        pipe(fds);
        for (;;)
        {
                write(fds[1], buf, 6);
                read(fds[0], buf, 6);
        }
}
```

Figure 5.18. Reading and Writing a Pipe

5.12.5 Examples

The program in Figure 5.18 illustrates an artificial use of pipes. The process creates a pipe and goes into an infinite loop, *writ*ing the string "hello" to the pipe and *read*ing it from the pipe. The kernel does not know nor does it care that the process that writes the pipe is the same process that reads the pipe.

A process executing the program in Figure 5.19 creates a named pipe node called "fifo". If invoked with a second (dummy) argument, it continually *writes*

```
#include <fcntl.h>
char string[] = "hello";
main(argc, argv)
        int argc;
        char *argv[];
{
        int fd;
        char buf[256];

        /* create named pipe with read/write permission for all users */
        mknod("fifo", 010777, 0);
        if (argc == 2)
                fd = open("fifo", O_WRONLY);
        else
                fd = open("fifo", O_RDONLY);
        for (;;)
                if (argc == 2)
                        write(fd, string, 6);
                else
                        read(fd, buf, 6);
}
```

Figure 5.19. Reading and Writing a Named Pipe

the string "hello" into the pipe; if invoked without a second argument, it *reads* the named pipe. The two processes are invocations of the identical program and have secretly agreed to communicate through the named pipe "fifo", but they need not be related. Other users could execute the program and participate in (or interfere with) the conversation.

5.13 DUP

The *dup* system call copies a file descriptor into the first free slot of the user file descriptor table, returning the new file descriptor to the user. It works for all file types. The syntax of the system call is

```
newfd = dup(fd);
```

where *fd* is the file descriptor being *dup*ed and *newfd* is the new file descriptor that references the file. Because *dup* duplicates the file descriptor, it increments the count of the corresponding file table entry, which now has one more file descriptor entry that points to it. For example, examination of the data structures depicted in Figure 5.20 indicates that the process did the following sequence of system calls: It *open*ed the file "/etc/passwd" (file descriptor 3), then *open*ed the file "local" (file descriptor 4), *open*ed the file "/etc/passwd" again (file descriptor 5), and finally,

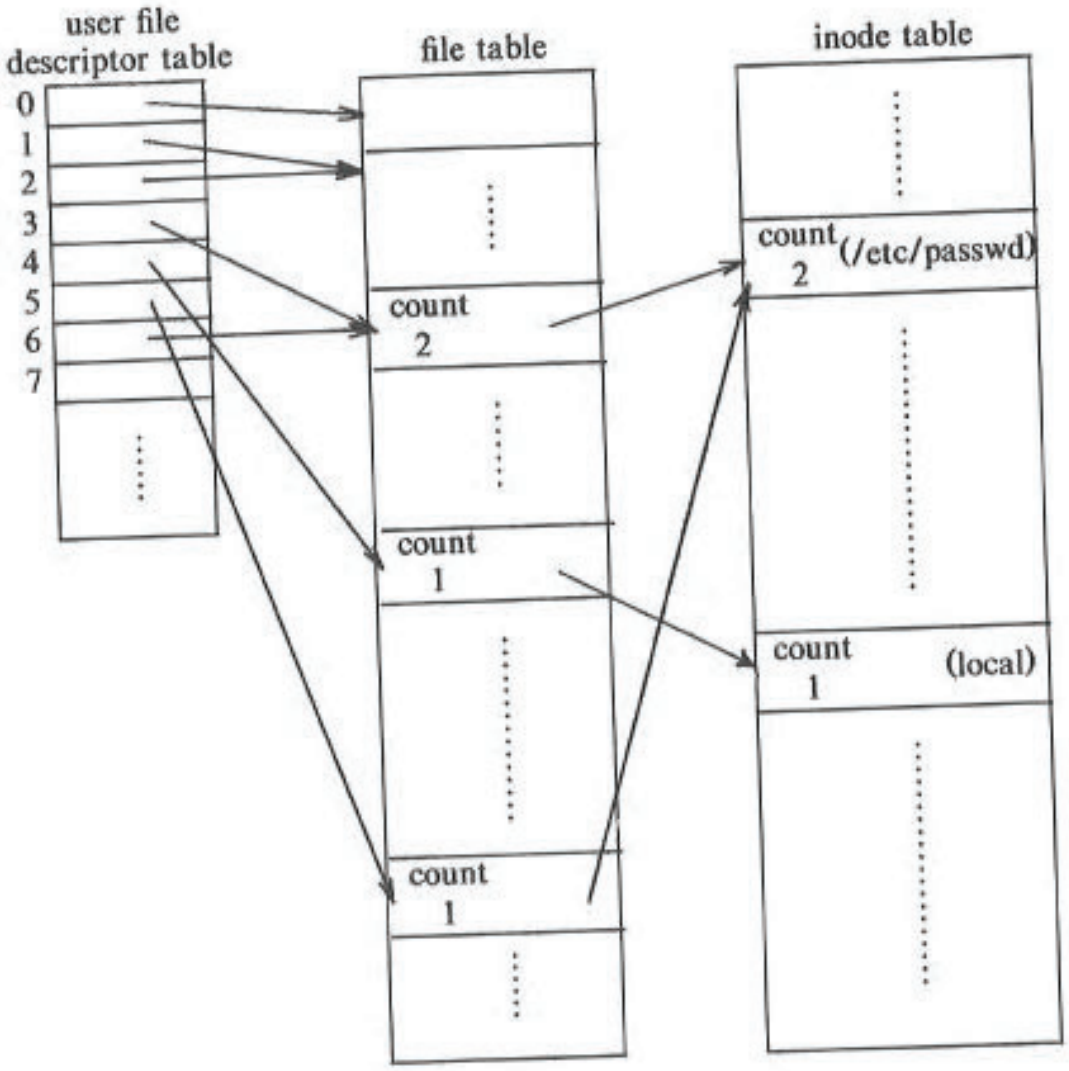


Figure 5.20. Data Structures after Dup

*dup*ed file descriptor 3, returning file descriptor 6.

Dup is perhaps an inelegant system call, because it assumes that the user knows that the system will return the lowest-numbered free entry in the user file descriptor table. However, it serves an important purpose in building sophisticated programs from simpler, building-block programs, as exemplified in the construction of shell pipelines (Chapter 7).

Consider the program in Figure 5.21. The variable *i* contains the file descriptor that the system returns as a result of opening the file "etc/passwd," and the variable *j* contains the file descriptor that the system returns as a result of *dup*ing the file descriptor *i*. In the *u area* of the process, the two user file descriptor entries represented by the user variables *i* and *j* point to one file table entry and therefore use the same file offset. The first two *read*s in the process thus read the data in sequence, and the two buffers, *buf1* and *buf2*, do not contain the same data.

```
#include <fcntl.h>
main()
{
        int i, j;
        char buf1[512], buf2[512];

        i = open("/etc/passwd", O_RDONLY);
        j = dup(i);
        read(i, buf1, sizeof(buf1));
        read(j, buf2, sizeof(buf2));
        close(i);
        read(j, buf2, sizeof(buf2));
}
```

Figure 5.21. C Program Illustrating Dup

This differs from the case where a process *open*s the same file twice and *read*s the same data twice (Section 5.2). A process can *close* either file descriptor if it wants, but I/O continues normally on the other file descriptor, as illustrated in the example. In particular, a process can *close* its standard output file descriptor (file descriptor 1), *dup* another file descriptor so that it becomes file descriptor 1, then treat the file as its standard output. Chapter 7 presents a more realistic example of the use of *pipe* and *dup* when it describes the implementation of the shell.

5.14 MOUNTING AND UNMOUNTING FILE SYSTEMS

A physical disk unit consists of several logical sections, partitioned by the disk driver, and each section has a device file name. Processes can access data in a section by *open*ing the appropriate device file name and then *read*ing and *writ*ing the "file," treating it as a sequence of disk blocks. Chapter 10 gives details on this interface. A section of a disk may contain a logical file system, consisting of a boot block, super block, inode list, and data blocks, as described in Chapter 2. The *mount* system call connects the file system in a specified section of a disk to the existing file system hierarchy, and the *umount* system call disconnects a file system from the hierarchy. The *mount* system call thus allows users to access data in a disk section as a file system instead of a sequence of disk blocks.

The syntax for the *mount* system call is

mount(special pathname, directory pathname, options);

where *special pathname* is the name of the device special file of the disk section containing the file system to be mounted, *directory pathname* is the directory in the existing hierarchy where the file system will be mounted (called the *mount point*), and *options* indicate whether the file system should be mounted "read-only"

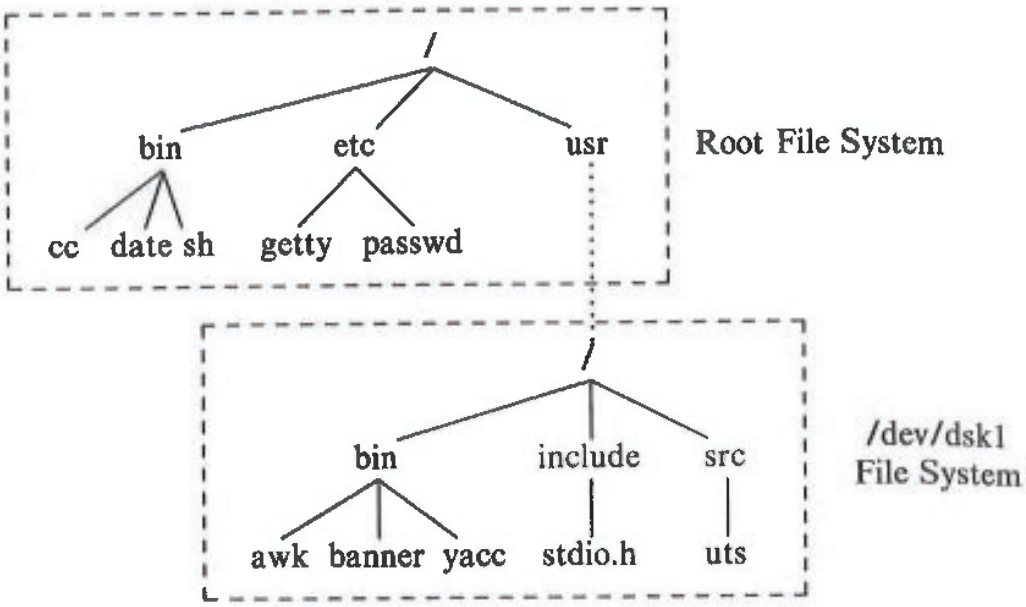


Figure 5.22. File System Tree Before and After Mount

(system calls such as *write* and *creat* that write the file system will fail). For example, if a process issues the system call

```
mount("/dev/dsk1", "/usr", 0);
```

the kernel attaches the file system contained in the portion of the disk called "/dev/dsk1" to directory "/usr" in the existing file system tree (see Figure 5.22). The file "/dev/dsk1" is a block special file, meaning that it is the name of a block device, typically a portion of a disk. The kernel assumes that the indicated portion of the disk contains a file system with a super block, inode list, and root inode. After completion of the *mount* system call, the root of the mounted file system is accessed by the name "/usr". Processes can access files on the mounted file system and ignore the fact that it is detachable. Only the *link* system call checks the file system of a file, because System V does not allow file links to span multiple file systems (see Section 5.15).

The kernel has a *mount table* with entries for every mounted file system. Each mount table entry contains

- a device number that identifies the mounted file system (this is the logical file system number mentioned previously);
- a pointer to a buffer containing the file system super block;
- a pointer to the root inode of the mounted file system ("/" of the "/dev/dsk1" file system in Figure 5.22);
- a pointer to the inode of the directory that is the mount point ("usr" of the root file system in Figure 5.22).

Association of the mount point inode and the root inode of the mounted file system, set up during the *mount* system call, allows the kernel to traverse the file system hierarchy gracefully, without special user knowledge.

```
algorithm mount
inputs:  file name of block special file
         directory name of mount point
         options (read only)
output: none
{
        if (not super user)
                return(error);
        get inode for block special file (algorithm namei);
        make legality checks;
        get inode for "mounted on" directory name (algorithm namei);
        if (not directory, or reference count > 1)
        {
                release inodes (algorithm iput);
                return(error);
        }
        find empty slot in mount table;
        invoke block device driver open routine;
        get free buffer from buffer cache;
        read super block into free buffer;
        initialize super block fields;
        get root inode of mounted device (algorithm iget), save in mount table;
        mark inode of "mounted on" directory as mount point;
        release special file inode (algorithm iput);
        unlock inode of mount point directory;
}
```

Figure 5.23. Algorithm for Mounting a File System

Figure 5.23 depicts the algorithm for mounting a file system. The kernel only allows processes owned by a superuser to *mount* or *umount* file systems. Yielding permission for *mount* and *umount* to the entire user community would allow malicious (or not so malicious) users to wreak havoc on the file system. Superusers should wreak havoc only by accident.

The kernel finds the inode of the special file that represents the file system to be mounted, extracts the major and minor numbers that identify the appropriate disk section, and finds the inode of the directory on which the file system will be mounted. The reference count of the directory inode must not be greater than 1 (it must be at least 1 — why?), because of potentially dangerous side effects (see exercise 5.27). The kernel then allocates a free slot in the mount table, marks the slot in use, and assigns the device number field in the mount table. The above

assignments are done immediately because the calling process could go to sleep in the ensuing device *open* procedure or in reading the file system super block, and another process could attempt to *mount* a file system. By having marked the mount table entry in use, the kernel prevents two *mount*s from using the same entry. By noting the device number of the attempted *mount*, the kernel can prevent other processes from *mount*ing the same file system again, because strange things could happen if a double mount were allowed (see exercise 5.26).

The kernel calls the *open* procedure for the block device containing the file system in the same way it invokes the procedure when opening the block device directly (Chapter 10). The device *open* procedure typically checks that the device is legal, sometimes initializing driver data structures and sending initialization commands to the hardware. The kernel then allocates a free buffer from the buffer pool (a variation of algorithm *getblk*) to hold the super block of the mounted file system and reads the super block using a variation of algorithm *read*. The kernel stores a pointer to the inode of the mounted-on directory of the original file tree to allow file path names containing ".." to traverse the mount point, as will be seen. It finds the root inode of the *mount*ed file system and stores a pointer to the inode in the mount table. To the user, the mounted-on directory and the root of the mounted file system are logically equivalent, and the kernel establishes their equivalence by their coexistence in the mount table entry. Processes can no longer access the inode of the mounted-on directory.

The kernel initializes fields in the file system super block, clearing the lock fields for the free block list and free inode list and setting the number of free inodes in the super block to 0. The purpose of the initializations is to minimize the danger of file system corruption when mounting the file system after a system crash: Making the kernel think that there are no free inodes in the super block forces algorithm *ialloc* to search the disk for free inodes. Unfortunately, if the linked list of free disk blocks is corrupt, the kernel does not fix the list internally (see Section 5.17 for file system maintenance). If the user *mount*s the file system *read-only* to disallow all write operations to the file system, the kernel sets a flag in the super block. Finally, the kernel marks the mounted-on inode as a mount point, so other processes can later identify it. Figure 5.24 depicts the various data structures at the conclusion of the *mount* call.

5.14.1 Crossing Mount Points in File Path Names

Let us reconsider algorithms *namei* and *iget* for the cases where a path name crosses a mount point. The two cases for crossing a mount point are: crossing from the mounted-on file system to the mounted file system (in the direction from the global system root towards a leaf node) and crossing from the mounted file system to the mounted-on file system. The following sequence of shell commands illustrates the two cases.

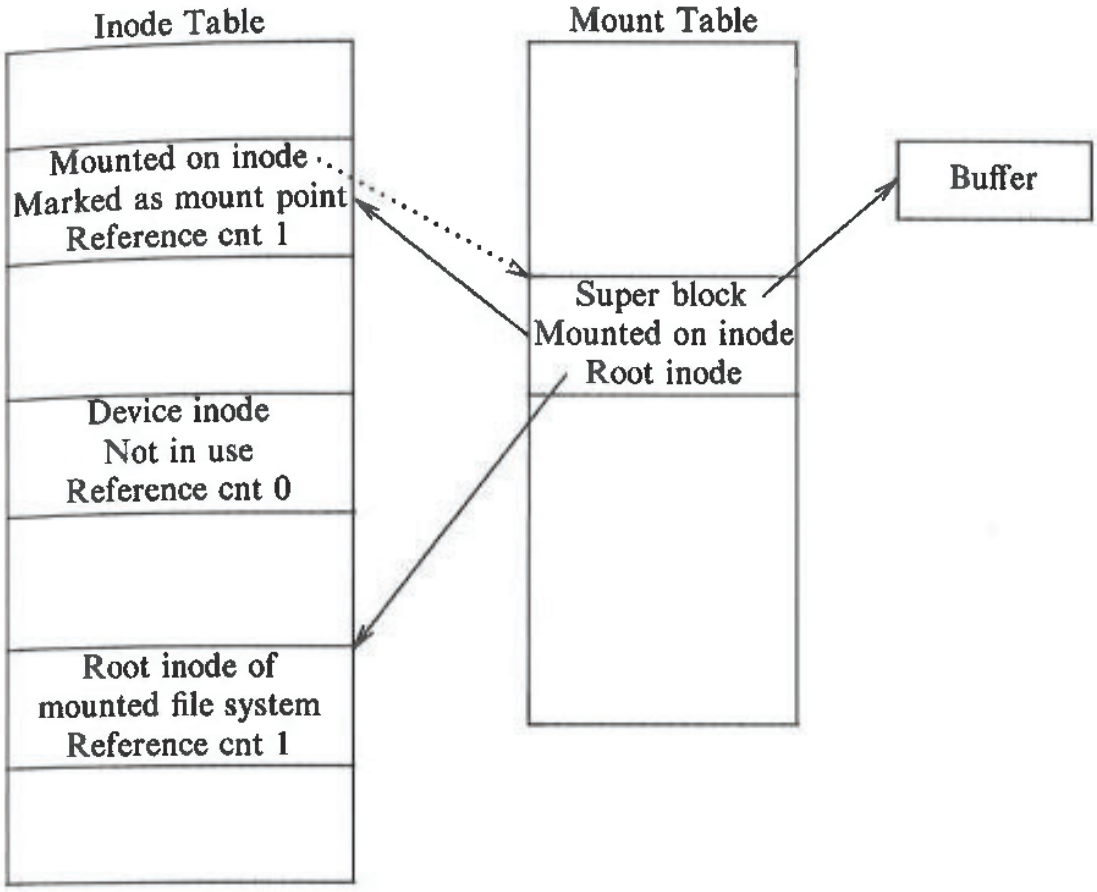


Figure 5.24. Data Structures after Mount

```
mount /dev/dsk1 /usr
cd /usr/src/uts
cd ../../..
```

The *mount* command invokes the *mount* system call after doing some consistency checks and mounts the file system in the disk section identified by "/dev/dsk1" onto the directory "/usr". The first *cd* (change directory) command causes the shell to execute the *chdir* system call, and the kernel parses the path name, crossing the mount point at "/usr". The second *cd* command results in the kernel parsing the path name and crossing the mount point at the third ".." in the path name.

For the case of crossing the mount point from the mounted-on file system to the mounted file system, consider the revised algorithm for *iget* in Figure 5.25, which is identical to that of Figure 4.3, except that it checks if the inode is a mount point: If the inode is marked "mounted-on," the kernel knows that it is a mount point. It finds the mount table entry whose mounted-on inode is the one just accessed and notes the device number of the mounted file system. Using the device number and the inode number for root, which is common to all file systems, it then accesses the

```
algorithm iget
input:   file system inode number
output: locked inode
{
        while (not done)
        {
                if (inode in inode cache)
                {
                        if (inode locked)
                        {
                                sleep (event inode becomes unlocked);
                                continue;       /* loop */
                        }
                        /* special processing for mount points---*/
                        if (inode a mount point)
                        {
                                find mount table entry for mount point;
                                get new file system number from mount table;
                                use root inode number in search;
                                continue;       /* loop again */
                        }
                        if (inode on inode free list)
                                remove from free list;
                        increment inode reference count;
                        return (inode);
                }

                /* inode not in inode cache */
                remove new inode from free list;
                reset inode number and file system;
                remove inode from old hash queue, place on new one;
                read inode from disk (algorithm bread);
                initialize inode (e.g. reference count to 1);
                return inode;
        }
}
```

Figure 5.25. Revised Algorithm for Accessing an Inode

root inode of the mounted device and returns that inode. In the first change directory example above, the kernel first accesses the inode for "/usr" in the mounted-on file system, finds that the inode is marked "mounted-on," finds the root inode of the mounted file system in the mount table, and accesses the root inode of the mounted file system.

```
algorithm namei                 /* convert path name to inode */
input:   path name
output: locked inode
{
        if (path name starts from root)
                working inode = root inode (algorithm iget);
        else
                working inode = current directory inode (algorithm iget);
        while (there is more path name)
        {
                read next path name component from input;
                verify that inode is of directory, permissions;
                if (inode is of changed root and component is "..")
                        continue;       /* loop */
    component search:
                read inode (directory) (algorithms bmap, bread, brelse);
                if (component matches a directory entry)
                {
                        get inode number for matched component;
                        if (found inode of root and working inode is root and
                                        and component name is "..")
                        {
                                /* crossing mount point */
                                get mount table entry for working inode;
                                release working inode (algorithm iput);
                                working inode = mounted on inode;
                                lock mounted on inode;
                                increment reference count of working inode;
                                go to component search (for "..");
                        }
                        release working inode (algorithm iput);
                        working inode = inode for new inode number (algorithm iget);
                }
                else        /* component not in directory */
                        return (no inode);
        }
        return (working inode);
}
```

Figure 5.26. Revised Algorithm for Parsing a File Name

For the second case of crossing the mount point from the mounted file system to the mounted-on file system, consider the revised algorithm for *namei* in Figure 5.26. It is similar to that of Figure 4.11. However, after finding the inode number for a path name component in a directory, the kernel checks if the inode number is the root inode of a file system. If it is, and if the inode of the current working inode is

also root, and the path name component is dot-dot (".."), the kernel identifies the inode as a mount point. It finds the mount table entry whose device number equals the device number of the last found inode, gets the inode of the mounted-on directory, and continues its search for dot-dot ("..") using the mounted-on inode as the working inode. At the root of the file system, however, ".." is the root.

In the example above (cd "../../.."), assume the starting current directory of the process is "/usr/src/uts". When parsing the path name in *namei*, the starting working inode is the current directory. The kernel changes the working inode to that of "/usr/src" as a result of parsing the first ".." in the path name. Then, it parses the second ".." in the path name, finds the root inode of the (previously) mounted file system, "usr", and makes it the working inode in *namei*. Finally, it parses the third ".." in the path name: It finds that the inode number for ".." is the root inode number, its working inode is the root inode, and ".." is the current path name component. The kernel finds the mount table entry for the "usr" mount point, releases the current working inode (the root of the "usr" file system), and allocates the mounted-on inode (the inode for directory "usr" in the root file system) as the new working inode. It then searches the directory structures in the mounted-on "/usr" for ".." and finds the inode number for the root of the file system ("/"). The *chdir* system call then completes as usual; the calling process is oblivious to the fact that it crossed a mount point.

5.14.2 Unmounting a File System

The syntax for the *umount* system call is

```
umount(special filename);
```

where *special filename* indicates the file system to be unmounted. When unmounting a file system (Figure 5.27), the kernel accesses the inode of the device to be unmounted, retrieves the device number for the special file, releases the inode (algorithm *iput*), and finds the mount table entry whose device number equals that of the special file. Before the kernel actually unmounts a file system, it makes sure that no files on that file system are still in use by searching the inode table for all files whose device number equals that of the file system being unmounted. Active files have a positive reference count and include files that are the current directory of some process, files with shared text that are currently being executed (Chapter 7), and open files that have not been closed. If any files from the file system are active, the *umount* call fails: if it were to succeed, the active files would be inaccessible.

The buffer pool may still contain "delayed write" blocks that were not written to disk, so the kernel flushes them from the buffer pool. The kernel removes shared text entries that are in the region table but not operational (see Chapter 7 for detail), writes out all recently modified super blocks to disk, and updates the disk copy of all inodes that need updating. It would suffice for the kernel to update the disk blocks, super block, and inodes for the unmounting file system only, but for

```
algorithm umount
input:  special file name of file system to be unmounted
output: none
{
        if (not super user)
                return(error);
        get inode of special file (algorithm namei);
        extract major, minor number of device being unmounted;
        get mount table entry, based on major, minor number,
                for unmounting file system;
        release inode of special file (algorithm iput);
        remove shared text entries from region table for files
                        belonging to file system; /* chap 7xxx */
        update super block, inodes, flush buffers;
        if (files from file system still in use)
                return(error);
        get root inode of mounted file system from mount table;
        lock inode;
        release inode (algorithm iput);  /* iget was in mount */
        invoke close routine for special device;
        invalidate buffers in pool from unmounted file system;
        get inode of mount point from mount table;
        lock inode;
        clear flag marking it as mount point;
        release inode (algorithm iput);          /* iget in mount */
        free buffer used for super block;
        free mount table slot;
}
```

Figure 5.27. Algorithm for Unmounting a File System

historical reasons it does so for all file systems. The kernel then releases the root inode of the mounted file system, held since its original access during the *mount* system call, and invokes the driver of the device that contains the file system to close the device. Afterwards, it goes through the buffers in the buffer cache and invalidates buffers for blocks on the now unmounted file system; there is no need to cache data in those blocks any longer. When invalidating the buffers, it moves the buffers to the beginning of the buffer free list, so that valid blocks remain in the buffer cache longer. It clears the "mounted-on" flag in the mounted-on inode set during the *mount* call and releases the inode. After marking the mount table entry free for general use, the *umount* call completes.

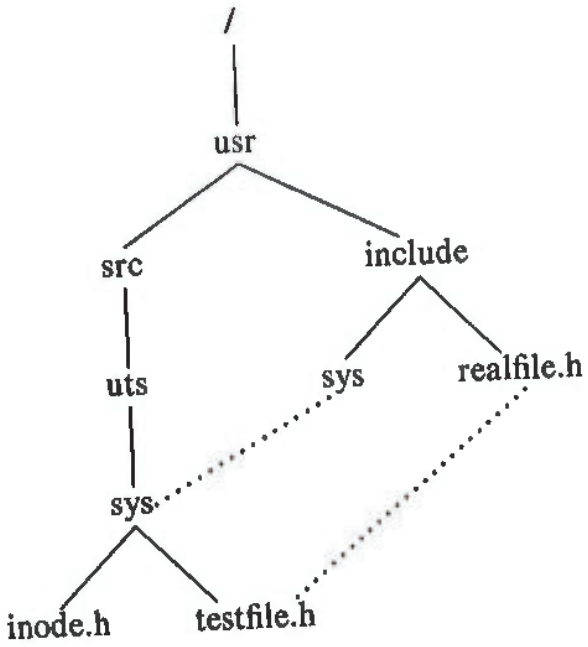


Figure 5.28. Linked Files in File System Tree

5.15 LINK

The *link* system call links a file to a new name in the file system directory structure, creating a new directory entry for an existing inode. The syntax for the *link* system call is

```
link(source file name, target file name);
```

where *source file name* is the name of an existing file and *target file name* is the new (additional) name the file will have after completion of the *link* call. The file system contains a path name for each link the file has, and processes can access the file by any of the path names. The kernel does not know which name was the original file name, so no file name is treated specially. For example, after executing the system calls

```
link("/usr/src/uts/sys", "/usr/include/sys");
link("/usr/include/realfile.h", "/usr/src/uts/sys/testfile.h");
```

the following three path names refer to the same file: "/usr/src/uts/sys/testfile.h", "/usr/include/sys/testfile.h", and "/usr/include/realfile" (see Figure 5.28).

The kernel allows only a superuser to *link* directories, simplifying the coding of programs that traverse the file system tree. If arbitrary users could *link* directories, programs designed to traverse the file hierarchy would have to worry about getting into an infinite loop if a user were to *link* a directory to a node name below it in the hierarchy. Superusers are presumably more careful about making such *links*. The capability to link directories had to be supported on early versions of the

system, because the implementation of the *mkdir* command, which creates a new directory, relies on the capability to link directories. Inclusion of the *mkdir* system call eliminates the need to link directories.

```
algorithm link
input:   existing file name
         new file name
output: none
{
        get inode for existing file name (algorithm namei);
        if (too many links on file or linking directory without super user permission)
        {
                release inode (algorithm iput);
                return(error);
        }
        increment link count on inode;
        update disk copy of inode;
        unlock inode;
        get parent inode for directory to contain new file name (algorithm namei);
        if (new file name already exists or existing file, new file on
                                        different file systems)
        {
                undo update done above;
                return(error);
        }
        create new directory entry in parent directory of new file name:
                include new file name, inode number of existing file name;
        release parent directory inode (algorithm iput);
        release inode of existing file (algorithm iput);
}
```

Figure 5.29. Algorithm for Linking Files

Figure 5.29 shows the algorithm for *link*. The kernel first locates the inode for the source file using algorithm *namei*, increments its link count, updates the disk copy of the inode (for consistency, as will be seen), and unlocks the inode. It then searches for the target file; if the file is present, the *link* call fails, and the kernel decrements the link count incremented earlier. Otherwise, it notes the location of an empty slot in the parent directory of the target file, writes the target file name and the source file inode number into that slot, and releases the inode of the target file parent directory via algorithm *iput*. Since the target file did not originally exist, there is no other inode to release. The kernel concludes by releasing the source file inode: Its link count is 1 greater than it was at the beginning of the call, and another name in the file system allows access to it. The link count keeps count of the directory entries that refer to the file and is thus distinct from the inode

reference count. If no other processes access the file at the conclusion of the *link* call, the inode reference count of the file is 0, and the link count of the file is at least 2.

For example, when executing

```
link("source", "dir/target");
```

the kernel locates the inode for file "source", increments its link count, remembers its inode number, say 74, and unlocks the inode. It locates the inode of "dir", the parent directory of "target", finds an empty directory slot in "dir", and writes the file name "target" and the inode number 74 into the empty directory slot. Finally, it releases the inode for "source" via algorithm *iput*. If the link count of "source" had been 1, it is now 2.

Two deadlock possibilities are worthy of note, both concerning the reason the process unlocks the source file inode after incrementing its link count. If the kernel did not unlock the inode, two processes could deadlock by executing the following system calls simultaneously.

```
process A:    link("a/b/c/d", "e/f/g");
process B:    link("e/f", "a/b/c/d/ee");
```

Suppose process A finds the inode for file "a/b/c/d" at the same time that process B finds the inode for "e/f". The phrase *at the same time* means that the system arrives at a state where each process has allocated its inode. Figure 5.30 illustrates an execution scenario. When process A now attempts to find the inode for directory "e/f", it would sleep awaiting the event that the inode for "f" becomes free. But when process B attempts to find the inode for directory "a/b/c/d", it would sleep awaiting the event that the inode for "d" becomes free. Process A would be holding a locked inode that process B wants, and process B would be holding a locked inode that process A wants. The kernel avoids this classic example of deadlock by releasing the source file's inode after incrementing its link count. Since the first resource (inode) is free when accessing the next resource, no deadlock can occur.

The last example showed how two processes could deadlock each other if the inode lock were not released. A single process could also deadlock itself. If it executed

```
link("a/b/c", "a/b/c/d");
```

it would allocate the inode for file "c" in the first part of the algorithm; if the kernel did not release the inode lock, it would deadlock when encountering the inode "c" in searching for the file "d". If two processes, or even one process, could not continue executing because of deadlock, what would be the effect on the system? Since inodes are finitely allocatable resources, receipt of a signal cannot awaken the process from its sleep (Chapter 7). Hence, the system could not break the deadlock without rebooting. If no other processes accessed the files over which the processes deadlock, no other processes in the system would be affected.

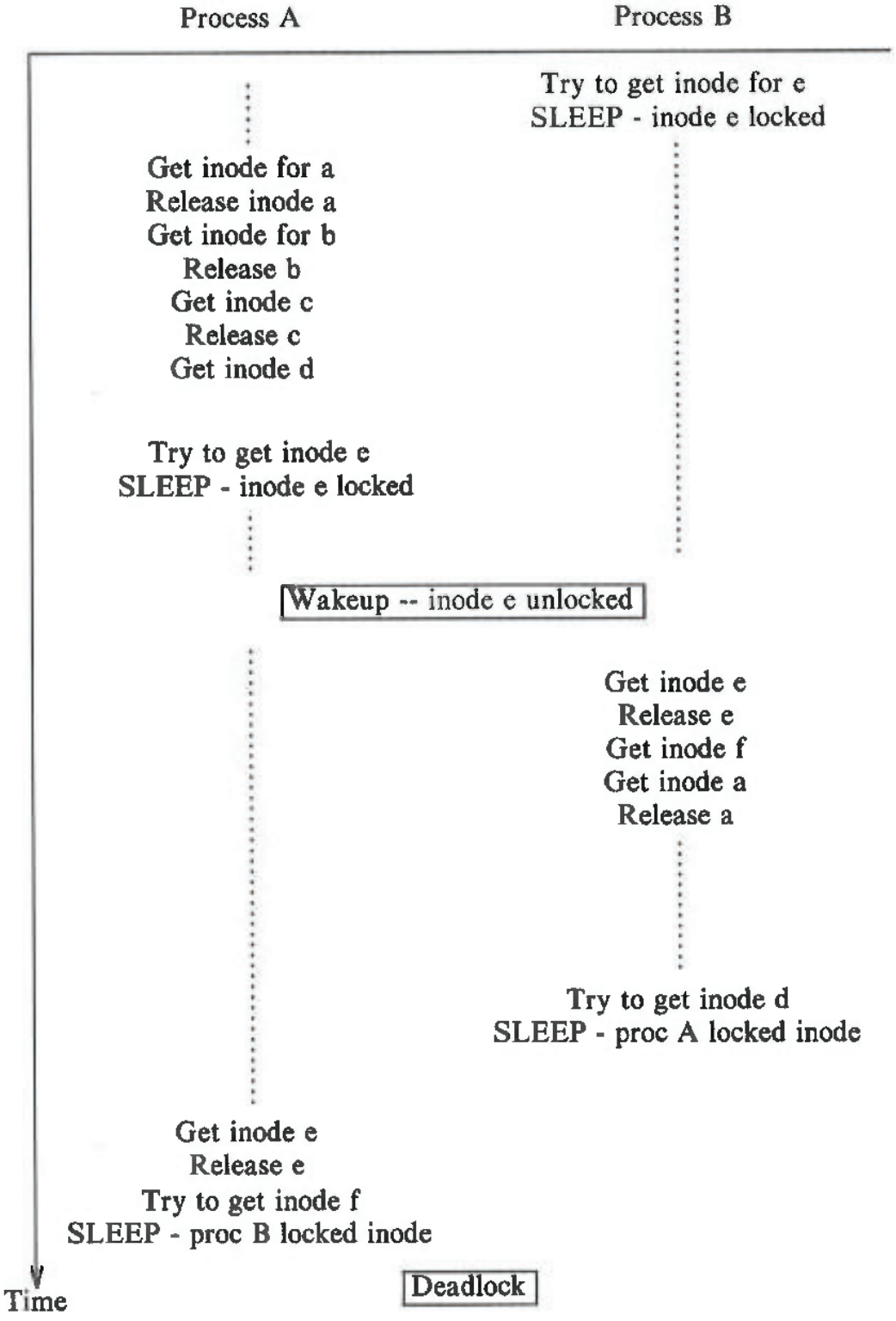


Figure 5.30. Deadlock Scenario for Link

However, any processes that accessed those files (or attempted to access other files via the locked directory) would deadlock. Thus, if the file were "/bin" or "/usr/bin" (typical depositories for commands) or "/bin/sh" (the shell) the effect on the system would be disastrous.

5.16 UNLINK

The *unlink* system call removes a directory entry for a file. The syntax for the *unlink* call is

```
unlink(pathname);
```

where *pathname* identifies the name of the file to be *unlink*ed from the directory hierarchy. If a process *unlink*s a given file, no file is accessible by that name until another directory entry with that name is created. In the following code fragment, for example,

```
unlink("myfile");
fd = open("myfile", O_RDONLY);
```

the *open* call should fail, because the current directory no longer contains a file called *myfile*. If the file being *unlink*ed is the last link of the file, the kernel eventually frees its data blocks. However, if the file had several links, it is still accessible by its other names.

Figure 5.31 gives the algorithm for *unlink*ing a file. The kernel first uses a variation of algorithm *namei* to find the file that it must *unlink*, but instead of returning its inode, it returns the inode of the parent directory. It accesses the in-core inode of the file to be *unlink*ed, using algorithm *iget*. (The special case for unlinking the file "." is covered in an exercise.) After checking error conditions and, for executable files, removing inactive shared text entries from the region table (Chapter 7), the kernel clears the file name from the parent directory: Writing a 0 for the value of the inode number suffices to clear the slot in the directory. The kernel then does a synchronous write of the directory to disk to ensure that the file is inaccessible by its old name, decrements the link count, and releases the in-core inodes of the parent directory and the unlinked file via algorithm *iput*.

When releasing the in-core inode of the unlinked file in *iput*, if the reference count drops to 0, and if the link count is 0, the kernel reclaims the disk blocks occupied by the file. No file names refer to the inode any longer and the inode is not active. To reclaim the disk blocks, the kernel loops through the inode table of contents, freeing all direct blocks immediately (according to algorithm *free*). For the indirect blocks, it recursively frees all blocks that appear in the various levels of indirection, freeing the more direct blocks first. It zeroes out the block numbers in the inode table of contents and sets the file size in the inode to 0. It then clears the inode file type field to indicate that the inode is free and frees the inode with algorithm *ifree*. It updates the disk since the disk copy of the inode still indicated that the inode was in use; the inode is now free for assignment to other files.

```
algorithm unlink
input:  file name
output: none
{
        get parent inode of file to be unlinked (algorithm namei);
        /* if unlinking the current directory... */
        if (last component of file name is ".")
                increment inode reference count;
        else
                get inode of file to be unlinked (algorithm iget);
        if (file is directory but user is not super user)
        {
                release inodes (algorithm iput);
                return(error);
        }
        if (shared text file and link count currently 1)
                remove from region table;
        write parent directory:  zero inode number of unlinked file;
        release inode parent directory (algorithm iput);
        decrement file link count;
        release file inode (algorithm iput);
                /* iput checks if link count is 0:  if so,
                 * releases file blocks (algorithm free) and
                 * frees inode (algorithm ifree);
                 */
}
```

Figure 5.31. Algorithm for Unlinking a File

5.16.1 File System Consistency

The kernel orders its writes to disk to minimize file system corruption in event of system failure. For instance, when it removes a file name from its parent directory, it writes the directory synchronously to the disk — before it destroys the contents of the file and frees the inode. If the system were to crash before the file contents were removed, damage to the file system would be minimal: There would be an inode that would have a link count 1 greater than the number of directory entries that access it, but all other paths to the file would still be legal. If the directory write were not synchronous, it would be possible for the directory entry on disk to point to a free (or reallocated!) inode after a system crash. Thus there would be more directory entries in the file system that refer to the inode than the inode would have link counts. In particular, if the file name was that of the last link to the file, it would refer to an unallocated inode. System damage is clearly less severe and easier to correct in the first case (see Section 5.18).

For example, suppose a file has two links with path names "a" and "b", and suppose a process *unlink*s "a". If the kernel orders the disk write operations, then it zeros the directory entry for "a" and writes it to disk. If the system crashes after the write to disk completes, file "b" has link count of 2, but file "a" does not exist because its old entry had been zeroed before the system crash. File "b" has an extra link count, but the system functions properly when rebooted.

Now suppose the kernel ordered the disk write operations in the reverse order and the system crashes: That is, it decrements the link count for the file "b" to 1, writes the inode to disk, and crashes before it could zero the directory entry for file "a". When the system is rebooted, entries for files "a" and "b" exist in their respective directories, but the link count for the file they reference is 1. If a process then *unlink*s file "a", the file link count drops to 0 even though file "b" still references the inode. If the kernel were later to reassign the inode as the result of a *creat* system call, the new file would have link count 1 but two path names that reference it. The system cannot rectify the situation except via maintenance programs (*fsck*, described in Section 5.18) that access the file system through the block or raw interface.

The kernel also frees inodes and disk blocks in a specific order to minimize corruption in event of system failure. When removing the contents of a file and clearing its inode, it is possible to free the blocks containing the file data first, or it is possible to free and write out the inode first. The result is usually identical for both cases, but it differs if the system crashes in the middle. Suppose the kernel first frees the disk blocks of a file and crashes. When the system is rebooted, the inode still contains references to the old disk blocks, which may no longer contain data relevant to the file. The kernel would see an apparently good file, but a user accessing the file would notice corruption. It is also possible that other files were assigned those disk blocks. The effort to clean the file system with the *fsck* program would be great. However, if the system first writes the inode to disk and the system crashes, a user would not notice anything wrong with the file system when the system is rebooted. The data blocks that previously belonged to the file would be inaccessible to the system, but users would notice no apparent corruption. The *fsck* program also finds the task of reclaiming unlinked disk blocks easier than the clean-up it would have to do for the first sequence of events.

5.16.2 Race Conditions

Race conditions abound in the *unlink* system call, particularly when unlinking directories. The *rmdir* command removes a directory after verifying that the directory contains no files (it *read*s the directory and checks that all directory entries have inode value 0). But since *rmdir* runs at user level, the actions of verifying that a directory is empty and removing the directory are not atomic; the system could do a context switch between execution of the *read* and *unlink* system calls. Hence, another process could *creat* a file in the directory after *rmdir* determined that the directory was empty. Users can prevent this situation only by

use of file and record locking. Once a process begins execution of the *unlink* call, however, no other process can access the file being unlinked since the inodes of the parent directory and the file are locked.

Recall the algorithm for the *link* system call and how the kernel unlocks the inode before completion of the call. If another process should *unlink* the file while the inode lock is free, it would only decrement the link count; since the link count had been incremented before unlinking the inode, the count would still be greater than 0. Hence, the file cannot be removed, and the system is safe. The condition is equivalent to the case where the *unlink* happens immediately after the *link* call completes.

Another race condition exists in the case where one process is converting a file path name to an inode using algorithm *namei* and another process is removing a directory in that path. Suppose process A is parsing the path name "a/b/c/d" and goes to sleep while allocating the in-core inode for "c". It could go to sleep while trying to lock the inode or while trying to access the disk block in which the inode resides (see algorithms *iget* and *bread*). If process B wants to *unlink* the directory "c", it may go to sleep, possibly for the same reasons that process A is sleeping. Suppose the kernel later schedules process B to run before process A. Process B would run to completion, unlinking directory "c" and removing it and its contents (for the last link) before process A runs again. Later, process A would try to access an illegal in-core inode that had been removed. Algorithm *namei* therefore checks that the link count is not 0 before proceeding, reporting an error otherwise.

The check is not sufficient, however, because another process could conceivably create a new directory somewhere in the file system and allocate the inode that had previously been used for "c". Process A is tricked into thinking that it accessed the correct inode (see Figure 5.32). Nevertheless, the system maintains its integrity; the worst that could happen is that the wrong file is accessed — a possible security breach — but the race condition is rare in practice.

A process can *unlink* a file while another process has the file open. (The *unlink*ing process could even be the process that did the *open*). Since the kernel unlocks the inode at the end of the *open* call, the *unlink* call will succeed. The kernel will follow the *unlink* algorithm as if the file were not open, and it will remove the directory entry for the file. No other processes will be able to access the now *unlink*ed file. However, since the *open* system call had incremented the inode reference count, the kernel does not clear the file contents when executing the *iput* algorithm at the conclusion of the *unlink* call. So the opening process can do all the normal file operations with its file descriptor, including *read*ing and *writ*ing the file. But when it *close*s the file, the inode reference count drops to 0 in *iput*, and the kernel clears the contents of the file. In short, the process that had *open*ed the file proceeds as if the *unlink* did not occur, and the *unlink* happens as if the file were not open. Other system calls will continue to work for the opening process, too.

In Figure 5.33 for example, a process *open*s a file supplied as a parameter and then *unlink*s the file it just *open*ed. The *stat* call fails because the original path

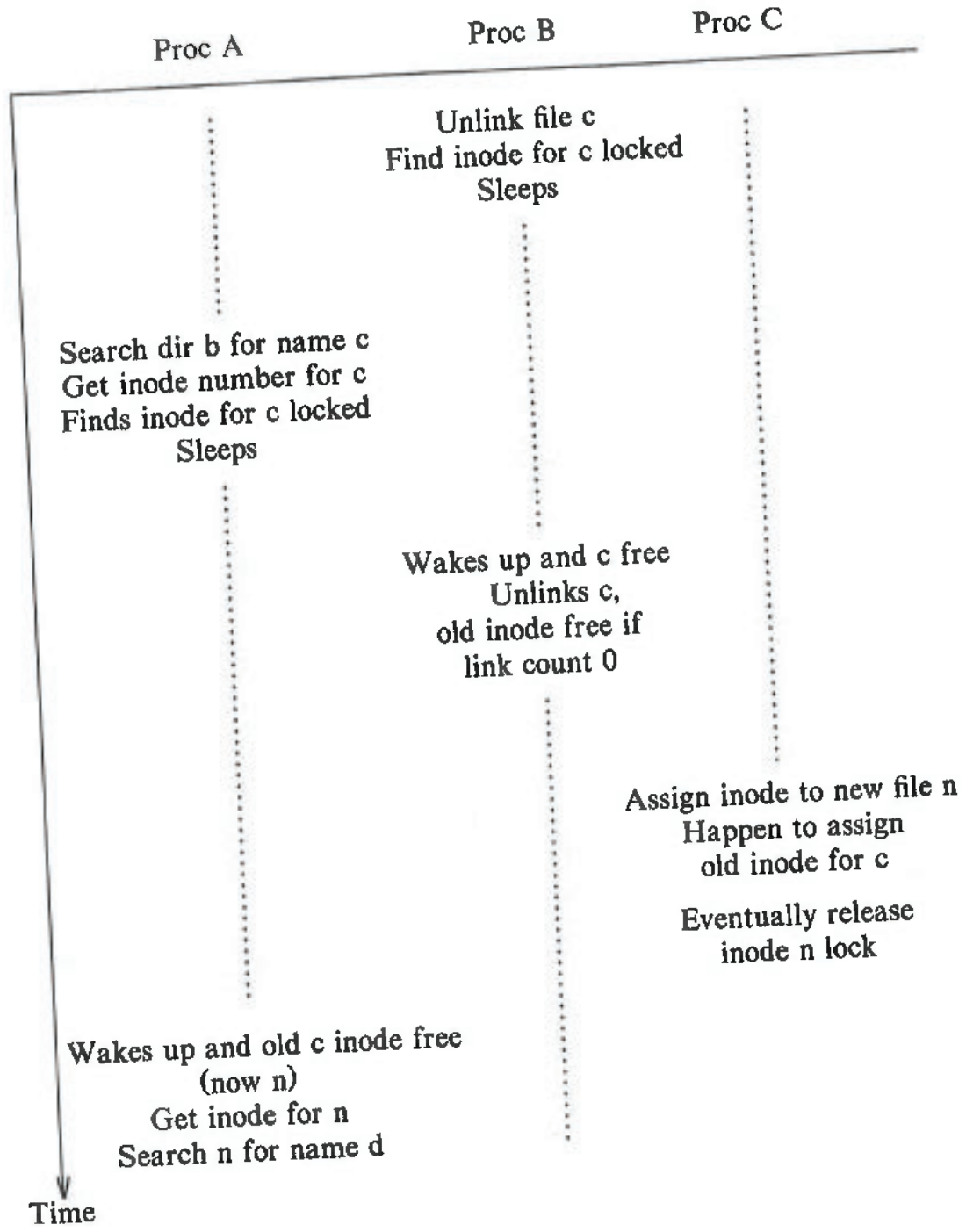


Figure 5.32. Unlink Race Condition

```
#include  <sys/types.h>
#include  <sys/stat.h>
#include  <fcntl.h>

main(argc, argv)
        int argc;
        char *argv[];
{
        int fd;
        char buf[1024];
        struct stat statbuf;

        if (argc != 2)                  /* need a parameter */
                exit();
        fd = open(argv[1], O_RDONLY);
        if (fd == -1)                   /* open fails */
                exit();
        if (unlink(argv[1]) == -1)      /* unlink file just opened */
                exit();
        if (stat(argv[1], &statbuf) == -1)      /* stat the file by name*/
                printf("stat %s fails as it should\n", argv[1]);
        else
                printf("stat %s succeeded!!!!\n", argv[1]);
        if (fstat(fd, &statbuf) == -1)          /* stat the file by fd */
                printf("fstat %s fails!!!\n", argv[1]);
        else
                printf("fstat %s succeeds as it should\n", argv[1]);
        while (read(fd, buf, sizeof(buf)) > 0)  /* read open/unlinked file */
                printf("%1024s", buf);          /* prints 1K byte field */
}
```

Figure 5.33. Unlinking an Opened File

name no longer refers to a file after the *unlink* (assuming no other process created a file by that name in the meantime), but the *fstat* call succeeds because it gets to the inode via the file descriptor. The process loops, *read*ing the file 1024 bytes at a time and printing the file to the standard output. When the *read* encounters the end of the file, the process *exit*s: After the close in *exit*, the file no longer exists. Processes commonly create temporary files and immediately unlink them; they can continue to read and write them, but the file name no longer appears in the directory hierarchy. If the process should fail for some reason, it leaves no trail of temporary files behind it.

5.17 FILE SYSTEM ABSTRACTIONS

Weinberger introduced *file system types* to support his network file system (see [Killian 84] for a brief description of this mechanism), and the latest release of System V supports a derivation of his scheme. File system types allow the kernel to support multiple file systems simultaneously, such as network file systems (Chapter 13) or even file systems of other operating systems. Processes use the usual UNIX system calls to access files, and the kernel maps a generic set of file operations into operations specific to each file system type.

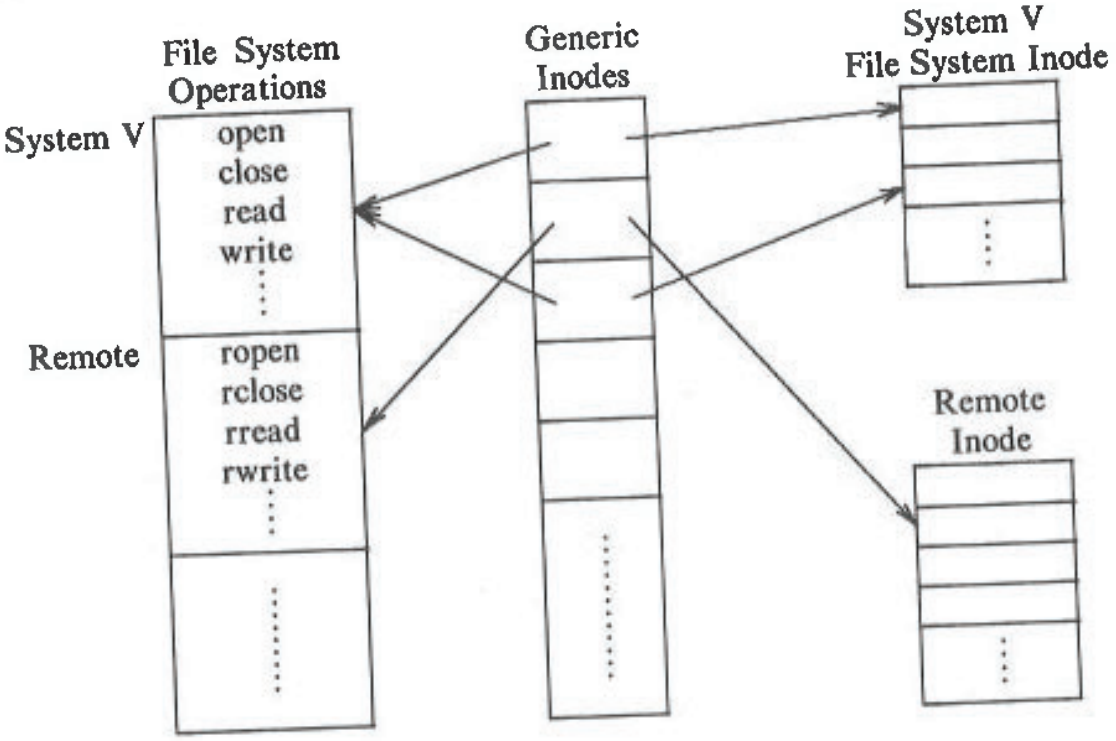


Figure 5.34. Inodes for File System Types

The inode is the interface between the abstract file system and the specific file system. A generic in-core inode contains data that is independent of particular file systems, and points to a file-system-specific inode that contains file-system-specific data. The file-system-specific inode contains information such as access permissions and block layout, but the generic inode contains the device number, inode number, file type, size, owner, and reference count. Other data that is file-system-specific includes the super block and directory structures. Figure 5.34 depicts the generic in-core inode table and two tables of file-system-specific inodes, one for System V file system structures and the other for a remote (network) inode. The latter inode presumably contains enough information to identify a file on a remote system. A file system may not have an inode-like structure; but the file-system-specific code manufactures an object that satisfies UNIX file system semantics and allocates its "inode" when the kernel allocates a generic inode.

Each file system type has a structure that contains the addresses of functions that perform abstract operations. When the kernel wants to access a file, it makes an indirect function call, based on the file system type and the operation (see Figure 5.34). Some abstract operations are to open a file, close it, read or write data, return an inode for a file name component (like *namei* and *iget*), release an inode (like *iput*), update an inode, check access permissions, set file attributes (permissions), and mount and unmount file systems. Chapter 13 will illustrate the use of file system abstractions in the description of a distributed file system.

5.18 FILE SYSTEM MAINTENANCE

The kernel maintains consistency of the file system during normal operation. However, extraordinary circumstances such as a power failure may cause a system crash that leaves a file system in an inconsistent state: most of the data in the file system is acceptable for use, but some inconsistencies exist. The command *fsck* checks for such inconsistencies and repairs the file system if necessary. It accesses the file system by its block or raw interface (Chapter 10) and bypasses the regular file access methods. This section describes several inconsistencies checked by *fsck*.

A disk block may belong to more than one inode or to the list of free blocks and an inode. When a file system is originally set up, all disk blocks are on the free list. When a disk block is assigned for use, the kernel removes it from the free list and assigns it to an inode. The kernel may not reassign the disk block to another inode until the disk block has been returned to the free list. Therefore, a disk block is either on the free list or assigned to a single inode. Consider the possibilities if the kernel freed a disk block in a file, returning the block number to the in-core copy of the super block, and allocated the disk block to a new file. If the kernel wrote the inode and blocks of the new file to disk but crashed before updating the inode of the old file to disk, the two inodes would address the same disk block number. Similarly, if the kernel wrote the super block and its free list to disk and crashed before writing the old inode out, the disk block would appear on the free list and in the old inode.

If a block number is not on the free list of blocks nor contained in a file, the file system is inconsistent because, as mentioned above, all blocks must appear somewhere. This situation could happen if a block was removed from a file and placed on the super block free list. If the old file was written to disk and the system crashed before the super block was written to disk, the block would not appear on any lists stored on disk.

An inode may have a non-0 link count, but its inode number may not exist in any directories in the file system. All files except (unnamed) pipes must exist in the file system tree. If the system crashes after creating a pipe or after creating a file but before creating its directory entry, the inode will have its link field set even though it does not appear to be in the file system. The problem could also arise if a directory were *unlink*ed before making sure that all files contained in the directory were *unlink*ed.

If the format of an inode is incorrect (for instance, if the file type field has an undefined value), something is wrong. This could happen if an administrator mounted an improperly formatted file system. The kernel accesses disk blocks that it thinks contain inodes but in reality contain data.

If an inode number appears in a directory entry but the inode is free, the file system is inconsistent because an inode number that appears in a directory entry should be that of an allocated inode. This could happen if the kernel was creating a new file and wrote the directory entry to disk but did not write the inode to disk before the crash. It could also occur if a process *unlink*ed a file and wrote the freed inode to disk, but did not write the directory element to disk before it crashed. These situations are avoided by ordering the write operations properly.

If the number of free blocks or free inodes recorded in the super block does not conform to the number that exist on disk, the file system is inconsistent. The summary information in the super block must always be consistent with the state of the file system.

5.19 SUMMARY

This chapter concludes the first part of the book, the explanation of the file system. It introduced three kernel tables: the user file descriptor table, the system file table, and the mount table. It described the algorithms for many system calls relating to the file system and their interaction. It introduced file system abstractions, which allow the UNIX system to support varied file system types. Finally, it described how *fsck* checks the consistency of the file system.

5.20 EXERCISES

1. Consider the program in Figure 5.35. What is the return value for all the *read*s and what is the contents of the buffer? Describe what is happening in the kernel during each *read*.
2. Reconsider the program in Figure 5.35 but suppose the statement

   ```
   lseek(fd, 9000L, 0);
   ```

   is placed before the first *read*. What does the process see and what happens inside the kernel?
3. A process can *open* a file in write-append mode, meaning that every write operations starts at the byte offset marking the current end of file. Therefore, two processes can *open* a file in write-append mode and write the file without overwriting data. What happens if a process *open*s a file in write-append mode and seeks to the beginning of the file?
4. The standard I/O library makes user reading and writing more efficient by buffering the data in the library and thus potentially saving the number of system calls a user has to make. How would you implement the library functions *fread* and *fwrite*? What should the library functions *fopen* and *fclose* do?

```
#include <fcntl.h>
main()
{
        int fd;
        char buf[1024];
        fd = creat("junk", 0666);
        lseek(fd, 2000L, 2);            /* seek to byte 2000 */
        write(fd, "hello", 5);
        close(fd);

        fd = open("junk", O_RDONLY);
        read(fd, buf, 1024);            /* read zero's */
        read(fd, buf, 1024);            /* catch something */
        read(fd, buf, 1024);
}
```

Figure 5.35. Reading 0s and End of File

5. If a process is reading data consecutively from a file, the kernel notes the value of the read-ahead block in the in-core inode. What happens if several processes simultaneously read data consecutively from the same file?

```
#include <fcntl.h>
main()
{
        int fd;
        char buf[256];

        fd = open("/etc/passwd", O_RDONLY);
        if (read(fd, buf, 1024) < 0)
                printf("read fails\n");
}
```

Figure 5.36. A Big Read in a Little Buffer

6. Consider the program in Figure 5.36. What happens when the program is executed? Why? What would happen if the declaration of *buf* were sandwiched between the declaration of two other arrays of size 1024? How does the kernel recognize that the *read* is too big for the buffer?

* 7. The BSD file system allows fragmentation of the last block of a file as needed, according to the following rules:

   - Structures similar to the super block keep track of free fragments;
   - The kernel does not keep a preallocated pool of free fragments but breaks a free block into fragments when necessary;

- The kernel can assign block fragments only for the last block of a file;
- If a block is partitioned into several fragments, the kernel can assign them to different files;
- The number of fragments in a block is fixed per file system;
- The kernel allocates fragments during the *write* system call.

Design an algorithm that allocates block fragments to a file. What changes must be made to the inode to allow for fragments? How advantageous is it from a performance standpoint to use fragments for files that use indirect blocks? Would it be more advantageous to allocate fragments during a *close* call instead of during a *write* call?

* 8. Recall the discussion in Chapter 4 for placing data in a file's inode. If the size of the inode is that of a disk block, design an algorithm such that the last data of a file is written in the inode block if it fits. Compare this method with that described in the previous problem.

* 9. System V uses the *fcntl* system call to implement file and record locking:

```
fcntl(fd, cmd, arg);
```

where *fd* is the file descriptor, *cmd* specifies the type of locking operation, and *arg* specifies various parameters, such as lock type (read or write) and byte offsets (see the appendix). The locking operations include

- Test for locks belonging to other processes and return immediately, indicating whether other locks were found,
- Set a lock and sleep until successful,
- Set a lock but return immediately if unsuccessful.

The kernel automatically releases locks set by a process when it *close*s the file. Describe an algorithm that implements file and record locking. If the locks are *mandatory*, other processes should be prevented from accessing the file. What changes must be made to *read* and *write*?

* 10. If a process goes to sleep while waiting for a file lock to become free, the possibility for deadlock exists: process A may lock file "one" and attempt to lock file "two," and process B may lock file "two" and attempt to lock file "one." Both processes are in a state where they cannot continue. Extend the algorithm of the previous problem so that the kernel detects the deadlock situation as it is about to occur and fails the system call. Is the kernel the right place to check for deadlocks?

11. Before the existence of a file locking system call, users could get cooperating processes to implement a locking mechanism by executing system calls that exhibited atomic features. What system calls described in this chapter could be used? What are the dangers inherent in using such methods?

12. Ritchie claims (see [Ritchie 81]) that file locking is not sufficient to prevent the confusion caused by programs such as editors that make a copy of a file while editing and then write the original file when done. Explain what he meant and comment.

13. Consider another method for locking files to prevent destructive update: Suppose the inode contains a new permission setting such that it allows only one process at a time to *open* the file for writing, but many processes can *open* the file for reading. Describe an implementation.

* 14. Consider the program in Figure 5.37 that creates a directory node in the wrong format (there are no directory entries for "." and ".."). Try a few commands on the new directory such as *ls −l*, *ls −ld*, or *cd*. What is happening?

```
main(argc, argv)
        int argc;
        char *argv[];
{
        if (argc != 2)
        {
                printf("try: command directory name\n");
                exit();
        }

        /* modes indicate: directory (04) rwx permission for all */
        /* only super user can do this */
        if (mknod(argv[1], 040777, 0) == -1)
                printf("mknod fails\n");
}
```

Figure 5.37. A Half-Baked Directory

15. Write a program that prints the owner, file type, access permissions, and access times of files supplied as parameters. If a file (parameter) is a directory, the program should *read* the directory and print the above information for all files in the directory.
16. Suppose a directory has read permission for a user but not execute permission. What happens when the directory is used as a parameter to *ls* with the "–i" option? What about the "–l" option? Explain the answers. Repeat the problem for the case that the directory has execute permission but not read permission.
17. Compare the permissions a process must have for the following operations and comment.
    - Creating a new file requires write permission in a directory.
    - Creating an existing file requires write permission on the file.
    - Unlinking a file requires write permission in the directory, not on the file.

* 18. Write a program that visits every directory, starting with the current directory. How should it handle loops in the directory hierarchy?

19. Execute the program in Figure 5.38 and describe what happens in the kernel. (Hint: Execute *pwd* when the program completes.)
20. Write a program that changes its root to a particular directory, and investigate the directory tree accessible to that program.
21. Why can't a process undo a previous *chroot* system call? Change the implementation so that it can change its root back to a previous root. What are the advantages and disadvantages of such a feature?
22. Consider the simple pipe example in Figure 5.19, where a process *write*s the string "hello" in the pipe then *read*s the string. What would happen if the count of data written to the pipe were 1024 instead of 6 (but the count of read data stays at 6)? What would happen if the order of the *read* and *write* system calls were reversed?
23. In the program illustrating the use of named pipes (Figure 5.19), what happens if *mknod* discovers that the named pipe already exists? How does the kernel implement this? What would happen if many reader and writer processes all attempted to

```
main(argc, argv)
        int argc;
        char *argv[];
{
        if (argc != 2)
        {
                printf("need 1 dir arg\n");
                exit();
        }

        if (chdir(argv[1]) == -1)
                printf("%s not a directory\n", argv[1]);
}
```

Figure 5.38. Sample Program with Chdir System Call

communicate through the named pipe instead of the one reader and one writer implicit in the text? How could the processes ensure that only one reader and one writer process were communicating?

24. When *open*ing a named pipe for reading, a process sleeps in the *open* until another process *open*s the pipe for writing. Why? Couldn't the process return successfully from the *open*, continue processing until it tried to *read* from the pipe, and sleep in the *read*?
25. How would you implement the *dup2* (from Version 7) system call with syntax

    dup2(oldfd, newfd);

    where *oldfd* is the file descriptor to be *dup*ed to file descriptor number *newfd*? What should happen if *newfd* already refers to an open file?

* 26. What strange things could happen if the kernel would allow two processes to mount the same file system simultaneously at two mount points?

27. Suppose a process changes its current directory to "/mnt/a/b/c" and a second process then *mount*s a file system onto "/mnt". Should the *mount* succeed? What happens if the first process executes *pwd*? The kernel does not allow the *mount* to succeed if the inode reference count of "/mnt" is greater than 1. Comment.
28. In the algorithm for crossing a mount point on recognition of ".." in the file path name, the kernel checks three conditions to see if it is at a mount point: that the found inode has the root inode number, that the working inode is root of the file system, and that the path name component is "..". Why must it check all three conditions? Show that checking any two conditions is insufficient to allow the process to cross the mount point.
29. If a user *mount*s a file system "read-only," the kernel sets a flag in the super block. How should it prevent write operations during the *write*, *creat*, *link*, *unlink*, *chown*, and *chmod* system calls? What write operations do all the above system calls do to the file system?

* 30. Suppose a process attempts to *umount* a file system and another process is simultaneously attempting to *creat* a new file on that file system. Only one system call can succeed. Explore the race condition.

* 31. When the *umount* system call checks that no more files are active on a file system, it has a problem with the file system root inode, allocated via *iget* during the *mount* system call and hence having reference count greater than 0. How can *umount* be sure there are no active files and take account for the file system root? Consider two cases:
  - *umount* releases the root inode with the *iput* algorithm before checking for active inodes. (How does it recover if there were active files after all?)
  - *umount* checks for active files before releasing the root inode but permits the root inode to remain active. (How active can the root inode get?)

32. When executing the command *ls −ld* on a directory, note that the number of links to the directory is never 1. Why?

33. How does the command *mkdir* (make a new directory) work? (Hint: When *mkdir* completes, what are the inode numbers for "." and ".."?)

* 34. Symbolic links refer to the capability to *link* files that exist on different file systems. A new type indicator specifies a symbolic link file; the data of the file is the path name of the file to which it is linked. Describe an implementation of symbolic links.

* 35. What happens when a process executes

```
unlink(".");
```

What is the current directory of the process? Assume superuser permissions.

36. Design a system call that truncates an existing file to arbitrary sizes, supplied as an argument, and describe an implementation. Implement a system call that allows a user to remove a file segment between specified byte offsets, compressing the file size. Without such system calls, encode a program that provides this functionality.

37. Describe all conditions where the reference count of an inode can be greater than 1.

38. In file system abstractions, should each file system type support a private lock operation to be called from the generic code, or does a generic lock operation suffice?

10

THE I/O SUBSYSTEM

The I/O subsystem allows a process to communicate with peripheral devices such as disks, tape drives, terminals, printers, and networks, and the kernel modules that control devices are known as *device drivers*. There is usually a one-to-one correspondence between device drivers and device types: Systems may contain one disk driver to control all disk drives, one terminal driver to control all terminals, and one tape driver to control all tape drives. Installations that have devices from more than one manufacturer — for example, two brands of tape drives — may treat the devices as two different device types and have two separate drivers, because such devices may require different command sequences to operate properly. A device driver controls many physical devices of a given type. For example, one terminal driver may control all terminals connected to the system. The driver distinguishes among the many devices it controls: Output intended for one terminal must not be sent to another.

The system supports "software devices," which have no associated physical device. For example, it treats physical memory as a device to allow a process access to physical memory outside its address space, even though memory is not a peripheral device. The *ps* command, for instance, *read*s kernel data structures from physical memory to report process statistics. Similarly, drivers may write trace records useful for debugging, and a trace driver may allow users to read the records. Finally, the kernel profiler described in Chapter 8 is implemented as a driver: A process *write*s addresses of kernel routines found in the kernel symbol

table and *read*s profiling results.

This chapter examines the interfaces between processes and the I/O subsystem and between the machine and the device drivers. It investigates the general structure and function of device drivers, then treats disk drivers and terminal drivers as detailed examples of the general interface. It concludes with a description of a new method for implementing device drivers called *streams*.

10.1 DRIVER INTERFACES

The UNIX system contains two types of devices, *block* devices and *raw* or *character* devices. As defined in Chapter 2, block devices, such as disks and tapes, look like random access storage devices to the rest of the system; character devices include all other devices such as terminals and network media. Block devices may have a character device interface, too.

The user interface to devices goes through the file system (recall Figure 2.1): Every device has a name that looks like a file name and is accessed like a file. The device special file has an inode and occupies a node in the directory hierarchy of the file system. The device file is distinguished from other files by the file type stored in its inode, either "block" or "character special," corresponding to the device it represents. If a device has both a block and character interface, it is represented by two device files: its block device special file and its character device special file. System calls for regular files, such as *open*, *close*, *read*, and *write*, have an appropriate meaning for devices, as will be explained later. The *ioctl* system call provides an interface that allows processes to control character devices, but it is not applicable to regular files.[1] However, each device driver need not support every system call interface. For example, the trace driver mentioned earlier allows users to *read* records written by other drivers, but it does not allow users to *write* it.

10.1.1 System Configuration

System configuration is the procedure by which administrators specify parameters that are installation dependent. Some parameters specify the sizes of kernel tables, such as the process table, inode table, and file table, and the number of buffers to be allocated for the buffer pool. Other parameters specify device configuration, telling the kernel which devices are included in the installation and their "address." For instance, a configuration may specify that a terminal board is plugged into a

1. Conversely, the *fcntl* system call provides control of operations at the file descriptor level, not the device level. Other implementations interpret *ioctl* for all file types.

particular slot on the hardware backplane.

There are three stages at which device configuration can be specified. First, administrators can hard-code configuration data into files that are compiled and linked when building the kernel code. The configuration data is typically specified in a simple format, and a configuration program converts it into a file suitable for compilation. Second, administrators can supply configuration information after the system is already running; the kernel updates internal configuration tables dynamically. Finally, self-identifying devices permit the kernel to recognize which devices are installed. The kernel reads hardware switches to configure itself. The details of system configuration are beyond the scope of this book, but in all cases, the configuration procedure generates or fills in tables that form part of the code of the kernel.

The kernel to driver interface is described by the *block device switch* table and the *character device switch* table (Figure 10.1). Each device type has entries in the table that direct the kernel to the appropriate driver interfaces for the system calls. The *open* and *close* system calls of a device file funnel through the two device switch tables, according to the file type. The *mount* and *umount* system calls also invoke the device open and close procedures for block devices. *Read*, *write*, and *ioctl* system calls of character special files pass through the respective procedures in the character device switch table. *Read* and *write* system calls of block devices and of files on mounted file systems invoke the algorithms of the buffer cache, which invoke the device strategy procedure. Some drivers invoke the strategy procedure internally from their read and write procedures, as will be seen. The next section explores each driver interface in greater detail.

The hardware to driver interface consists of machine-dependent control registers or I/O instructions for manipulating devices and interrupt vectors: When a device interrupt occurs, the system identifies the interrupting device and calls the appropriate interrupt handler. Obviously, software devices such as the kernel profiler driver (Chapter 8) do not have a hardware interface, but other interrupt handlers may call a "software interrupt handler" directly. For example, the clock interrupt handler calls the kernel profiler interrupt handler.

Administrators set up device special files with the *mknod* command, supplying file type (block or character) and major and minor numbers. The *mknod* command invokes the *mknod* system call to create the device file. For example, in the command line

```
mknod /dev/tty13 c 2 13
```

"/dev/tty13" is the file name of the device, *c* specifies that it is a character special file (*b* specifies a block special file), 2 is the major number, and 13 is the minor number. The major number indicates a device type that corresponds to the appropriate entry in the block or character device switch tables, and the minor number indicates a unit of the device. If a process *opens* the block special file "/dev/dsk1" and its major number is 0, the kernel calls the routine *gdopen* in entry 0 of the block device switch table (Figure 10.2); if a process *read*s the character

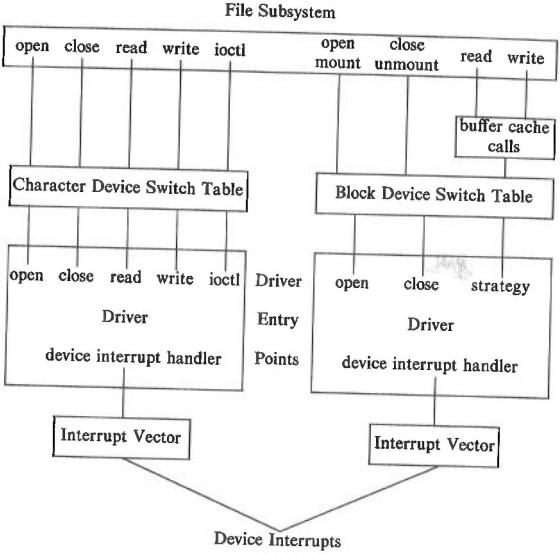


Figure 10.1. Driver Entry Points

special file "/dev/mem" and its major number is 3, the kernel calls the routine *mmread* in entry 3 of the character device switch table. The routine *nulldev* is an "empty" routine, used when there is no need for a particular driver function. Many peripheral devices can be associated with a major device number; the minor device number distinguishes them from each other. Device special files do not have to be created every time the system is booted; they need be changed only if the configuration changes, such as when adding devices to an installation.

10.1.2 System Calls and the Driver Interface

This section describes the interface between the kernel and device drivers. For system calls that use file descriptors, the kernel follows pointers from the user file

block device switch table			
entry	open	close	strategy
0	gdopen	gdclose	gdstrategy
1	gtopen	gtclose	gtstrategy

character device switch table					
entry	open	close	read	write	ioctl
0	conopen	conclose	conread	conwrite	conioctl
1	dzbopen	dzbclose	dzbread	dzbwrite	dzbioctl
2	syopen	nulldev	syread	sywrite	syioctl
3	nulldev	nulldev	mmread	mmwrite	nodev
4	gdopen	gdclose	gdread	gdwrite	nodev
5	gtopen	gtclose	gtread	gtwrite	nodev

Figure 10.2. Sample Block and Character Device Switch Tables

descriptor to the kernel file table and inode, where it examines the file type and accesses the block or character device switch table, as appropriate. It extracts the major and minor numbers from the inode, uses the major number as an index into the appropriate table, and calls the driver function according to the system call being made, passing the minor number as a parameter. An important difference between system calls for devices and regular files is that the inode of a special file is *not* locked while the kernel executes the driver. Drivers frequently sleep, waiting for hardware connections or for the arrival of data, so the kernel cannot determine how long a process will sleep. If the inode was locked, other processes that access the inode (via the *stat* system call, for example) would sleep indefinitely because another process is asleep in the driver.

The device driver interprets the parameters of the system call as appropriate for the device. A driver maintains data structures that describe the state of each unit that it controls; driver functions and interrupt handlers execute according to the state of the driver and the action being done (for example, data being input or output). Each interface will now be described in greater detail.

10.1.2.1 Open

The kernel follows the same procedure for *open*ing a device as it does for *open*ing regular files (see Section 5.1), allocating an in-core inode, incrementing its reference count, and assigning a file table entry and user file descriptor. The kernel eventually returns the user file descriptor to the calling process, so that *open*ing a device looks like *open*ing a regular file. However, it invokes the device-specific *open* procedure before returning to user mode, (Figure 10.3). For a block device, it

```
algorithm open                  /* for device drivers */
input:   pathname
         openmode
output: file descriptor
{
        convert pathname to inode, increment inode reference count,
        allocate entry in file table, user file descriptor,
                as in open of regular file;

        get major, minor number from inode;

        save context (algorithm setjmp) in case of long jump from driver;

        if (block device)
        {
                use major number as index to block device switch table;
                call driver open procedure for index:
                                pass minor number, open modes;
        }
        else
        {
                use major number as index to character device switch table;
                call driver open procedure for index:
                                pass minor number, open modes;
        }

        if (open fails in driver)
                decrement file table, inode counts;
}
```

Figure 10.3. Algorithm for Opening a Device

invokes the *open* procedure encoded in the block device switch table, and for a character device, it invokes the *open* procedure in the character device switch table. If a device is both a block and a character device, the kernel will invoke the appropriate *open* procedure depending on the particular device file the user *opened*: The two open procedures may even be identical, depending on the driver.

The device-specific *open* procedure establishes a connection between the calling process and the *opened* device and initializes private driver data structures. For a terminal, for example, the *open* procedure may put the process to sleep until the machine detects a (hardware) carrier signal indicating that a user is trying to log in. It then initializes driver data structures according to appropriate terminal settings (such as the terminal baud rate). For software devices such as system memory, the *open* procedure may have no initialization to do.

If a process must sleep for some external reason when *open*ing a device, it is possible that the event that should awaken the process from its sleep may never occur. For example, if no user ever logs in to a particular terminal, the *getty* process that *open*ed the terminal (Section 7.9) sleeps until a user attempts to log in, potentially a long time. The kernel must be able to awaken the process from its sleep and cancel the *open* call on receipt of a signal: It must reset the inode, file table entry, and user file descriptor that it had allocated before entry into the driver, because the *open* fails. Hence, the kernel saves the process context using algorithm *setjmp* (Section 6.4.4) before entering the device-specific *open* routine; if the process awakens from its sleep because of a signal, the kernel restores the process context to its state before entering the driver using algorithm *longjmp* (Section 6.4.4) and releases all data structures it had allocated for the *open*. Similarly, the driver can catch the signal and clean up private data structures, if necessary. The kernel also readjusts the file system data structures when the driver encounters error conditions, such as when a user attempts to access a device that was not configured. The *open* call fails in such cases.

Processes may specify various options to qualify the device *open*. The most common option is "no delay," meaning that the process will not sleep during the *open* procedure if the device is not ready. The *open* system call returns immediately, and the user process has no knowledge of whether a hardware connection was made or not. *Open*ing a device with the "no delay" option also affects the semantics of the *read* system call, as will be seen (Section 10.3.4).

If a device is *open*ed many times, the kernel manipulates the user file descriptors and the inode and file table entries as described in Chapter 5, invoking the device specific *open* procedure for each *open* system call. The device driver can thus count how many times a device was *open*ed and fail the *open* call if the count is inappropriate. For example, it makes sense to allow multiple processes to *open* a terminal for writing so that users can exchange messages. But it does not make sense to allow multiple processes to *open* a printer for writing simultaneously, since they could overwrite each other's data. The differences are practical rather than implementational: allowing simultaneous writing to terminals fosters communication between users; preventing simultaneous writing to printers increases the chance of getting readable printouts.[2]

10.1.2.2 Close

A process severs its connection to an *open* device by *clos*ing it. However, the kernel invokes the device-specific *close* procedure only for the last *close* of the

2. In practice, printers are usually controlled by special spooler processes, and permissions are set up so that only the spooler can access the printer. But the analogy is still applicable.

device, that is, only if no other processes have the device *open*, because the device *close* procedure terminates hardware connections; clearly this must wait until no processes are accessing the device. Because the kernel invokes the device *open* procedure during every *open* system call but invokes the device *close* procedure only once, the device driver is never sure how many processes are still using the device. Drivers can easily put themselves out of state if not coded carefully: If they sleep in the *close* procedure and another process *opens* the device before the close completes, the device can be rendered useless if the combination of open and close results in an unrecognized state.

```
algorithm close                    /* for devices */
input:  file descriptor
output: none
{
      do regular close algorithm (chapter 5xxx);
      if (file table reference count not 0)
            goto finish;
      if (there is another open file and its major, minor numbers
                        are same as device being closed)
            goto finish;                /* not last close after all */
      if (character device)
      {
            use major number to index into character device switch table;
            call driver close routine:  parameter minor number;
      }
      if (block device)
      {
            if (device mounted)
                  goto finish;
            write device blocks in buffer cache to device;
            use major number to index into block device switch table;
            call driver close routine:  parameter minor number;
            invalidate device blocks still in buffer cache;
      }
   finish:
      release inode;
}
```

Figure 10.4. Algorithm for Closing a Device

The algorithm for *closing* a device is similar to the algorithm for closing a regular file (Figure 10.4). However, before the kernel releases the inode it does operations specific to device files.

1. It searches the file table to make sure that no other processes still have the device *open*. It is not sufficient to rely on the file table count to indicate the

last close of a device, because several processes may access the device via a different file table entry. It is also not sufficient to rely on the inode table count, because several device files may specify the same device. For example, the results of the following *ls −l* command show two character device files (the first "c" on the line) that refer to one device, because their major and minor numbers (9 and 1) are equal. The link count of 1 for each file implies that there are two inodes.

```
crw--w--w-   1 root   vis    9, 1 Aug 6 1984    /dev/tty01
crw--w--w-   1 root   unix   9, 1 May 3 15:02   /dev/fty01
```

If processes *open* the two files independently, they access different inodes but the same device.

2. For a character device, the kernel invokes the device *close* procedure and returns to user mode. For a block device, the kernel searches the mount table to make sure that the device does not contain a mounted file system. If there is a mounted file system from the block device, the kernel cannot invoke the device close procedure, because it is not the last *close* of the device. Even if the device does not contain a mounted file system, the buffer cache could still contain blocks of data that were left over from a previously mounted file system and never written to the device, because they were marked "delayed write." The kernel therefore searches the buffer cache for such blocks and writes them to the device before invoking the device *close* procedure. After *clos*ing the device, the kernel again goes through the buffer cache and invalidates all buffers that contain blocks for the now *close*d device, allowing buffers with useful data to stay in the cache longer.
3. The kernel releases the inode of the device file.

To summarize, the device *close* procedure severs the device connection and reinitializes driver data structures and device hardware, so that the kernel can reopen the device later on.

10.1.2.3 Read and Write

The kernel algorithms for *read* and *write* of a device are similar to those for a regular file. If the process is *read*ing or *writ*ing a character device, the kernel invokes the device driver *read* or *write* procedure. Although there are important cases where the kernel transmits data directly between the user address space and the device, device drivers may buffer data internally. For example, terminal drivers use *clist*s to buffer data (Section 10.3.1). In such cases, the device driver allocates a "buffer," copies data from user space during a *write*, and outputs the data from the "buffer" to the device. The driver write procedure throttles the amount of data being output (called flow control): If processes generate data faster than the device can output it, the write procedure puts processes to sleep until the device can accept more data. For a *read*, the device driver receives the data from the device in a

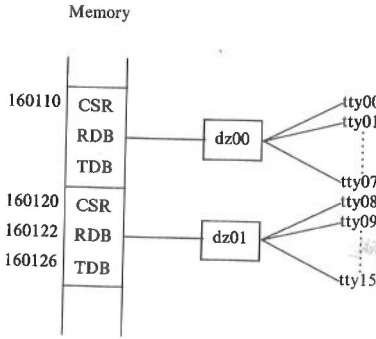


Figure 10.5. Memory Mapped I/O with the VAX DZ11 Controller

buffer and copies the data from the buffer to the user address specified in the system call.

The precise method in which a driver communicates with a device depends on the hardware. Some machines provide *memory mapped I/O*, meaning that certain addresses in the kernel address space are not locations in physical memory but are special registers that control particular devices. By writing control parameters to specified registers according to hardware specifications, the driver controls the device. For example, I/O controllers for the VAX-11 computer contain special registers for recording device status (*control and status registers*) and for data transmission (*data buffer registers*), which are configured at specific addresses in physical memory. In particular, the VAX DZ11 terminal controller controls 8 asynchronous lines for terminal communication (see [Levy 80] for more detail on the VAX architecture). Assume that the *control and status register* of a particular DZ11 is at address 160120, the *transmit data buffer register* is at address 160126, and the *receive data buffer register* is at address 160122 (Figure 10.5). To write a character to terminal "/dev/tty09", the terminal driver writes the number 1 (1 = 9 modulo 8) to a specified bit position in the *control and status register* and then writes the character to the *transmit data buffer register*. The operation of writing the *transmit data buffer register* transmits the data. The DZ11 controller sets a *done* bit in the *control and status register* when it is ready to accept more data. The driver can optionally set a *transmit interrupt enable* bit in the *control and status register*, which causes the DZ11 controller to interrupt the system when it is ready to accept more data. Reading data from the DZ11 is similar.

Other machines have *programmed I/O*, meaning that the machine contains instructions to control devices. Drivers control devices by executing the appropriate instructions. For example, the IBM 370 computer has a *Start I/O* instruction to initiate an I/O operation to a device. The method a driver uses to communicate with peripherals is transparent to the user.

Because the interface between device drivers and the underlying hardware is machine dependent, no standard interfaces exist at this level. For both memory-mapped I/O and programmed I/O, a driver can issue control sequences to a device to set up *direct memory access (DMA)* between the device and memory. The system allows bulk DMA transfer of data between the device and memory in parallel to CPU operations, and the device interrupts the system when such a transfer has completed. The driver sets up the virtual memory mapping so that the correct locations in memory are used for DMA.

High-speed devices can sometimes transfer data directly between the device and the user's address space, without intervention of a kernel buffer. This results in higher transfer speed because there is one less copy operation in the kernel, and the amount of data transmitted per transfer operation is not bounded by the size of kernel buffers. Drivers that make use of this "raw" I/O transfer usually invoke the block strategy interface from the character read and write procedures if they have a block counterpart.

10.1.2.4 Strategy Interface

The kernel uses the *strategy* interface to transmit data between the buffer cache and a device, although as mentioned above, the read and write procedures of character devices sometimes use their (block counterpart) *strategy* procedure to transfer data directly between the device and the user address space. The strategy procedure may queue I/O jobs for a device on a work list or do more sophisticated processing to schedule I/O jobs. Drivers can set up data transmission for one physical address or many, as appropriate. The kernel passes a buffer header address to the driver strategy procedure; the header contains a list of (page) addresses and sizes for transmission of data to or from the device. This is also how the swapping operations discussed in Chapter 9 work. For the buffer cache, the kernel transmits data from one data address; when swapping, the kernel transmits data from many data addresses (pages). If data is being copied to or from the user's address space, the driver must lock the process (or at least, the relevant pages) in memory until the I/O transfer is complete.

For example, after *mount*ing a file system, the kernel identifies every file in the file system by its device number and inode number. The device number is an encoding of the device major and minor numbers. When the kernel accesses a block from a file, it copies the device number and block number into the buffer header, as described in Chapter 3. When the buffer cache algorithms (*bread* or *bwrite*, for example) access the disk, they invoke the strategy procedure indicated by the device major number. The strategy procedure uses the minor number and

APPENDIX – SYSTEM CALLS

This appendix contains a brief synopsis of the UNIX system calls. Refer to the UNIX System V User Programmer's Manual for a complete specification of these calls. The specification here is sufficient for reference when reading the various program examples in the book.

The specified file names are null terminated character strings, whose individual components are separated by slash characters. All system calls return −1 on error, and the external variable errno indicates the specific error. Unless specified otherwise, system calls return 0 on success. Some system calls are the entry point for several functions: this means that the assembly language interface for the functions is the same. The list here follows the usual conventions for UNIX system manuals, but the programmer should not care whether a system call entry point handles one or many system calls.

access

```
access(filename, mode)
char *filename;
int mode;
```

Access checks if the calling process has read, write, or execute permission for the file, according to the value of *mode*. The value of *mode* is a combination of the bit

patterns 4 (for read), 2 (for write), and 1 (for execute). The real-user ID is checked instead of the effective user ID.

acct

```
acct(filename)
char *filename;
```

Acct enables system accounting if *filename* is non-null, and disables it otherwise.

alarm

```
unsigned alarm(seconds)
unsigned seconds;
```

Alarm schedules the occurrence of an alarm signal for the calling process in the indicated number of *seconds*. It returns the amount of time remaining until the alarm signal at the time of the call.

brk

```
int brk(end_data_seg)
char *end_data_seg;
```

Brk sets the highest address of a process's data region to *end_data_seg*. Another function, *sbrk*, uses this system call entry point and changes the highest address of a process's data region according to a specified increment.

chdir

```
chdir(filename)
char *filename;
```

Chdir changes the current directory of the calling process to *filename*.

chmod

```
chmod(filename, mode)
char *filename;
```

Chmod changes the access permissions of the indicated file to the specified *mode*, which is a combination of the following bits (in octal):

04000	setuid bit
02000	set group ID bit

01000	sticky bit
00400	read for owner
00200	write for owner
00100	execute for owner
00040	read for group
00020	write for group
00010	execute for group
00004	read for others
00002	write for others
00001	execute for others

chown

```
chown(filename, owner, group)
char *filename;
int owner, group;
```

Chown changes the owner and group of the indicated file to the specified *owner* and *group* IDs.

chroot

```
chroot(filename)
char *filename;
```

Chroot sets the private, changed-root of the calling process to *filename*.

close

```
close(fildes)
int fildes;
```

Close closes a file descriptor obtained from a prior *open*, *creat*, *dup*, *pipe*, or *fcntl* system call, or a file descriptor inherited from a *fork* call.

creat

```
creat(filename, mode)
char *filename;
int mode;
```

Creat creates a new file with the indicated file name and access permission modes. *Mode* is as specified in *access*, except that the sticky-bit is cleared and bits set via *umask* are cleared. If the file already exists, *creat* truncates the file. *Creat* returns a file descriptor for use in other system calls.

dup

```
dup(fildes)
int fildes;
```

Dup duplicates the specified file descriptor, returning the lowest available file descriptor. The old and new file descriptors use the same file pointer and share other attributes.

exec

```
execve(filename, argv, envp)
char *filename;
char *argv[];
char *envp[];
```

Execve executes the program file *filename*, overlaying the address space of the executing process. *Argv* is an array of character strings parameters to the *exec*ed program, and *envp* is an array of character strings that are the environment of the new process.

exit

```
exit(status)
int status;
```

Exit causes the calling process to terminate, reporting the 8 low-order bits of status to its waiting parent. The kernel may call exit internally, in response to certain signals.

fcntl

```
fcntl(fildes, cmd, arg)
int fildes, cmd, arg;
```

Fcntl supports a set of miscellaneous operations for open files, identified via the file descriptor *fildes*. The interpretation of *cmd* and *arg* is as follows (manifest constants are defined in file "/usr/include/fcntl.h"):

F_DUPFD	return lowest numbered file descriptor >= arg
F_SETFD	set close-on-exec flag to low order bit of arg (if 1, file is closed in exec)
F_GETFD	return value of close-on-exec flag
F_SETFL	set file status flags (O_NDELAY do not sleep for I/O and

O_APPEND append written data to end of file)

F_GETFL get file status flags

```
struct flock
        short l_type;    /* F_RDLCK for read lock, F_WRLCK for write lock,
                            F_UNLCK for unlock operations */
        short l_whence; /* lock offset is from beginning of file (0), current position of file
                            pointer (1), or end of file (2) */
        long l_start;    /* byte offset, interpreted according to l_whence */
        long l_len;      /* number of bytes to lock. If 0, lock from l_start to end of file */
        long l_pid;      /* ID of process that locked file */
        long l_sysid;    /* sys ID of process that locked file */
```

F_GETLK get first lock that would prevent application of the lock specified by *arg* and overwrite *arg*. If no such lock exists, change l_type in *arg* to F_UNLCK

F_SETLK lock or unlock the file as specified by *arg*. Return -1 if unable to lock.

F_SETLKW lock or unlock data in a file as specified by *arg*. Sleep if unable to lock.

Several read locks can overlap in a file. No locks can overlap a write lock.

fork

fork()

Fork creates a new process. The child process is a logical copy of the parent process, except that the parent's return value from the *fork* is the process ID of the child, and the child's return value is 0.

getpid

getpid()

Getpid returns the process ID of the calling process. Other calls that use this entry point are *getpgrp*, which returns the process group of the calling process, and *getppid*, which returns the parent process ID of the calling process.

getuid

getuid()

Getuid returns the real user ID of the calling process. Other calls that use this system call entry point are *geteuid*, which returns the effective user ID, *getgid*, which returns the group ID, and *getegid*, which returns the effective group ID of the calling process.

ioctl

```
ioctl(fildes, cmd, arg)
int fildes, cmd;
```

Ioctl does device-specific operations on the open device whose file descriptor is *fildes*. *Cmd* specifies the command to be done on the device, and *arg* is a parameter whose type depends on the command.

kill

```
kill(pid, sig)
int pid, sig;
```

Kill sends the signal sig to the processes identified by *pid*.

pid positive	send signal to process whose PID is pid.
pid 0	send signal to processes whose process group ID is PID of sender.
pid −1	if effective UID of sender is super user, send signal to all processes otherwise, send signal to all processes whose real UID equals effective UID of sender.
pid < −1	send signal to processes whose process group ID is pid.

The effective UID of the sender must be superuser, or the sender's real or effective UID must equal the real or effective UID of the receiving processes.

link

```
link(filename1, filename2)
char *filename1, *filename2;
```

Link gives another name, *filename2*, to the file *filename1*. The file becomes accessible through either name.

lseek

```
lseek(fildes, offset, origin)
int fildes, origin;
long offset;
```

Lseek changes the position of the read-write pointer for the file descriptor *fildes* and returns the new value. The value of the pointer depends on *origin*:

0	set the pointer to offset bytes from the beginning of the file.
1	increment the current value of the pointer by offset.
2	set the pointer to the size of the file plus offset bytes.

mknod

```
mknod(filename, modes, dev)
char *filename;
int mode, dev;
```

Mknod creates a special file, directory, or FIFO according to the type of *modes*:

010000	FIFO (named pipe)
020000	character special device file
040000	directory
060000	block special device file

The 12 low order bits of *modes* have the same meaning as described above for *chmod*. If the file is block special or character special, *dev* gives the major and minor numbers of the device.

mount

```
mount(specialfile, dir, rwflag)
char *specialfile, *dir;
int rwflag;
```

Mount mounts the file system specified by *specialfile* onto the directory *dir*. If the low-order bit of *rwflag* is 1, the file system is mounted read-only.

msgctl

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

msgctl(id, cmd, buf)
int id, cmd;
struct msqid_ds *buf;
```

Msgctl allows processes to set or query the status of the message queue id, or to remove the queue, according to the value of *cmd*. The structure *msqid_ds* is defined as follows:

```
struct ipc_perm {
        ushort          uid;        /* current user id */
        ushort          gid;        /* current group id */
        ushort          cuid;       /* creator user id */
        ushort          cgid;       /* creator group id */
        ushort          mode;       /* access modes */
        short           pad1;       /* used by system */
        long            pad2;       /* used by system */
};
```

```
struct msqid_ds {
        struct ipc_perm     msg_perm;    /* permission struct */
        short               pad1[7];     /* used by system */
        ushort              msg_qnum;    /* number of messages on q */
        ushort              msg_qbytes;  /* max number of bytes on q */
        ushort              msg_lspid;   /* pid of last msgsnd operation */
        ushort              msg_lrpid;   /* pid of last msgrcv operation */
        time_t              msg_stime;   /* last msgsnd time */
        time_t              msg_rtime;   /* last msgrcv time */
        time_t              msg_ctime;   /* last change time */
};
```

The commands and their meaning are as follows:

IPC_STAT	Read the message queue header associated with id into buf.
IPC_SET	Set the values of msg_perm.uid, msg_perm.gid, msg_perm.mode (9 low-order bits), and msg_qbytes from the corresponding values in buf.
IPC_RMID	Remove the message queue for id.

msgget

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

msgget(key, flag)
key_t key;
int flag;
```

Msgget returns an identifier to a message queue whose name is *key*. *Key* can specify that the returned queue identifier should refer to a private queue (*IPC_PRIVATE*), in which case a new message queue is created. *Flag* specifies if the queue should be created (*IPC_CREAT*), and if creation of the queue should be exclusive (*IPC_EXCL*). In the latter case, *msgget* fails if the queue already exists.

msgsnd and msgrcv

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

msgsnd(id, msgp, size, flag)
int id, size, flag;
struct msgbuf *msgp;
```

```
msgrcv(id, msgp, size, type, flag)
int id, size, type, flag;
struct msgbuf *msgmp;
```

Msgsnd sends a message of size bytes in the buffer *msgp* to the message queue *id*. *Msgbuf* is defined as

```
struct msgbuf
        long mtype;
        char mtext[];
};
```

If the *IPC_NOWAIT* bit is off in *flag*, *msgsnd* sleeps if the number of bytes on the message queue exceeds the maximum, or if the number of messages system-wide exceeds a maximum value. If *IPC_NOWAIT* is set, *msgsnd* returns immediately in these cases.

Msgrcv receives messages from the queue identified by *id*. If *type* is 0, the first message on the queue is received; if positive, the first message of that type is received; if negative, the first message of the lowest type less than or equal to type is received. *Size* indicates the maximum size of message text the user wants to receive. If *MSG_NOERROR* is set in *flag*, the kernel truncates the received message if its size is larger than *size*. Otherwise it returns an error. If *IPC_NOWAIT* is not set in *flag*, *msgrcv* sleeps until a message that satisfies *type* is sent. If *IPC_NOWAIT* is set, it returns immediately. *Msgrcv* returns the number of bytes in the message text.

nice

```
nice(increment)
int increment;
```

Nice adds *increment* to the process nice value. A higher nice value gives the process lower scheduling priorities.

open

```
#include <fcntl.h>

open(filename, flag, mode)
char *filename;
int flag, mode;
```

Open opens the specified file according to the value of *flag*. The value of *flag* is a combination of the following bits (exactly one of the first three bits must be used).

O_RDONLY	open for reading only.
O_WRONLY	open for writing only.
O_RDWR	open for reading and writing.
O_NDELAY	For special devices, open returns without waiting for carrier, if set. For named pipes, open will return immediately (with an error if O_WRONLY set), instead of waiting for another process to open the named pipe.
O_APPEND	causes all writes to append data to the end of the file.
O_CREAT	create the file if it does not exist. Mode specifies permissions as in creat system call. The flag has no meaning if the file already exists.
O_TRUNC	Truncate length of file to 0.
O_EXCL	Fail the open call if this bit and O_CREAT are set and file exists. This is a so-called exclusive open.

Open returns a file descriptor for use in other system calls.

pause

```
pause()
```

Pause suspends the execution of the calling process until it receives a signal.

pipe

```
pipe(fildes)
int fildes[2];
```

Pipe returns a read and write file descriptor (*fildes[0]* and *fildes[1]*, respectively). Data is transmitted through a pipe in first-in-first-out order; data cannot be read twice.

plock

```
#include <sys/lock.h>

plock(op)
int op;
```

Plock locks and unlocks process regions in memory according to the value of *op*:

PROCLOCK	lock text and data regions in memory.
TXTLOCK	lock text region in memory.
DATLOCK	lock data region in memory.
UNLOCK	remove locks for all regions.

profil

```
profil(buf, size, offset, scale)
char *buf;
int size, offset, scale;
```

Profil requests that the kernel give an execution profile of the process. *Buf* is an array in the process that accumulates frequency counts of execution in different addresses of the process. *Size* is the size of the buf array, *offset* is the starting address in the process that should be profiled, and *scale* is a scaling factor.

ptrace

```
ptrace(cmd, pid, addr, data)
int cmd, pid, addr, data;
```

Ptrace allows a process to trace the execution of another process, *pid*, according to the value of *cmd*.

0	enable child for tracing (called by child).
1,2	return word at location addr in traced process pid.
3	return word from offset addr in traced process u area.
4,5	write value of data into location addr in traced process.
6	write value of data into offset addr in u area.
7	cause traced process to resume execution.
8	cause traced process to exit.
9	machine dependent – set bit in PSW for single-stepping execution.

read

```
read(fildes, buf, size)
int fildes,
char *buf;
int size;
```

Read reads up to size bytes from the file *fildes* into the user buffer *buf*. *Read* returns the number of bytes it read. For special devices and pipes, *read* returns immediately if *O_NDELAY* was set in open and no data is available for return.

semctl

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
```

```
semctl(id, num, cmd, arg)
int id, num, cmd;
union semun {
        int val;
        struct semid_ds *buf;
        ushort *array;
} arg;
```

Semctl does the specified *cmd* on the semaphore queue indicated by *id*.

GETVAL	return the value of the semaphore whose index is num.
SETVAL	set the value of the semaphore whose index is num to arg.val.
GETPID	return value of last PID that did a semop on the semaphore whose index is num.
GETNCNT	return number of processes waiting for semaphore value to become positive.
GETZCNT	return number of processes waiting for semaphore value to become 0.
GETALL	return values of all semaphores into array arg.array.
SETALL	set values of all semaphores according to array arg.array.
IPC_STAT	read structure of semaphore header for id into arg.buf.
IPC_SET	set sem_perm.uid, sem_per.gid, and sem_perm.mode (low-order 9 bits) according to arg.buf.
IPC_RMID	remove the semaphores associated with id.

Num gives the number of semaphores in the set to be processed. The structure *semid_ds* is defined by:

```
struct semid_ds {
    struct ipc_perm    sem_perm;    /* permission struct */
    int *              pad;         /* used by system */
    ushort             sem_nsems;   /* number of semaphores in set */
    time_t             sem_otime;   /* last semop operation time */
    time_t             sem_ctime;   /* last change time */
};
```

The structure *ipc_perm* is the same as defined in *msgctl*.

semget

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

semget(key, nsems, flag)
key_t key;
int nsems, flag;
```

Semget creates an array of semaphores, corresponding to *key*. *Key* and *flag* take on the same meaning as they do in *msgget*.

semop

```
semop(id, ops, num)
int id, num;
struct sembuf **ops;
```

Semop does the set of semaphore operations in the array of structures *ops*, to the set of semaphores identified by *id*. *Num* is the number of entries in *ops*. The structure of *sembuf* is:

```
struct sembuf {
        short   sem_num;    /* semaphore number */
        short   sem_op;     /* semaphore operation */
        short   sem_flg;    /* flag */
};
```

Sem_num specifies the index in the semaphore array for the particular operation, and *sem_flg* specifies flags for the operation. The operations *sem_op* for semaphores are:

negative	if sum of semaphore value and sem_op >= 0, add sem_op to to semaphore value. Otherwise, sleep, as per flag.
positive	add sem_op to semaphore value.
zero	continue, if semaphore value is 0. Otherwise, sleep as per flag.

If *IPC_NOWAIT* is set in *sem_flg* for a particular operation, *semop* returns immediately for those occasions it would have slept. If the *SEM_UNDO* flag is set, the operation is subtracted from a running sum of such values. When the process exits, this sum is added to the value of the semaphore. *Semop* returns the value of the last semaphore operation in *ops* at the time of the call.

setpgrp

```
setpgrp()
```

Setpgrp sets the process group ID of the calling process to its process ID and returns the new value.

setuid

```
setuid(uid)
int uid;
```

```
setgid(gid)
int gid;
```

Setuid sets the real and effective user ID of the calling process. If the effective user ID of the caller is superuser, *setuid* resets the real and effective user IDs. Otherwise, if its real user ID equals *uid*, *setuid* resets the effective user ID to *uid*. Finally, if its saved user ID (set by executing a setuid program in *exec*) equals *uid*, *setuid* resets the effective user ID to *uid*. *Setgid* works the same way for real and effective group IDs.

shmctl

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

shmctl(id, cmd, buf)
int id, cmd;
struct shmid_ds *buf;
```

Shmctl does various control operations on the shared memory region identified by *id*. The structure *shmid_ds* is defined by:

```
struct shmid_ds {
        struct ipc_perm   shm_perm;    /* permission struct */
        int               shm_segsz;   /* size of segment */
        int *             pad1;        /* used by system */
        ushort            shm_lpid;    /* pid of last operation */
        ushort            shm_cpid;    /* pid of creator */
        ushort            shm_nattch;  /* number currently attached */
        short             pad2;        /* used by system */
        time_t            shm_atime;   /* last attach time */
        time_t            shm_dtime;   /* last detach time */
        time_t            shm_ctime;   /* last change time */
};
```

The operations are:

IPC_STAT	read values of shared memory header for id into buf.
IPC_SET	set shm_perm.uid, shm_perm.gid, and shm_perm.mode (9 low-order bits) in shared memory header according to values in buf.
IPC_RMID	remove shared memory region for id.

shmget

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
```

```
shmget(key, size, flag)
key_t key;
int size, flag;
```

Shmget accesses or creates a shared memory region of *size* bytes. The parameters *key* and *flag* have the same meaning as they do for *msgget*.

shmop

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

shmat(id, addr, flag)
int id, flag;
char *addr;

shmdt(addr)
char *addr;
```

Shmat attaches the shared memory region identified by *id* to the address space of a process. If *addr* is 0, the kernel chooses an appropriate address to attach the region. Otherwise, it attempts to attach the region at the specified address. If the *SHM_RND* bit is on in *flag*, the kernel rounds off the address, if necessary. *Shmat* returns the address where the region is attached.

Shmdt detaches the shared memory region previously attached at *addr*.

signal

```
#include <signal.h>

signal(sig, function)
int sig;
void (*func)();
```

Signal allows the calling process to control signal processing. The values of *sig* are:

SIGHUP	hangup
SIGINT	interrupt
SIGQUIT	quit
SIGILL	illegal instruction
SIGTRAP	trace trap
SIGIOT	IOT instruction
SIGEMT	EMT instruction
SIGFPE	floating point exception
SIGKILL	kill

SIGBUS	bus error
SIGSEGV	segmentation violation
SIGSYS	bad argument in system call
SIGPIPE	write on a pipe with no reader
SIGALRM	alarm
SIGTERM	software termination
SIGUSR1	user-defined signal
SIGUSR2	second user-defined signal
SIGCLD	death of child
SIGPWR	power failure

The interpretation of *function* is as follows:

SIG_DFL	default operation. For all signals except SIGPWR and SIGCLD, process terminates. It creates a core image for signals SIGQUIT, SIGILL, SIGTRAP, SIGIOT, SIGEMT, SIGFPE, SIGBUS, SIGSEGV, and SIGSYS.
SIG_IGN	ignore the occurrence of the signal.
function	an address of a procedure in the process. The kernel arranges to call the function with the signal number as argument when it returns to user mode. The kernel automatically resets the value of the signal handler to SIG_DFL for all signals except SIGILL, SIGTRAP, and SIGPWR. A process cannot catch SIGKILL signals.

stat

```
stat(filename, statbuf)
char *filename;
struct stat *statbuf;

fstat(fd, statbuf)
int fd;
struct stat *statbuf;
```

Stat returns status information about the specified file. *Fstat* does the same for the open file whose descriptor is *fd*. The structure of *statbuf* is:

```
struct stat {
        dev_t   st_dev;     /* device number for dev containing file */
        ino_t   st_ino;     /* inode number */
        ushort  st_mode;    /* file type (see mknod) and perms (see chmod) */
        short   st_nlink;   /* number of links for file */
        ushort  st_uid;     /* user ID of file's owner */
        ushort  st_gid;     /* group ID of file's group */
        dev_t   st_rdev;    /* major and minor device numbers */
        off_t   st_size;    /* size in bytes */
```

```
        time_t    st_atime;    /* time of last access */
        time_t    st_mtime;    /* time of last modification */
        time_t    st_ctime;    /* time of last status change */
};
```

stime

```
stime(tptr)
long *tptr;
```

Stime sets the system time and date, according to the value pointed to by *tptr*. Times are specified in seconds since 00:00:00 January, 1, 1970, GMT.

sync

```
sync()
```

Sync flushes file system data in system buffers onto disk.

time

```
time(tloc)
long *tloc;
```

Time returns the number of seconds since 00:00:00 January 1, 1970, GMT. If *tloc* is not 0, it will contain the return value, too.

times

```
#include <sys/types.h>
#include <sys/times.h>

times(tbuf)
struct tms *tbuf;
```

Times returns the elapsed real time in clock ticks from an arbitrary fixed time in the recent past, and fills *tbuf* with accounting information:

```
struct tms {
        time_t    tms_utime;    /* CPU time spent in user mode */
        time_t    tms_stime;    /* CPU time spent in kernel mode */
        time_t    tms_cutime;   /* Sum of tms_utime and tms_cutime of children */
        time_t    tms_sutime;   /* Sum of tms_stime and tms_sutime of children */
};
```

ulimit

```
ulimit(cmd, limit)
int cmd;
long limit;
```

Ulimit allows a process to set various limits according to the value of *cmd*:

1 return maximum file size (in 512 byte blocks) the process can write
2 set maximum file size to limit.
3 return maximum possible break value (highest possible address in data region).

umask

```
umask(mask)
int mask;
```

Set the file mode creation *mask* and return the old value. When creating a file, permissions are turned off if the corresponding bits in *mask* are set.

umount

```
umount(specialfile)
char *specialfile;
```

Unmount the file system in the block special device *specialfile*.

uname

```
#include <sys/utsname.h>

uname(name)
struct utsname *name;
```

Uname returns system-specific information according to the following structure:

```
struct utsname {
        char    sysname[9];     /* name */
        char    nodename[9];    /* network node name */
        char    release[9];     /* system version information */
        char    version[9];     /* more version information */
        char    machine[9];     /* hardware */
};
```

unlink

```
unlink(filename)
char *filename;
```

Remove the directory entry for the indicated file.

ustat

```
#include <sys/types.h>
#include <ustat.h>

ustat(dev, ubuf)
int dev;
struct ustat *ubuf;
```

Ustat returns statistics about the file system identified by *dev* (the major and minor number). The structure *ustat* is defined by:

```
struct ustat {
        daddr_t     f_tfree;        /* number of free blocks */
        ino_t       f_tinode;       /* number of free inodes */
        char        f_fname[6];     /* filsys name */
        char        f_fpack[6];     /* filsys pack name */
};
```

utime

```
#include <sys/types.h>

utime(filename, times)
char *filename;
struct utimbuf *times;
```

Utime sets the access and modification times of the specified file according to the value of *times*. If 0, the current time is used. Otherwise, *times* points to the following structure:

```
struct utimbuf {
        time_t      axtime;     /* access time */
        time_t      modtime;    /* modification time */
};
```

All times are measured from 00:00:00 January 1, 1970 GMT.

wait

```
wait(wait_stat)
int *wait_stat;
```

Wait causes the process to sleep until it discovers a child process that had exited or a process asleep in trace mode. If *wait_stat* is not 0, it points to an address that contains status information on return from the call. Only the 16 low order bits are written. If wait returns because it found a child process that had exited, the low order 8 bits are 0, and the high order 8 bits contain the low order 8 bits the child process had passed as a parameter to *exit*. If the child exited because of a signal, the high order 8 bits are 0, and the low order 8 bits contain the signal number. In addition, bit 0200 is set if core was dumped. If *wait* returns because it found a traced process, the high order 8 bits (of the 16 bits) contain the signal number that caused it to stop, and the low order 8 bits contain octal 0177.

write

```
write(fd, buf, count)
int fd, count;
char *buf;
```

Write writes *count* bytes of data from user address *buf* to the file whose descriptor is *fd*.