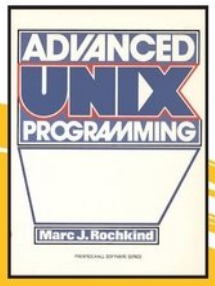
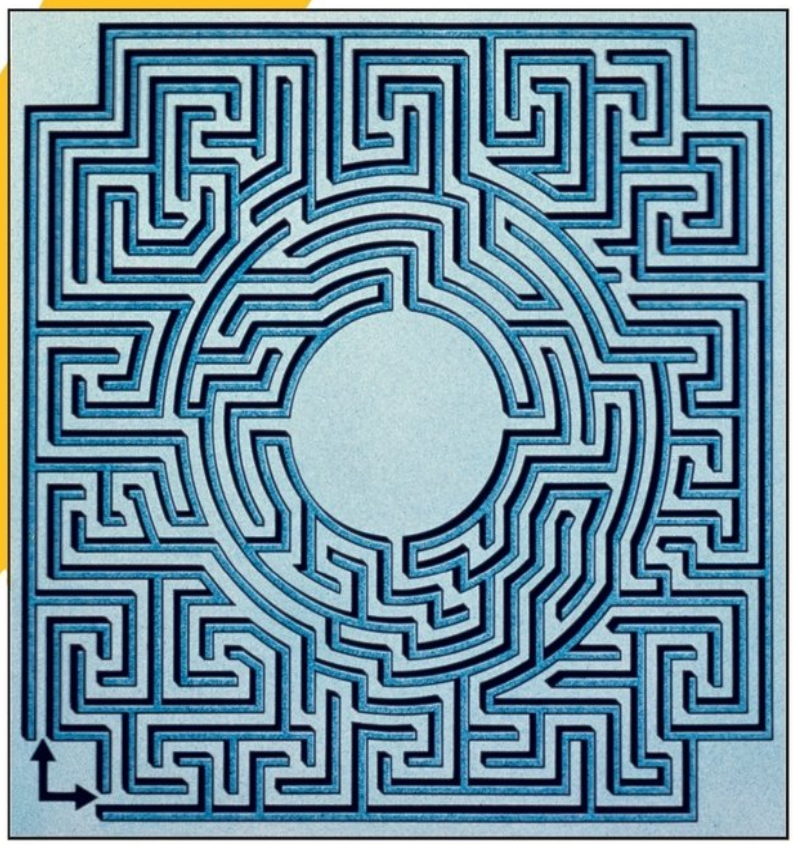
Updated Classic!



Advanced UNIX® Programming

SECOND EDITION



MARC J. ROCHKIND

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

Advanced UNIX Programming

Addison-Wesley Professional Computing Series

Brian W. Kernighan, Consulting Editor

Matthew H. Austern, *Generic Programming and the STL: Using and Extending the C++ Standard Template Library*
David R. Butenhof, *Programming with POSIX® Threads*
Brent Callaghan, *NFS Illustrated*
Tom Cargill, *C++ Programming Style*
William R. Cheswick/Steven M. Bellovin/Aviel D. Rubin, *Firewalls and Internet Security, Second Edition: Repelling the Wily Hacker*
David A. Curry, *UNIX® System Security: A Guide for Users and System Administrators*
Stephen C. Dewhurst, *C++ Gotchas: Avoiding Common Problems in Coding and Design*
Dan Farmer/Wietse Venema, *Forensic Discovery*
Erich Gamma/Richard Helm/Ralph Johnson/John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*
Erich Gamma/Richard Helm/Ralph Johnson/John Vlissides, *Design Patterns CD: Elements of Reusable Object-Oriented Software*
Peter Haggar, *Practical Java™ Programming Language Guide*
David R. Hanson, *C Interfaces and Implementations: Techniques for Creating Reusable Software*
Mark Harrison/Michael McLennan, *Effective Tcl/Tk Programming: Writing Better Programs with Tcl and Tk*
Michi Henning/Steve Vinoski, *Advanced CORBA® Programming with C++*
Brian W. Kernighan/Rob Pike, *The Practice of Programming*
S. Keshav, *An Engineering Approach to Computer Networking: ATM Networks, the Internet, and the Telephone Network*
John Lakos, *Large-Scale C++ Software Design*
Scott Meyers, *Effective C++ CD: 85 Specific Ways to Improve Your Programs and Designs*
Scott Meyers, *Effective C++, Third Edition: 55 Specific Ways to Improve Your Programs and Designs*
Scott Meyers, *More Effective C++: 35 New Ways to Improve Your Programs and Designs*
Scott Meyers, *Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library*
Robert B. Murray, *C++ Strategies and Tactics*
David R. Musser/Gillmer J. Derge/Atul Saini, *STL Tutorial and Reference Guide, Second Edition: C++ Programming with the Standard Template Library*
John K. Ousterhout, *Tcl and the Tk Toolkit*
Craig Partridge, *Gigabit Networking*
Radia Perlman, *Interconnections, Second Edition: Bridges, Routers, Switches, and Internetworking Protocols*
Stephen A. Rago, *UNIX® System V Network Programming*
Eric S. Raymond, *The Art of UNIX Programming*
Marc J. Rochkind, *Advanced UNIX Programming, Second Edition*
Curt Schimmel, *UNIX® Systems for Modern Architectures: Symmetric Multiprocessing and Caching for Kernel Programmers*
W. Richard Stevens, *TCP/IP Illustrated, Volume 1: The Protocols*
W. Richard Stevens, *TCP/IP Illustrated, Volume 3: TCP for Transactions, HTTP, NNTP, and the UNIX® Domain Protocols*
W. Richard Stevens/Bill Fenner/Andrew M. Rudoff, *UNIX Network Programming Volume 1, Third Edition: The Sockets Networking API*
W. Richard Stevens/Stephen A. Rago, *Advanced Programming in the UNIX® Environment, Second Edition*
W. Richard Stevens/Gary R. Wright, *TCP/IP Illustrated Volumes 1-3 Boxed Set*
John Viega/Gary McGraw, *Building Secure Software: How to Avoid Security Problems the Right Way*
Gary R. Wright/W. Richard Stevens, *TCP/IP Illustrated, Volume 2: The Implementation*
Ruixi Yuan/W. Timothy Strayer, *Virtual Private Networks: Technologies and Solutions*

Visit www.awprofessional.com/series/professionalcomputing for more information about these titles.

Advanced UNIX Programming

Second Edition

Marc J. Rochkind

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

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and Addison-Wesley was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

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

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

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

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

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

Copyright © 2004 by Pearson Education, Inc.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher. Printed in the United States of America. Published simultaneously in Canada.

For information on obtaining permission for use of material from this work, please submit a written request to:

Pearson Education, Inc.
Rights and Contracts Department
75 Arlington Street, Suite 300
Boston, MA 02116
Fax: (617) 848-7047

ISBN 0-13-141154-3

Text printed in the United States on recycled paper at RR Donnelley Crawfordsville in Crawfordsville, Indiana.

6th Printing September 2008

For Claire and Gillian

This page intentionally left blank

Contents

Preface **xi**

Chapter 1 Fundamental Concepts **1**

1.1 A Whirlwind Tour of UNIX and Linux 1
1.2 Versions of UNIX 16
1.3 Using System Calls 19
1.4 Error Handling 24
1.5 UNIX Standards 38
1.6 Common Header File 55
1.7 Dates and Times 56
1.8 About the Example Code 67
1.9 Essential Resources 68

Chapter 2 Basic File I/O 71

2.1 Introduction to File I/O 71
2.2 File Descriptors and Open File Descriptions 72
2.3 Symbols for File Permission Bits 75
2.4 `open` and `creat` System Calls 76
2.5 `umask` System Call 86
2.6 `unlink` System Call 86
2.7 Creating Temporary Files 88
2.8 File Offsets and `O_APPEND` 90
2.9 `write` System Call 92
2.10 `read` System Call 96
2.11 `close` System Call 97
2.12 User Buffered I/O 98
2.13 `lseek` System Call 105
2.14 `pread` and `pwrite` System Calls 108

2.15 `readv` and `writev` System Calls 110
2.16 Synchronized I/O 114
2.17 `truncate` and `ftruncate` System Calls 119

Chapter 3 Advanced File I/O 123

3.1 Introduction 123
3.2 Disk Special Files and File Systems 123
3.3 Hard and Symbolic Links 137
3.4 Pathnames 144
3.5 Accessing and Displaying File Metadata 147
3.6 Directories 158
3.7 Changing an I-Node 181
3.8 More File-Manipulation Calls 185
3.9 Asynchronous I/O 189

Chapter 4 Terminal I/O 203

4.1 Introduction 203
4.2 Reading from a Terminal 204
4.3 Sessions and Process Groups (Jobs) 224
4.4 ioctl System Call 232
4.5 Setting Terminal Attributes 233
4.6 Additional Terminal-Control System Calls 245
4.7 Terminal-Identification System Calls 248
4.8 Full-Screen Applications 250
4.9 STREAMS I/O 255
4.10 Pseudo Terminals 256

Chapter 5 Processes and Threads 277

5.1 Introduction 277
5.2 Environment 277
5.3 `exec` System Calls 284
5.4 Implementing a Shell (Version 1) 292
5.5 `fork` System Call 296
5.6 Implementing a Shell (Version 2) 300
5.7 `exit` System Calls and Process Termination 301

5.8 `wait`, `waitpid`, and `waitid` System Calls 304
5.9 Signals, Termination, and Waiting 313
5.10 Implementing a Shell (Version 3) 314
5.11 Getting User and Group IDs 315
5.12 Setting User and Group IDs 317
5.13 Getting Process IDs 319
5.14 `chroot` System Call 319
5.15 Getting and Setting the Priority 320
5.16 Process Limits 322
5.17 Introduction to Threads 329
5.18 The Blocking Problem 350

Chapter 6 Basic Interprocess Communication 361

6.1 Introduction 361
6.2 Pipes 362
6.3 dup and dup2 System Calls 371
6.4 A Real Shell 376
6.5 Two-Way Communication with Unidirectional Pipes 390
6.6 Two-Way Communication with Bidirectional Pipes 399

Chapter 7 Advanced Interprocess Communication 405

7.1 Introduction 405
7.2 FIFOs, or Named Pipes 406
7.3 An Abstract Simple Messaging Interface (SMI) 414
7.4 System V IPC (Interprocess Communication) 428
7.5 System V Message Queues 434
7.6. POSIX IPC 442
7.7 POSIX Message Queues 445
7.8 About Semaphores 458
7.9 System V Semaphores 460
7.10 POSIX Semaphores 469
7.11 File Locking 477
7.12 About Shared Memory 488
7.13 System V Shared Memory 489
7.14 POSIX Shared Memory 504
7.15 Performance Comparisons 515

Chapter 8 Networking and Sockets 519

8.1 Socket Basics 520
8.2 Socket Addresses 533
8.3 Socket Options 544
8.4 Simple Socket Interface (SSI) 549
8.5 Socket Implementation of SMI 563
8.6 Connectionless Sockets 567
8.7 Out-of-Band Data 577
8.8 Network Database Functions 578
8.9 Miscellaneous System Calls 593
8.10 High-Performance Considerations 597

Chapter 9 Signals and Timers 601

9.1 Signal Basics 601
9.2 Waiting for a Signal 624
9.3 Miscellaneous Signal System Calls 634
9.4 Deprecated Signal System Calls 635
9.5 Realtime Signals Extension (RTS) 637
9.6 Global Jumps 648
9.7 Clocks and Timers 651

Appendix A Process Attributes 667

Appendix B Ux: A C++ Wrapper for Standard UNIX Functions 673

Appendix C Jtux: A Java/Jython Interface to Standard UNIX Functions 677

Appendix D Alphabetical and Categorical Function Lists 685

References 703

Index 705

Preface

This book updates the 1985 edition of *Advanced UNIX Programming* to cover a few changes that have occurred in the last eighteen years. Well, maybe "few" isn't the right word! And "updates" isn't right either. Indeed, aside from a sentence here and there, this book is all new. The first edition included about 70 system calls; this one includes about 300. And none of the UNIX standards and implementations discussed in this book—POSIX, Solaris, Linux, FreeBSD, and Darwin (Mac OS X)—were even around in 1985. A few sentences from the 1985 Preface, however, are among those that I can leave almost unchanged:

> The subject of this book is UNIX system calls—the interface between the UNIX kernel and the user programs that run on top of it. Those who interact only with commands, like the shell, text editors, and other application programs, may have little need to know much about system calls, but a thorough knowledge of them is essential for UNIX programmers. System calls are the *only* way to access kernel facilities such as the file system, the multitasking mechanisms, and the interprocess communication primitives.
>
> System calls define what UNIX is. Everything else—subroutines and commands—is built on this foundation. While the novelty of many of these higher-level programs has been responsible for much of UNIX's renown, they could as well have been programmed on any modern operating system. When one describes UNIX as elegant, simple, efficient, reliable, and portable, one is referring not to the commands (some of which are none of these things), but to the kernel.

That's all still true, except that, regrettably, the programming interface to the kernel is no longer elegant or simple. In fact, because UNIX development has splintered into many directions over the last couple of decades, and because the principal standards organization, The Open Group, sweeps up almost everything that's out there (1108 functions altogether), the interface is clumsy, inconsistent, redundant, error-prone, and confusing. But it's still efficient, reliably implemented, and portable, and that's why UNIX and UNIX-like systems are so successful. Indeed, the UNIX system-call interface is the only widely implemented portable one we have and are likely to have in our lifetime.

To sort things out, it's not enough to have complete documentation, just as the Yellow Pages isn't enough to find a good restaurant or hotel. You need a guide that tells you what's good and bad, not just what exists. That's the purpose of this book, and why it's different from most other UNIX programming books. I tell you not only how to use the system calls, but also which ones to stay away from because they're unnecessary, obsolete, improperly implemented, or just plain poorly designed.

Here's how I decided what to include in this book: I started with the 1108 functions defined in Version 3 of the Single UNIX Specification and eliminated about 590 Standard C and other library functions that aren't at the kernel-interface level, about 90 POSIX Threads functions (keeping a dozen of the most important), about 25 accounting and logging functions, about 50 tracing functions, about 15 obscure and obsolete functions, and about 40 functions for scheduling and other things that didn't seem to be generally useful. That left exactly 307 for this book. (See Appendix D for a list.) Not that the 307 are all good—some of them are useless, or even dangerous. But those 307 are the ones you need to know.

This book doesn't cover kernel implementation (other than some basics), writing device drivers, C programming (except indirectly), UNIX commands (shell, vi, emacs, etc.), or system administration.

There are nine chapters: Fundamental Concepts, Basic File I/O, Advanced File I/O, Terminal I/O, Processes and Threads, Basic Interprocess Communication, Advanced Interprocess Communication, Networking and Sockets, and Signals and Timers. Read all of Chapter 1, but then feel free to skip around. There are lots of cross-references to keep you from getting lost.

Like the first edition, this new book includes thousands of lines of example code, most of which are from realistic, if simplified, applications such as a shell, a full-screen menu system, a Web server, and a real-time output recorder. The examples are all in C, but I've provided interfaces in Appendices B and C so you can program in C++, Java, or Jython (a variant of Python) if you like.

The text and example code are just resources; you really learn UNIX programming by doing it. To give you something to do, I've included exercises at the end of each chapter. They range in difficulty from questions that can be answered in a few sentences to simple programming problems to semester-long projects.

I used four UNIX systems for nuts-and-bolts research and to test the examples: Solaris 8, SuSE Linux 8 (2.4 kernel), FreeBSD 4.6, and Darwin (the Mac OS X

kernel) 6.8. I kept the source on the FreeBSD system, mounted on the others with NFS or Samba.[1]

I edited the code with TextPad on a Windows system and accessed the four test systems with Telnet or SSH (PuTTY) or with the X Window System (XFree86 and Cygwin). Having the text editor and the four Telnet/SSH/Xterm windows open on the same screen turned out to be incredibly convenient, because it takes only a few minutes to write some code and check it out on the four systems. In addition, I usually had one browser window open to the Single UNIX Specification and one to Google, and another window running Microsoft Word for editing the book. With the exception of Word, which is terrible for big documents like books (crashes, mixed-up styles, weak cross-referencing, flakey document-assembly), all of these tools worked great.[2] I used Perl and Python for various things like extracting code samples and maintaining the database of system calls.

All of the example code (free open source), errata, and more is on the book Web site at *www.basepath.com/aup.*

I'd like to thank those who reviewed the drafts or who otherwise provided technical assistance: Tom Cargill, Geoff Clare, Andrew Gierth, Andrew Josey, Brian Kernighan, Barry Margolin, Craig Partridge, and David Schwartz. And, special thanks to one dedicated, meticulous reviewer who asked to remain anonymous. Naturally, none of these folks is to be blamed for any mistakes you find—I get full credit for those.

I'd also like to thank my editor, Mary Franz, who suggested this project a year or so ago. Luckily, she caught me at a time when I was looking into Linux in depth and started to get excited about UNIX all over again. Reminds me a bit of 1972….

I hope you enjoy the book! If you find anything wrong, or if you port the code to any new systems, or if you just want to share your thoughts, please email me at aup@basepath.com.

Marc J. Rochkind
Boulder, Colorado
April, 2004

1. The four systems are running on various old PCs I've collected over the years and on a Mac I bought on eBay for $200. I had no particular reason for using SuSE Linux and have since switched that machine over to RedHat 9.

2. I could have used any of the systems as my base. Windows turned out to be convenient because my big LCD monitor is attached to that system and because I like TextPad (*www.textpad.com*). Information on PuTTY is at *www.chiark.greenend.org.uk/~sgtatham/putty/.* (Google "PuTTY" if that link doesn't work.)

This page intentionally left blank

1 Fundamental Concepts

1.1 A Whirlwind Tour of UNIX and Linux

This section takes you on a quick tour of the facilities provided by the UNIX and Linux kernels. I won't deal with the user programs (commands) that normally come with UNIX, such as `ls`, `vi`, and `grep`. A discussion of these is well outside the scope of this book. And I won't say much about the internals of the kernel (such as how the file system is implemented) either. (From now on, whenever I say UNIX, I mean Linux, too, unless I say otherwise.)

This tour is meant to be a refresher. I'll use terms such as *process* before defining them, because I assume you already know roughly what they mean. If too much sounds new to you, you may want to become more familiar with UNIX before proceeding. (If you don't know what a process is, you definitely need to get more familiar!) There are lots of introductory UNIX books to start with. Two good ones are *The UNIX Programming Environment* [Ker1984] and *UNIX for the Impatient* [Abr1996] (Chapter 2 is a great introduction).[1]

1.1.1 Files

There are several kinds of UNIX files: regular files, directories, symbolic links, special files, named pipes (FIFOs), and sockets. I'll introduce the first four here and the last two in Section 1.1.7.

1.1.1.1 Regular Files

Regular files contain bytes of data, organized into a linear array. Any byte or sequence of bytes may be read or written. Reads and writes start at a byte loca-

1. You'll find the References at the end of the book.

tion specified by the *file offset,* which can be set to any value (even beyond the end of the file). Regular files are stored on disk.

It isn't possible to insert bytes into the middle of a file (spreading the file apart), or to delete bytes from the middle (closing the gap). As bytes are written onto the end of a file, it gets bigger, one byte at a time. A file can be shrunk or enlarged to any length, discarding bytes or adding bytes of zeroes.

Two or more processes can read and write the same file concurrently. The results depend on the order in which the individual I/O requests occur and are in general unpredictable. To maintain order, there are file-locking facilities and *semaphores*, which are system-wide flags that processes can test and set (more on them in Section 1.1.7).

Files don't have names; they have numbers called *i-numbers.* An i-number is an index into an array of *i-nodes,* kept at the front of each region of disk that contains a UNIX file system. Each i-node contains important information about one file. Interestingly, this information doesn't include either the name or the data bytes. It does include the following: type of file (regular, directory, socket, etc.); number of links (to be explained shortly); owner's user and group ID; three sets of access permissions—for the owner, the group, and others; size in bytes; time of last access, last modification, and status change (when the i-node itself was last modified); and, of course, pointers to disk blocks containing the file's contents.

1.1.1.2 Directories and Symbolic Links

Since it's inconvenient to refer to files by i-number, *directories* are provided to allow names to be used. In practice, a directory is almost always used to access a file.

Each directory consists, conceptually, of a two-column table, with a name in one column and its corresponding i-number in the other column. A name/i-node pair is called a *link.* When the UNIX kernel is told to access a file by name, it automatically looks in a directory to find the i-number. Then it gets the corresponding i-node, which contains more information about the file (such as who can access it). If the data itself is to be accessed, the i-node tells where to find it on the disk.

Directories, which are almost like regular files, occupy an i-node and have data. Therefore, the i-node corresponding to a particular name in a directory could be the i-node of another directory. This allows users to arrange their files into the hierarchical structure that's familiar to users of UNIX. A *path* such as `memo/july/smith` instructs the kernel to get the i-node of the *current directory* to

locate its data bytes, find `memo` among those data bytes, take the corresponding i-number, get that i-node to locate the `memo` directory's data bytes, find `july` among those, take the corresponding i-number, get the i-node to locate the `july` directory's data bytes, find `smith`, and, finally, take the corresponding i-node, the one associated with `memo/july/smith`.

In following a *relative path* (one that starts with the current directory), how does the kernel know where to start? It simply keeps track of the i-number of the current directory for each process. When a process changes its current directory, it must supply a path to the new directory. That path leads to an i-number, which then is saved as the i-number of the new current directory.

An *absolute path* begins with a / and starts with the *root* directory. The kernel simply reserves an i-number (2, say) for the root directory. This is established when a file system is first constructed. There is a system call to change a process's root directory (to an i-number other than *2*).

Because the two-column structure of directories is used directly by the kernel (a rare case of the kernel caring about the contents of files), and because an invalid directory could easily destroy an entire UNIX system, a program (even if run by the superuser) cannot write a directory as if it were a regular file. Instead, a program manipulates a directory by using a special set of system calls. After all, the only legal writing actions are to add or remove a link.

It is possible for two or more links, in the same or different directories, to refer to the same i-number. This means that the same file may have more than one name. There is no ambiguity when accessing a file by a given path, since only one i-number will be found. It might have been found via another path also, but that's irrelevant. When a link is removed from a directory, however, it isn't immediately clear whether the i-node and the associated data bytes can be thrown away too. That is why the i-node contains a link count. Removing a link to an i-node merely decrements the link count; when the count reaches zero, the kernel discards the file.

There is no structural reason why there can't be multiple links to directories as well as to regular files. However, this complicates the programming of commands that scan the entire file system, so most kernels outlaw it.

Multiple links to a file using i-numbers work only if the links are in the same file system, as i-numbers are unique only within a file system. To get around this, there are also *symbolic links*, which put the path of the file to be linked to in the

data part of an actual file. This is more overhead than just making a second directory link somewhere, but it's more general. You don't read and write these symbolic-link files, but instead use special system calls just for symbolic links.

1.1.1.3 Special Files

A *special file* is typically some type of *device* (such as a CD-ROM drive or communications link).[2]

There are two principal kinds of device special files: block and character. *Block special files* follow a particular model: The device contains an array of fixed-size blocks (say, 4096 bytes each), and a pool of kernel *buffers* are used as a cache to speed up I/O. *Character special files* don't have to follow any rules at all. They might do I/O in very small chunks (characters) or very big chunks (disk tracks), and so they're too irregular to use the buffer cache.

The same physical device could have both block and character special files, and, in fact, this is usually true for disks. Regular files and directories are accessed by the file-system code in the kernel via a block special file, to gain the benefits of the buffer cache. Sometimes, primarily in high-performance applications, more direct access is needed. For instance, a database manager can bypass the file system entirely and use a character special file to access the disk (but not the same area that's being used by the file system). Most UNIX systems have a character special file for this purpose that can directly transfer data between a process's address space and the disk using *direct memory access* (*DMA*), which can result in orders-of-magnitude better performance. More robust error detection is another benefit, since the indirectness of the buffer cache tends to make error detection difficult to implement.

A special file has an i-node, but there aren't any data bytes on disk for the i-node to point to. Instead, that part of the i-node contains a *device number*. This is an index into a table used by the kernel to find a collection of subroutines called a *device driver*.

When a system call is executed to perform an operation on a special file, the appropriate device driver subroutine is invoked. What happens then is entirely up to the designer of the device driver; since the driver runs in the kernel, and not as

2. Sometimes named pipes are considered special files, too, but we'll consider them a category of their own.

a user process, it can access—and perhaps modify—any part of the kernel, any user process, and any registers or memory of the computer itself. It is relatively easy to add new device drivers to the kernel, so this provides a hook with which to do many things besides merely interfacing to new kinds of I/O devices. It's the most popular way to get UNIX to do something its designers never intended it to do. Think of it as the approved way to do something wild.

1.1.2 Programs, Processes, and Threads

A *program* is a collection of *instructions* and *data* that is kept in a regular file on disk. In its i-node the file is marked executable, and the file's contents are arranged according to rules established by the kernel. (Another case of the kernel caring about the contents of a file.)

Programmers can create executable files any way they choose. As long as the contents obey the rules and the file is marked executable, the program can be run. In practice, it usually goes like this: First, the source program, in some programming language (C or C++, say), is typed into a regular file, often referred to as a *text file,* because it's arranged into text lines. Next, another regular file, called an *object file,* is created that contains the machine-language translation of the source program. This job is done by a compiler or assembler (which are themselves programs). If this object file is complete (no missing subroutines), it is marked executable and may be run as is. If not, the *linker* (sometimes called a "loader" in UNIX jargon) is used to bind this object file with others previously created, possibly taken from collections of object files called *libraries.* Unless the linker couldn't find something it was looking for, its output is complete and executable.[3]

In order to run a program, the kernel is first asked to create a new *process,* which is an environment in which a *program* executes. A process consists of three segments: *instruction segment*,[4] *user data segment,* and *system data segment.* The program is used to initialize the instructions and user data. After this initialization, the process begins to deviate from the program it is running. Although modern programmers don't normally modify instructions, the data does get modi-

3. This isn't how interpretive languages like Java, Perl, Python, and shell scripts work. For them, the executable is an interpreter, and the program, even if compiled into some intermediate code, is just data for the interpreter and isn't something the UNIX kernel ever sees or cares about. The kernel's customer is the interpreter.

4. In UNIX jargon, the instruction segment is called the "text segment," but I'll avoid that confusing term.

fied. In addition, the process may acquire resources (more memory, open files, etc.) not present in the program.

While the process is running, the kernel keeps track of its *threads,* each of which is a separate flow of control through the instructions, all potentially reading and writing the same parts of the process's data. (Each thread has its own stack, however.) When you're programming, you start with one thread, and that's all you get unless you execute a special system call to create another. So, beginners can think of a process as being single-threaded.[5]

Several concurrently running processes can be initialized from the same program. There is no functional relationship, however, between these processes. The kernel might be able to save memory by arranging for such processes to share instruction segments, but the processes involved can't detect such sharing. By contrast, there is a strong functional relationship between threads in the same process.

A process's *system data* includes attributes such as current directory, open file descriptors, accumulated CPU time, and so on. A process can't access or modify its system data directly, since it is outside of its address space. Instead, there are various system calls to access or modify attributes.

A process is created by the kernel on behalf of a currently executing process, which becomes the *parent* of the new *child* process. The child inherits most of the parent's system-data attributes. For example, if the parent has any files open, the child will have them open too. Heredity of this sort is absolutely fundamental to the operation of UNIX, as we shall see throughout this book. This is different from a thread creating a new thread. Threads in the same process are equal in most respects, and there's no inheritance. All threads have equal access to all data and resources, not copies of them.

1.1.3 Signals

The kernel can send a *signal* to a process. A signal can be originated by the kernel itself, sent from a process to itself, sent from another process, or sent on behalf of the user.

5. Not every version of UNIX supports multiple threads. They're part of an optional feature called POSIX Threads, or "pthreads," and were introduced in the mid-1990s. More on POSIX in Section 1.5 and threads in Chapter 5.

An example of a kernel-originated signal is a segmentation-violation signal, sent when a process attempts to access memory outside of its address space. An example of a signal sent by a process to itself is an abort signal, sent by the `abort` function to terminate a process with a core dump. An example of a signal sent from one process to another is a termination signal, sent when one of several related processes decides to terminate the whole family. Finally, an example of a user-originated signal is an interrupt signal, sent to all processes created by the user when he or she types Ctrl-c.

There are about 28 types of signals (some versions of UNIX have a few more or a few less). For all but 2, the kill and stop signals, a process can control what happens when it receives the signal. It can accept the default action, which usually results in termination of the process; it can ignore the signal; or it can catch the signal and execute a function, called a signal handler, when the signal arrives. The signal type (`SIGALRM`, say) is passed as an argument to the handler. There isn't any direct way for the handler to determine who sent the signal, however.[6] When the signal handler returns, the process resumes executing at the point of interruption. Two signals are left unused by the kernel. These may be used by an application for its own purposes.

1.1.4 Process-IDs, Process Groups, and Sessions

Every process has a *process-ID,* which is a positive integer. At any instant, these are guaranteed to be unique. Every process but one has a parent.

A process's system data also records its *parent-process-ID,* the process-ID of its parent. If a process is orphaned because its parent terminated before it did, its parent-process-ID is changed to 1 (or some other fixed number). This is the process-ID of the initialization process (init), created at boot time, which is the ancestor of all other processes. In other words, the initialization process adopts all orphans.

Sometimes programmers choose to implement a subsystem as a group of related processes instead of as a single process. The UNIX kernel allows these related processes to be organized into a *process group*. Process groups are further organized into *sessions*.

6. That's true of the basic 28 signals. The Realtime Signals option adds some signals for which this information, and more, is available. More in Chapter 9.

One of the session members is the *session leader,* and one member of each process group is the *process-group leader*. For each process, UNIX keeps track of the process IDs of its process-group leader and session leader so that a process can locate itself in the hierarchy.

The kernel provides a system call to send a signal to each member of a process group. Typically, this would be used to terminate the entire group, but any signal can be broadcast in this way.

UNIX shells generally create one session per login. They usually form a separate process group for the processes in a pipeline; in this context a process group is also called a *job*. But that's just the way shells like to do things; the kernel allows more flexibility so that any number of sessions can be formed within a single login, and each session can have its process groups formed any way it likes, as long as the hierarchy is maintained.

It works this way: Any process can resign from its session and then become a leader of its own session (of one process group containing only one process). It can then create child processes to round out the new process group. Additional process groups within the session can be formed and processes assigned to the various groups. (The assignments can be changed, too.) Hence, a single user could be running, say, a session of 10 processes formed into, say, three process groups.

A session, and thus the processes in it, can have a *controlling terminal,* which is the first terminal device opened by the session leader; the session leader then becomes the *controlling process*. Normally, the controlling terminal for a user's processes is the terminal from which the user logged in. When a new session is formed, the processes in the new session no longer have a controlling terminal, until one is opened. Not having a controlling terminal is typical of so-called *daemons*[7] (Web servers, the `cron` facility, etc.) which, once started, run divorced from whatever terminal was used to start them.

It's possible to organize things so that only one process group in a session—the foreground job—has access to the terminal and to move process groups back and forth between foreground and background. This is called *job control*. A background process group that attempts to access the terminal is suspended until

7. "Demon" and "daemon" are two spellings of the same word, but the former connotes an evil spirit, whereas the latter connotes a supernatural being somewhere between god and man. Misbehaving daemons may properly be referred to as demons.

moved to the foreground. This prevents it from stealing input from the foreground job or scribbling on the foreground process group's output.

If job control isn't being used (generally, only shells use it), all the processes in a session are effectively foreground processes and are equally free to access the terminal, even if havoc ensues.

The terminal device driver sends interrupt, quit, and hangup signals coming from a terminal to every process in the foreground process group for which that terminal is the controlling terminal. For example, unless precautions are taken, hanging up a terminal (e.g., closing a `telnet` connection) will terminate all of the user's processes in that group. To prevent this, a process can arrange to ignore hang-ups.

Additionally, when a session leader (controlling process) terminates for *any* reason, all processes in the foreground process group are sent a hangup signal. So simply logging out usually terminates all of a user's processes unless specific steps have been taken to either make them background processes or otherwise make them immune to the hangup signal.

In summary, there are four process-IDs associated with each process:

- process-ID: Positive integer that uniquely identifies this process.
- parent-process-ID: Process-ID of this process's parent.
- process-group-ID: Process-ID of the process-group leader. If equal to the process-ID, this process is the group leader.
- session-leader-ID: Process-ID of the session leader.

1.1.5 Permissions

A *user-ID* is a positive integer that is associated with a user's *login name* in the *password file* (/etc/passwd). When a user logs in, the `login` command makes this ID the user-ID of the first process created, the login shell. Processes descended from the shell inherit this user-ID.

Users are also organized into *groups* (not to be confused with process groups), which have IDs too, called *group-IDs.* A user's login group-ID is made the group-ID of his or her login shell.

Groups are defined in the *group file* (/etc/group). While logged in, a user can change to another group of which he or she is a member. This changes the group-ID of the process that handles the request (normally the shell, via the `newgrp` com-

mand), which then is inherited by all descendent processes. As a user may be a member of more than one group, a process also has a list of *supplementary group-IDs*. For most purposes these are also checked when a process's group permissions are at issue, so a user doesn't have to be constantly switching groups manually.

These two user and group login IDs are called the *real user-ID* and the *real group-ID* because they are representative of the real user, the person who is logged in. Two other IDs are also associated with each process: the *effective user-ID* and the *effective group-ID.* These IDs are normally the same as the corresponding real IDs, but they can be different, as we shall see shortly.

The effective ID is always used to determine permissions. The real ID is used for accounting and user-to-user communication. One indicates the user's permissions; the other indicates the user's identity.

Each file (regular, directory, socket, etc.) has, in its i-node, an *owner user-ID* (*owner* for short) and an *owner group-ID* (*group* for short). The i-node also contains three sets of three permission bits (nine bits in all). Each set has one bit for *read permission,* one bit for *write permission,* and one bit for *execute permission.* A bit is 1 if the permission is granted and 0 if not. There is a set for the owner, for the group, and for others (not in either of the first two categories). Table 1.1 shows the bit assignments (bit 0 is the rightmost bit).

Table 1.1 Permission Bits

Bit	Meaning
8	owner read
7	owner write
6	owner execute
5	group read
4	group write
3	group execute
2	others read
1	others write
0	others execute

Permission bits are frequently specified using an octal (base 8) number. For example, octal 775 would mean read, write, and execute permission for the owner and the group, and only read and execute permission for others. The `ls` command would show this combination of permissions as `rwxrwxr-x`; in binary it would be 111111101, which translates directly to octal 775. (And in decimal it would be 509, which is useless, because the numbers don't line up with the permission bits.)

The permission system determines whether a given process can perform a desired action (read, write, or execute) on a given file. For regular files, the meaning of the actions is obvious. For directories, the meaning of read is obvious. "Write" permission on a directory means the ability to issue a system call that would modify the directory (add or remove a link). "Execute" permission means the ability to use the directory in a path (sometimes called "search" permission). For special files, read and write permissions mean the ability to execute the read and write system calls. What, if anything, that implies is up to the designer of the device driver. Execute permission on a special file is meaningless.

The permission system determines whether permission will be granted using this algorithm:

1. If the effective user-ID is zero, permission is instantly granted (the effective user is the *superuser*).
2. If the process's effective user-ID matches the file's user-ID, then the owner set of bits is used to see if the action will be allowed.
3. If the process's effective group-ID or one of the supplementary group-IDs matches the file's group-ID, then the group set of bits is used.
4. If neither the user-IDs nor group-IDs match, then the process is an "other" and the third set of bits is used.

The steps go in order, so if in step 3 access is denied because, say, write permission is denied for the group, then the process cannot write, even though the "other" permission (step 4) might allow writing. It might be unusual for the group to be more restricted than others, but that's the way it works. (Imagine an invitation to a surprise party for a team of employees. Everyone *except* the team should be able to read it.)

There are other actions, which might be called "change i-node," that only the owner or the superuser can do. These include changing the user-ID or group-ID of

a file, changing a file's permissions, and changing a file's access or modification times. As a special case, write permission on a file allows setting of its access and modification times to the current time.

Occasionally, we want a user to temporarily take on the privileges of another user. For example, when we execute the `passwd` command to change our password, we would like the effective user-ID to be that of the superuser, because only root can write into the password file. This is done by making root (the superuser's login name) the owner of the `passwd` command (i.e., the regular file containing the `passwd` program) and then turning on another permission bit in the `passwd` command's i-node, called the *set-user-ID* bit. Executing a program with this bit on changes the effective user-ID of the process to the owner of the file containing the program. Since it's the effective user-ID, rather than the real user-ID, that determines permissions, this allows a user to temporarily take on the permissions of someone else. The *set-group-ID* bit is used in a similar way.

Since both user-IDs (real and effective) are inherited from parent process to child process, it is possible to use the set-user-ID feature to run with an effective user-ID for a very long time. That's what the `su` command does.

There is a potential loophole. Suppose you do the following: Copy the `sh` command to your own directory (you will be the owner of the copy). Then use `chmod` to turn on the set-user-ID bit and `chown` to change the file's owner to root. Now execute your copy of `sh` and take on the privileges of root! Fortunately, this loophole was closed a long time ago. If you're not the superuser, changing a file's owner automatically clears the set-user-ID and set-group-ID bits.

1.1.6 Other Process Attributes

A few other interesting attributes are recorded in a process's system data segment.

There is one open *file descriptor* (an integer from 0 through 1000 or so) for each file (regular, special, socket, or named pipe) that the process has opened, and two for each unnamed pipe (see Section 1.1.7) that the process has created. A child doesn't inherit open file descriptors from its parent, but rather duplicates of them. Nonetheless, they are indices into the same system-wide open file table, which among other things means that parent and child share the same file offset (the byte where the next read or write takes place).

A process's *priority* is used by the kernel scheduler. Any process can lower its priority via the system call `nice`; a superuser process can raise its priority (i.e., be not nice) via the same system call. Technically speaking, `nice` sets an attribute called the *nice value,* which is only one factor in computing the actual priority.

A process's *file size limit* can be (and usually is) less than the system-wide limit; this is to prevent confused or uncivilized users from writing runaway files. A superuser process can raise its limit.

There are many more process attributes that are discussed throughout this book as the relevant system calls are described.

1.1.7 Interprocess Communication

In the oldest (pre-1980) UNIX systems, processes could communicate with one another via shared file offsets, signals, *process tracing,* files, and *pipes*. Then *named pipes* (FIFOs) were added, followed by *semaphores, file locks, messages, shared memory,* and networking *sockets*. As we shall see throughout this book, none of these eleven mechanisms is entirely satisfactory. That's why there are eleven! There are even more because there are two versions each of semaphores, messages, and shared memory. There's one very old collection from AT&T's System V UNIX, called "System V IPC," and one much newer, from a real-time standards group that convened in the early 1990s, called "POSIX IPC." Almost every version of UNIX (including FreeBSD and Linux) has the oldest eleven mechanisms, and the main commercial versions of UNIX (e.g., Sun's Solaris or HP's HP/UX) have all fourteen.

Shared file offsets are rarely used for interprocess communication. In theory, one process could position the file offset to some fictitious location in a file, and a second process could then find out where it points. The location (a number between, say, 0 and 100) would be the communicated data. Since the processes must be related to share a file offset, they might as well just use pipes.

Signals are sometimes used when a process just needs to poke another. For example, a print spooler could signal the actual printing process whenever a print file is spooled. But signals don't pass enough information to be helpful in most applications. Also, a signal interrupts the receiving process, making the programming more complex than if the receiver could get the communication when it was

ready. Signals are mainly used just to terminate processes or to indicate very unusual events.

With *process tracing,* a parent can control the execution of its child. Since the parent can read and write the child's data, the two can communicate freely. Process tracing is used only by debuggers, since it is far too complicated and unsafe for general use.

Files are the most common form of interprocess communication. For example, one might write a file with a process running `vi` and then execute it with a process running `python`. However, files are inconvenient if the two processes are running concurrently, for two reasons: First, the reader might outrace the writer, see an end-of-file, and think that the communication is over. (This can be handled through some tricky programming.) Second, the longer the two processes communicate, the bigger the file gets. Sometimes processes communicate for days or weeks, passing billions of bytes of data. This would quickly exhaust the file system.

Using an empty file for a semaphore is also a traditional UNIX technique. This takes advantage of some peculiarities in the way UNIX creates files. More details are given in Section 2.4.3.

Finally, something I can actually recommend: *Pipes* solve the synchronization problems of files. A pipe is not a type of regular file; although it has an i-node, there are no links to it. Reading and writing a pipe is somewhat like reading and writing a file, but with some significant differences: If the reader gets ahead of the writer, the reader *blocks* (stops running for a while) until there's more data. If the writer gets too far ahead of the reader, it blocks until the reader has a chance to catch up, so the kernel doesn't have too much data queued. Finally, once a byte is read, it is gone forever, so long-running processes connected via pipes don't fill up the file system.

Pipes are well known to shell users, who can enter command lines like

```
ls | wc
```

to see how many files they have. The kernel facility, however, is far more general than what the shell provides, as we shall see in Chapter 6.

Pipes, however, have three major disadvantages: First, the processes communicating over a pipe must be related, typically parent and child or two siblings. This is too constraining for many applications, such as when one process is a database manager and the other is an application that needs to access the database. The sec-

ond disadvantage is that writes of more than a locally set maximum (4096 bytes, say) are not guaranteed to be atomic, prohibiting the use of pipes when there are multiple writers—their data might get intermingled. The third disadvantage is that pipes might be too slow. The data has to be copied from the writing user process to the kernel and back again to the reader. No actual disk I/O is performed, but the copying alone can take too long for some critical applications. It's because of these disadvantages that fancier schemes have evolved.

Named pipes, also called *FIFOs,* were added to solve the first disadvantage of pipes. (FIFO stands for "first-in-first-out.") A named pipe exists as a special file, and any process with permission can open it for reading or writing. Named pipes are easy to program with, too, as we shall demonstrate in Chapter 7.

What's wrong with named pipes? They don't eliminate the second and third disadvantage of pipes: Interleaving can occur for big writes, and they are sometimes too slow. For the most critical applications, the newer interprocess communication features (e.g., messages or shared memory) can be used, but not nearly as easily.

A *semaphore* (in the computer business) is a counter that prevents two or more processes from accessing the same resource at the same time. As I've mentioned, files can be used for semaphores too, but the overhead is far too great for many applications. UNIX has two completely different semaphore mechanisms, one part of System V IPC, and the other part of POSIX IPC. (We'll get to POSIX in Section 1.5.)

A *file lock*, in effect a special-purpose semaphore, prevents two or more processes from accessing the same segment of a file. It's usually only effective if the processes bother to check; that weak form is called *advisory* locking. The stronger form, *mandatory* locking, is effective whether a process checks or not, but it's nonstandard and available only in some UNIX systems.

A *message* is a small amount of data (500 bytes, say) that can be sent to a *message queue.* Messages can be of different types. Any process with appropriate permissions can receive messages from a queue. It has lots of choices: either the first message, or the first message of a given type, or the first message of a group of types. As with semaphores, there are System V IPC messaging system calls, and completely different POSIX IPC system calls.

Shared memory potentially provides the fastest interprocess communication of all. The same memory is mapped into the address spaces of two or more processes.

As soon as data is written to the shared memory, it is instantly available to the readers. A semaphore or a message is used to synchronize the reader and writer. Sure enough, there are two versions of shared memory, and attentive readers can guess what they are.

I've saved the best for last: networking interprocess communication, using a group of system calls called *sockets*. (Other system calls in this group have names like `bind`, `connect`, and `accept`.) Unlike all of the other mechanisms I've talked about, the process you're communicating with via a socket doesn't have to be on the same machine. It can be on another machine, or anywhere on a local network or the Internet. That other machine doesn't even have to be running UNIX. It can be running Windows or whatever. It could be a networked printer or radio, for that matter.

Choosing the IPC mechanism for a particular application isn't easy, so I'll spend a lot of time comparing them in Chapters 7 and 8.

1.2 Versions of UNIX

Ken Thompson and Dennis Ritchie began UNIX as a research project at AT&T Bell Laboratories in 1969, and shortly thereafter it started to be widely used within AT&T for internal systems (e.g., to automate telephone-repair call centers). AT&T wasn't then in the business of producing commercial computers or of selling commercial software, but by the early 1970s it did make UNIX available to universities for educational purposes, provided the source code was disclosed only to other universities who had their own license. By the late 1970s AT&T was licensing source code to commercial vendors, too.

Many, if not most, of the licensees made their own changes to UNIX, to port it to new hardware, to add device drivers, or just to tune it. Perhaps the two most significant changes were made at the University of California at Berkeley: networking system-calls ("sockets") and support for virtual memory. As a result, universities and research labs throughout the world used the Berkeley system (called BSD, for Berkeley Software Distribution), even though they still got their licenses from AT&T. Some commercial vendors, notably Sun Microsystems, started with BSD UNIX as well. Bill Joy, a founder of Sun, was a principal BSD developer.

Development within Bell Labs continued as well, and by the mid-1980s the two systems had diverged widely. (AT&T's was by that time called System V.) Both supported virtual memory, but their networking system calls were completely different. System V had interprocess-communication facilities (messages, shared memory, and semaphores) that BSD provided for in different ways, and BSD had changed almost all of the commands and added lots more, notably `vi`.

There were lots of other UNIX variants, too, including a few UNIX clones that used no AT&T-licensed source code. But mostly the UNIX world divided into the BSD side, which included essentially all of academia and some important workstation makers, and the System V side, which included AT&T itself and some commercial vendors. Nearly all computer science students learned on a BSD system. (And when they came to Bell Labs they brought their favorite commands, and all their UNIX knowledge, with them.) Ironically, neither AT&T nor the University of California really wanted to be in the commercial software business! (AT&T said it did, but it was wrong.)

The Institute of Electrical and Electronics Engineers (IEEE) started an effort to standardize both UNIX system calls and commands in the mid-1980s, picking up where an industry group called /usr/group had left off. IEEE issued its first standard for the system calls in 1988. The official name was IEEE Std 1003.1-1988, which I'll call POSIX1988 for short.[8] The first standard for commands (e.g., `vi` and `grep`) came out in 1992 (IEEE Std 1003.2-1992). POSIX1988 was adopted as an international standard by the International Organization for Standardization (ISO) with minor changes, and became POSIX1990.

Standards only standardize the syntax and semantics of application program interfaces (APIs), not the underlying implementation. So, regardless of its origins or internal architecture, any system with enough functionality can conform to POSIX1990 or its successors, even systems like Microsoft Windows or Digital (now HP) VMS. If the application program conforms to the standard for operating-system APIs and for the language it's written in (e.g., C++), and the target system conforms, then the application's source code can be ported, and it will compile and run with no changes. In practice, applications, compilers, and operating systems have bugs, and probably every serious application has to use some nonstandard APIs, so porting does take some work, but the standards reduce it to a manageable level, way more manageable than it was before the standards came along.

8. POSIX stands for Portable Operating System Interface and is pronounced *pahz-icks*.

POSIX1990 was a tremendous achievement, but it wasn't enough to unify the System V and BSD camps because it didn't include important APIs, such as BSD sockets and System V messages, semaphores, and shared memory. Also, even newer APIs were being introduced for new applications of UNIX, principally real-time (applications that interface to the real world, such as engine-management for a car) and threads. Fortunately, IEEE had standards groups working on real-time and threads, too. (Section 1.5 has more on where standards went after POSIX1990.)

In 1987 AT&T and Sun, which used a BSD-derived UNIX, thought they could unify the two main versions of UNIX by working together to produce a single system, but this scared the rest of the industry into forming the Open Software Foundation, which ultimately produced its own version of UNIX (still containing code licensed from AT&T, however). So, in the early 1990s we had BSD, AT&T/Sun UNIX, the vendors' pre-OSF UNIX versions, and an OSF version.

It looked like UNIX might implode, with Microsoft Windows taking over the world, and this prompted the industry to get behind the X/Open Company, which was much more aggressive than IEEE about getting the important APIs into a standard. The newest standard defines 1108 function interfaces!

By the mid-1990s everything was actually starting to settle down a bit, and then Linux came along. It started quietly in 1991 with a Usenet newsgroup post titled "Free minix-like[9] kernel sources for 386-AT" from a grad student named Linus Torvalds, which started like this:

> Do you pine for the nice days of minix-1.1, when men were men and wrote their own device drivers? Are you without a nice project and just dying to cut your teeth on a OS you can try to modify for your needs? Are you finding it frustrating when everything works on minix? No more all-nighters to get a nifty program working? Then this post might be just for you :-)
>
> As I mentioned a month(?) ago, I'm working on a free version of a minix-lookalike for AT-386 computers. It has finally reached the stage where it's even usable (though may not be depending on what you want), and I am willing to put out the sources for wider distribution. It is just version 0.02 (+1 (very small) patch already), but I've successfully run bash/gcc/gnu-make/gnu-sed/compress etc under it.[10]

By the end of the 1990s, Linux had developed into a serious OS, largely through the efforts of hundreds of developers who work on its free and open source code.

9. Minix is a small instructional OS developed by Andrew Tanenbaum of the Free University of Amsterdam, first released in 1987. It used no AT&T code.

10. The original message and the other 700 million Usenet messages since 1981 are archived at Google Groups (*www.google.com/grphp*).

There are commercial vendors of Linux binary distributions for Intel-based PCs (e.g., Red Hat, SuSE, Mandrake) and hardware manufacturers who ship Linux computers (e.g., Dell, IBM, and even Sun). All of them make the source available, as required by the Linux license, which is the complete opposite of the AT&T license!

Meanwhile, although they had been making extensive changes to the kernel for 15 years, the programmers at Berkeley still had a system with AT&T licensed code in it. They began removing the AT&T code but ran out of funding before they could finish. They released an incomplete, but unencumbered, system as 4.4BSD-Lite in the early 1990s. (Had they done it a bit earlier, Torvalds would have started with that, instead of with Minix.) Several groups immediately set to work to flesh out 4.4BSD-Lite, and those efforts are known today as FreeBSD (for Intel CPUs and a few others), NetBSD/OpenBSD (portable), WindRiver's BSD/OS (commercially supported), and Darwin (the UNIX inside Mac OS X). Alas, the Berkeley group is no longer around.

Today, there are three principal strains of UNIX systems:

- Commercial, closed systems historically based on AT&T System V or BSD (Solaris, HP/UX, AIX, etc.)
- BSD-based systems, of which FreeBSD is the best-known, with Darwin coming on strong
- Linux

Which of these are "UNIX"? AT&T's UNIX-development organization, which owned the UNIX trademark, was sold to Novell in 1993. Novell promptly turned the UNIX trademark over to X/Open. In 1996, OSF and X/Open merged to form the Open Group, which today develops a comprehensive standard called the Single UNIX Specification [SUS2002]. "UNIX" is now a brand that can legally be applied to any system that meets the Open Group's requirements. The major hardware vendors all have UNIX-branded systems—even IBM's mainframe MVS (now called OS/390) carries the UNIX brand. But the open-source systems Linux and FreeBSD do not, even though they do claim to conform to one or another of the POSIX standards. It's convenient to refer to all of these systems as "UNIX-like."

1.3 Using System Calls

This section explains how system calls are used with C and other languages and presents some guidelines for using them correctly.

1.3.1 C (and C++) Bindings

How does a C or C++ programmer actually issue a system call? Just like any other function call. For example, `read` might be called like this:

```
amt = read(fd, buf, numbyte);
```

The implementation of the function `read` varies with the UNIX implementation. Usually it's a small function that executes some special code that transfers control from the user process to the kernel and then returns the results. The real work is done inside the kernel, which is why it's a system call and not just a library routine that can do its work entirely in user space, such as `qsort` or `strlen`.

Remember, though, that since a system call involves two context switches (from user to kernel and back), it takes much longer than a simple function call within a process's own address space. So it's a good idea to avoid excessive system calls. This point will be emphasized in Section 2.12 when we look into buffered I/O.

Every system call is defined in a header file, and you have to ensure that you've included the correct headers (sometimes more than one is needed) before you make the call. For instance, `read` requires that we include the header unistd.h, like this:

```
#include <unistd.h>
```

A single header typically defines more than one call (unistd.h defines about 80), so a typical program needs only a handful of includes. Even with a bunch more for Standard C[11] functions (e.g., string.h), it's still manageable. There's no harm in including a header you don't actually need, so it's easiest to collect the most common headers into a master header and just include that. For this book, the master header is defs.h, which I present in Section 1.6. I won't show the include for it in the example code in this book, but you should assume it's included in every C file I write.

Unfortunately, there are header-file conflicts caused by ambiguities in the various standards or by misunderstandings on the part of implementors, so you'll sometimes find that you need to jigger the order of includes or play some tricks with the C preprocessor to make everything come out OK. You can see a bit of this in defs.h, but at least it hides the funny business from the code that includes it.

11. By Standard C, I mean the original 1989 standard, the 1995 update, or the latest, known as C99. When I mean C99 specifically, I'll say so.

No standard requires that a given facility be implemented as a system call, rather than a library function, and I won't make much of the distinction between the two when I introduce one or when I show one being used in an example program. Don't take my use of the term "system call" too literally, either; I just mean that the facility is usually implemented that way.

1.3.2 Other Language Bindings

While the main UNIX standards documents, POSIX and SUS, describe the interfaces in C, they recognize that C is only one of many languages, and so they've defined bindings for other standardized languages as well, such as Fortran and Ada. No problem with easy functions like `read`, as long as the language has some way of passing in integers and addresses of buffers and getting an integer back, which most do (Fortran was a challenge). It's tougher to handle structures (used by `stat`, for instance) and linked lists (used by `getaddrinfo`) . Still, language experts usually find a way.[12]

Lots of languages, like Java, Perl, and Python, have standard facilities that support some POSIX facilities. For instance, here's a Python program that uses the UNIX system called `fork` to create a child process. (I'll get to `fork` in Chapter 5; for now all you need to know is that it creates a child process and then returns in both the child *and* the parent, with different values, so both the `if` and `else` are true, in the parent and child, respectively.)

```
import os
pid = os.fork()
if pid == 0:
    print 'Parent says, "HELLO!"'
else:
    print 'Child says, "hello!"'
```

This is the output:

```
Parent says, "HELLO!"
Child says, "hello!"
```

There won't be any further examples using languages other than C and C++ in this book other than in Appendix C, which talks about Java and Jython (a version of Python that runs in a Java environment).

12. For example, Jtux, described in Appendix C, makes extensive use of the Java Native Interface (JNI).

1.3.3 Guidelines for Calling Library Functions

Here are some general guidelines for calling library functions (from C or C++), which apply to system calls or to any other functions:

- Include the necessary **headers** (as I discussed previously).
- Make sure you know how the function indicates an **error**, and check for it unless you have a good reason not to (Section 1.4). If you're not going to check, document your decision by casting the return value to `void`, like this:

  ```
  (void)close(fd);
  ```

  We violate this guideline consistently for `printf`, but with few other exceptions I follow it in our example programs.

- Don't use **casts** unless absolutely necessary, because they might hide a mistake. Here's what you shouldn't do:

  ```
  int n;
  struct xyz *p;
  ...
  free((void *)p); /* gratuitous cast */
  ```

  The problem is that if you mistakenly write `n` instead of `p`, the cast will suppress a compiler warning. You don't need the cast when the function's prototype specifies `void *`. If you're not sure whether the type you want to use is OK, leave the cast off so the compiler can tell you. This would apply to the case when the function calls, say, for an `int` and you want to give it a `pid_t` (the type for a process ID). Better to get the warning if there will be one than to shut the compiler up before it has a chance to speak. If you do get a warning, and you've determined that a cast is the fix, you can put the cast in then.
- If you've got more than one **thread**, know whether the function you want to call is thread safe—that is, whether it handles global variables, mutexes (semaphores), signals, etc., correctly for a multithreaded program. Many functions don't.
- Try to write to the **standard interfaces** rather than to those on your particular system. This has a lot of advantages: Your code will be more portable, the standardized functionality is probably more thoroughly tested, it will be easier for another programmer to understand what you're doing, and your code is more likely to be compatible with the next version of the operating system. In practice, what you can do is keep a browser open to the Open

Group Web site (Section 1.9 and [SUS2002]), where there's a terrific alphabetized reference to all standard functions (including the Standard C library) and all header files. This is a much better approach than, say, using the `man` pages on your local UNIX system. Sure, sometimes you do need to look up particulars for your system, and occasionally you need to use something nonstandard. But make that the exception rather than the rule, and document that it's nonstandard with a comment. (Lots more about what I mean by "standard" in Section 1.5.)

1.3.4 Function Synopses

I'll formally introduce each system call or function that I discuss in detail with a brief synopsis of it that documents the necessary headers, the arguments, and how errors are reported. Here's one for `atexit`, a Standard C function, that we're going to need in Section 1.4.2:

atexit—register function to be called when process exits

```
#include <stdlib.h>

int atexit(
    void (*fcn)(void)      /* function to be called */
);
/* Returns 0 on success, non-zero on error (errno not defined) */
```

(If you're not already familiar with `errno`, it's explained in the next section.)

The way `atexit` works is that you declare a function like this:

```
static void fcn(void)
{
    /* work to be done at exit goes here */
}
```

and then you register it like this:

```
if (atexit(fcn) != 0) {
    /* handle error */
}
```

Now when the process exits, your function is automatically called. You can register more than 1 function (up to 32), and they're all called in the reverse order of their registration. Note that the test in the `if` statement was for non-zero, since that's exactly what the wording in the synopsis said. It wasn't a test for 1, or greater than zero, or –1, or `false`, or `NULL`, or anything else. That's the only safe

way to do it. Also, it makes it easier later if you're looking for a bug and are trying to compare the code to the documentation. You don't have to solve a little puzzle in your head to compare the two.

1.4 Error Handling

Testing error returns from system calls is tricky, and handling an error once you discover it is even tricker. This section explains the problem and gives some practical solutions.

1.4.1 Checking for Errors

Most system calls return a value. In the `read` example (Section 1.3.1), the number of bytes read is returned. To indicate an error, a system call usually returns a value that can't be mistaken for valid data, typically –1. Therefore, my example should have been coded something like this:

```
if ((amt = read(fd, buf, numbyte)) == -1) {
    fprintf(stderr, "Read failed!\n");
    exit(EXIT_FAILURE);
}
```

Note that `exit` is a system call too, but it can't return an error because it doesn't return. The symbol `EXIT_FAILURE` is in Standard C.

Of the system calls covered in this book, about 60% return –1 on an error, about 20% return something else, such as `NULL`, zero, or a special symbol like `SIG_ERR`, and about 20% don't report errors at all. So, you just can't assume that they all behave the same way—you have to read the documentation for each call. I'll provide the information when I introduce each system call.

There are lots of reasons why a system call that returns an error indication might have failed. For 80% of them, the integer symbol `errno` contains a code that indicates the reason. To get at `errno` you include the header errno.h. You can use `errno` like an integer, although it's not necessarily an integer variable. If you're using threads, `errno` is likely to involve a function call, because the different threads can't reliably all use the same global variable. So don't declare `errno` yourself (which used to be exactly what you were supposed to do), but use the definition in the header, like this (other headers not shown):

```
#include <errno.h>

if ((amt = read(fd, buf, numbyte)) == -1) {
    fprintf(stderr, "Read failed! errno = %d\n", errno);
    exit(EXIT_FAILURE);
}
```

If, say, the file descriptor is bad, the output would be:

```
Read failed! errno = 9
```

Almost always, you can use the value of `errno` only if you've first checked for an error; you can't just check `errno` to see if an error occurred, because its value is set only when a function that is specified to set it returns an error. So, this code would be wrong:

```
amt = read(fd, buf, numbyte);
if (errno != 0) { /* wrong! */
    fprintf(stderr, "Read failed! errno = %d\n", errno);
    exit(EXIT_FAILURE);
}
```

Setting `errno` to zero first works:

```
errno = 0;
amt = read(fd, buf, numbyte);
if (errno != 0) { /* bad! */
    fprintf(stderr, "Read failed! errno = %d\n", errno);
    exit(EXIT_FAILURE);
}
```

But it's still a bad idea because:

- If you modify the code later to insert another system call, or any function that eventually executes a system call, before the call to `read`, the value of `errno` may be set by *that* call.
- Not all system calls set the value of `errno`, and you should get into the habit of checking for an error that conforms exactly to the function's specification.

Thus, for almost all system calls, check `errno` only after you've established that an error occurred.

Now, having warned you about using `errno` alone to check for an error, this being UNIX, I have to say that there are a few exceptions (e.g., `sysconf` and `readdir`) that do rely on a changed `errno` value to indicate an error, but even they return a specific value that tells you to check `errno`. Therefore, the rule

about not checking errno before first checking the return value is a good one, and it applies to most of the exceptions, too.

The errno value 9 that was displayed a few paragraphs up doesn't mean much, and isn't standardized, so it's better to use the Standard C function perror, like this:

```
if ((amt = read(fd, buf, numbyte)) == -1) {
    perror("Read failed!");
    exit(EXIT_FAILURE);
}
```

Now the output is:

```
Read failed!: Bad file number
```

Another useful Standard C function, strerror, just provides the message as a string, without also displaying it like perror does.

But the message "Bad file number," while clear enough, isn't standardized either, so there's still a problem: The official documentation for system calls and other functions that use errno refer to the various errors with symbols like EBADF, not by the text messages. For example, here's an excerpt from the SUS entry for read:

> [EAGAIN]
>
> The O_NONBLOCK flag is set for the file descriptor and the process would be delayed.
>
> [EBADF]
>
> The fildes argument is not a valid file descriptor open for reading.
>
> [EBADMSG]
>
> The file is a STREAM file that is set to control-normal mode and the message waiting to be read includes a control part.

It's straightforward to match "Bad file number" to EBADF, even though those exact words don't appear in the text, but not for the more obscure errors. What you really want along with the text message is the actual symbol, and there's no Standard C or SUS function that gives it to you. So, we can write our own function that translates the number to the symbol. We built the list of symbols in the code that follows from the errno.h files on Linux, Solaris, and BSD, since many symbols are system specific. You'll probably have to adjust this code for your

own system. For brevity, not all the symbols are shown, but the complete code is on the AUP Web site (Section 1.8 and [AUP2003]).

```
static struct {
    int code;
    char *str;
} errcodes[] =
{
    { EPERM, "EPERM" },
    { ENOENT, "ENOENT" },
      …
    { EINPROGRESS, "EINPROGRESS" },
    { ESTALE, "ESTALE" },
#ifndef BSD
    { ECHRNG, "ECHRNG" },
    { EL2NSYNC, "EL2NSYNC" },
      …
    { ESTRPIPE, "ESTRPIPE" },
    { EDQUOT, "EDQUOT" },
#ifndef SOLARIS
    { EDOTDOT, "EDOTDOT" },
    { EUCLEAN, "EUCLEAN" },
      …
    { ENOMEDIUM, "ENOMEDIUM" },
    { EMEDIUMTYPE, "EMEDIUMTYPE" },
#endif
#endif
    { 0, NULL}
};

const char *errsymbol(int errno_arg)
{
    int i;

    for (i = 0; errcodes[i].str != NULL; i++)
        if (errcodes[i].code == errno_arg)
            return errcodes[i].str;
    return "[UnknownSymbol]";
}
```

Here's the error checking for `read` with the new function:

```
if ((amt = read(fd, buf, numbyte)) == -1) {
    fprintf(stderr, "Read failed!: %s (errno = %d; %s)\n",
      strerror(errno), errno, errsymbol(errno));
    exit(EXIT_FAILURE);
}
```

Now the output is complete:

```
Read failed!: Bad file descriptor (errno = 9; EBADF)
```

It's convenient to write one more utility function to format the error information, so we can use it in some code we're going to write in the next section:

```
char *syserrmsg(char *buf, size_t buf_max, const char *msg, int errno_arg)
{
    char *errmsg;

    if (msg == NULL)
        msg = "???";
    if (errno_arg == 0)
        snprintf(buf, buf_max, "%s", msg);
    else {
        errmsg = strerror(errno_arg);
        snprintf(buf, buf_max, "%s\n\t\t*** %s (%d: \"%s\") ***", msg,
          errsymbol(errno_arg), errno_arg,
          errmsg != NULL ? errmsg : "no message string");
    }
    return buf;
}
```

We would use `syserrmsg` like this:

```
if ((amt = read(fd, buf, numbyte)) == -1) {
    fprintf(stderr, "%s\n", syserrmsg(buf, sizeof(buf),
      "Call to read function failed", errno));
    exit(EXIT_FAILURE);
}
```

with output like this:

```
Call to read function failed
   *** EBADF (9: "Bad file descriptor") ***
```

What about the other 20% of the calls that report an error but don't set `errno`? Well, around 20 of them report the error another way, typically by simply returning the error code (that is, a non-zero return indicates that an error occurred and also what the code is), and the rest don't provide a specific reason at all. I'll provide the details for every function (all 300 or so) in this book. Here's one that returns the error code directly (what this code is supposed to do isn't important right now):

```
struct addrinfo *infop;

if ((r = getaddrinfo("localhost", "80", NULL, &infop)) != 0) {
```

```
    fprintf(stderr, "Got error code %d from getaddrinfo\n", r);
    exit(EXIT_FAILURE);
}
```

The function `getaddrinfo` is one of those that doesn't set `errno`, and you can't pass the error code it returns into `strerror`, because that function works only with `errno` values. The various error codes returned by the non-`errno` functions are defined in [SUS2002] or in your system's manual, and you certainly could write a version of `errsymbol` (shown earlier) for those functions. But what makes this difficult is that the symbols for one function (e.g., `EAI_BADFLAGS` for `getaddrinfo`) aren't specified to have values distinct from the symbols for another function. This means that you can't write a function that takes an error code alone and looks it up, like `errsymbol` did. You have to pass the name of the function in as well. (If you do, you could take advantage of `gai_strerror`, which is a specially tailored version of `strerror` just for `getaddrinfo`.)

There are about two dozen functions in this book for which the standard [SUS2002] doesn't define any `errno` values or even say that `errno` is set, but for which your implementation may set `errno`. The phrase "errno not defined" appears in the function synopses for these.

Starting to get a headache? UNIX error handling is a real mess. This is unfortunate because it's hard to construct test cases to make system calls misbehave so the error handling you've coded can be checked, and the inconsistencies make it hard to get it right every single time. But the chances of it getting any better soon are zero (it's frozen by the standards), so you'll have to live with it. Just be careful!

1.4.2 Error-Checking Convenience Macros for C

It's tedious to put every system call in an `if` statement and follow it with code to display the error message and exit or return. When there's cleanup to do, things get even worse, as in this example:

```
if ((p = malloc(sizeof(buf))) == NULL) {
    fprintf(stderr, "%s\n", syserrmsg(buf, sizeof(buf),
      "malloc failed", errno));
    return false;
}
if ((fdin = open(filein, O_RDONLY)) == -1) {
    fprintf(stderr, "%s\n", syserrmsg(buf, sizeof(buf),
      "open (input) failed", errno));
```

```
        free(p);
        return false;
    }
    if ((fdout = open(fileout, O_WRONLY)) == -1) {
        fprintf(stderr, "%s\n", syserrmsg(buf, sizeof(buf),
          "open (output) failed", errno));
        (void)close(fdin);
        free(p);
        return false;
    }
```

The cleanup code gets longer and longer as we go. It's hard to write, hard to read, and hard to maintain. Many programmers will just use a `goto` so the cleanup code can be written just once. Ordinarily, `goto`s are to be avoided, but here they seem worthwhile. Note that we have to carefully initialize the variables that are involved in cleanup and then test the file-descriptor values so that the cleanup code can execute correctly no matter what the incomplete state.

```
    char *p = NULL;
    int fdin = -1, fdout = -1;

    if ((p = malloc(sizeof(buf))) == NULL) {
        fprintf(stderr, "%s\n", syserrmsg(buf, sizeof(buf),
          "malloc failed", errno));
        goto cleanup;
    }
    if ((fdin = open(filein, O_RDONLY)) == -1) {
        fprintf(stderr, "%s\n", syserrmsg(buf, sizeof(buf),
          "open (input) failed", errno));
        goto cleanup;
    }
    if ((fdout = open(fileout, O_WRONLY)) == -1) {
        fprintf(stderr, "%s\n", syserrmsg(buf, sizeof(buf),
          "open (output) failed", errno));
        goto cleanup;
    }
    return true;

cleanup:
    free(p);
    if (fdin != -1)
        (void)close(fdin);
    if (fdout != -1)
        (void)close(fdout);
    return false;
```

Still, coding all those `if`s, `fprintf`s, and `goto`s is a pain. The system calls themselves are almost buried!

We can streamline the jobs of checking for the error, displaying the error information, and getting out of the function with some macros. I'll first show how they're used and then how they're implemented. (This stuff is just Standard C coding, not especially connected to system calls or even to UNIX, but I've included it because it'll be used in all the subsequent examples in this book.)

Here's the previous example rewritten to use these error-checking ("ec") macros. The context isn't shown, but this code is inside of a function named `fcn`:

```
    char *p = NULL;
    int fdin = -1, fdout = -1;

    ec_null( p = malloc(sizeof(buf)) )
    ec_neg1( fdin = open(filein, O_RDONLY) )
    ec_neg1( fdout = open(fileout, O_WRONLY) )

    return true;

EC_CLEANUP_BGN
    free(p);
    if (fdin != -1)
        (void)close(fdin);
    if (fdout != -1)
        (void)close(fdout);
    return false;
EC_CLEANUP_END
```

Here's the call to the function. Because it's in the `main` function, it makes sense to exit on an error.

```
    ec_false( fcn() )

    /* other stuff here */

    exit(EXIT_SUCCESS);

EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
EC_CLEANUP_END
```

Here's what's going on: The macros `ec_null`, `ec_neg1`, and `ec_false` check their argument expression against `NULL`, –1, and `false`, respectively, store away the error information, and go to a label that was placed by the `EC_CLEANUP_BGN` macro. Then the same cleanup code as before is executed. In `main`, the test of the return value of `fcn` also causes a jump to the same label in `main` and the program exits. A function installed with `atexit` (introduced in Section 1.3.4) displays all the accumulated error information:

```
ERROR:  0: main [/aup/c1/errorhandling.c:41] fcn()
        1: fcn [/aup/c1/errorhandling.c:15] fdin = open(filein, 0x0000)
                *** ENOENT (2: "No such file or directory") ***
```

What we see is the `errno` symbol, value, and descriptive text on the last line. It's preceded by a reverse trace of the error returns. Each trace line shows the level, the name of the function, the file name, the line number, and the code that returned the error indication. This isn't the sort of information you want your end users to see, but during development it's terrific. Later, you can change the macros (we'll see how shortly) to put these details in a log file, and your users can see something more meaningful to them.

We accumulated the error information rather than printing it as we went along because that gives an application developer the most freedom to handle errors as he or she sees fit. It really doesn't do for a function in a library to just write error messages to `stderr`. That may not be the right place, and the wording may not be appropriate for the application's end users. In the end we did print it, true, but that decision can be easily changed if you use these macros in a real application.

So, what these macros give us is:

- Easy and readable error checking
- An automatic jump to cleanup code
- Complete error information along with a back trace

The downside is that the macros have a somewhat strange syntax (no semicolon at the end) and a buried jump in control flow, which some programmers think is a very bad idea. If you think the benefits outweigh the costs, use the macros (as I will in this book). If not, work out your own set (maybe with an explicit `goto` instead of a buried one), or skip them entirely.

Here's most of the header file (ec.h) that implements the error-checking macros (function declarations and a few other minor details are omitted):

```
extern const bool ec_in_cleanup;

typedef enum {EC_ERRNO, EC_EAI} EC_ERRTYPE;

#define EC_CLEANUP_BGN\
    ec_warn();\
    ec_cleanup_bgn:\
    {\
        bool ec_in_cleanup;\
        ec_in_cleanup = true;
```

```
#define EC_CLEANUP_END\
    }

#define ec_cmp(var, errrtn)\
    {\
        assert(!ec_in_cleanup);\
        if ((intptr_t)(var) == (intptr_t)(errrtn)) {\
            ec_push(__func__, __FILE__, __LINE__, #var, errno, EC_ERRNO);\
            goto ec_cleanup_bgn;\
        }\
    }

#define ec_rv(var)\
    {\
        int errrtn;\
        assert(!ec_in_cleanup);\
        if ((errrtn = (var)) != 0) {\
            ec_push(__func__, __FILE__, __LINE__, #var, errrtn, EC_ERRNO);\
            goto ec_cleanup_bgn;\
        }\
    }

#define ec_ai(var)\
    {\
        int errrtn;\
        assert(!ec_in_cleanup);\
        if ((errrtn = (var)) != 0) {\
            ec_push(__func__, __FILE__, __LINE__, #var, errrtn, EC_EAI);\
            goto ec_cleanup_bgn;\
        }\
    }

#define ec_neg1(x) ec_cmp(x, -1)
#define ec_null(x) ec_cmp(x, NULL)
#define ec_false(x) ec_cmp(x, false)
#define ec_eof(x) ec_cmp(x, EOF)
#define ec_nzero(x)\
    {\
        if ((x) != 0)\
            EC_FAIL\
    }

#define EC_FAIL ec_cmp(0, 0)

#define EC_CLEANUP goto ec_cleanup_bgn;
```

```
#define EC_FLUSH(str)\
    {\
        ec_print();\
        ec_reinit();\
    }
```

Before I explain the macros, I have to bring up a problem and talk about its solution. The problem is that if you call one of the error-checking macros (e.g., `ec_neg1`) inside the cleanup code and an error occurs, there will most likely be an infinite loop, since the macro will jump to the cleanup code! Here's what I'm worried about:

```
EC_CLEANUP_BGN
    free(p);
    if (fdin != -1)
        ec_neg1( close(fdin) )
    if (fdout != -1)
        ec_neg1( close(fdout) )
    return false;
EC_CLEANUP_END
```

It looks like the programmer is being very careful to check the error return from `close`, but it's a disaster in the making. What's really bad about this is that the loop would occur only when there was an error cleaning up after an error, a rare situation that's unlikely to be caught during testing. We want to guard against this—the error-checking macros should increase reliability, not reduce it!

Our solution is to set a local variable `ec_in_cleanup` to `true` in the cleanup code, which you can see in the definition for the macro `EC_CLEANUP_BGN`. The test against it is in the macro `ec_cmp`—if it's set, the `assert` will fire and we'll know right away that we goofed.

(The type `bool` and its values, `true` and `false`, are new in C99. If you don't have them, you can just stick the code

```
typedef int bool;
#define true 1
#define false 0
```

in one of your header files.)

To prevent the assertion from firing when `ec_cmp` is called outside of the cleanup code (i.e., a normal call), we have a global variable, also named `ec_in_cleanup`, that's permanently set to `false`. This is a rare case when it's OK (essential, really) to hide a global variable with a local one.

Why have the local variable at all? Why not just set the global to `true` at the start of the cleanup code, and back to `false` at the end? That won't work if you call a function from within the cleanup code that happens to use the `ec_cmp` macro legitimately. It will find the global set to true and think it's in its own cleanup code, which it isn't. So, each function (that is, each unique cleanup-code section) needs a private guard variable.

Now I'll explain the macros one-by-one:

- `EC_CLEANUP_BGN` includes the label for the cleanup code (`ec_cleanup_bgn`), preceded by a function call that just outputs a warning that control flowed into the label. This guards against the common mistake of forgetting to put a `return` statement before the label and flowing into the cleanup code even when there was no error. (I put this in after I wasted an hour looking for an error that wasn't there.) Then there's the local `ec_in_cleanup`, which I already explained.
- `EC_CLEANUP_END` just supplies the closing brace. We needed the braces to create the local context.
- `ec_cmp` does most of the work: Ensuring we're not in cleanup code, checking the error, calling `ec_push` (which I'll get to shortly) to push the location information (`__FILE__`, etc.) onto a stack, and jumping to the cleanup code. The type `intptr_t` is new in C99: It's an integer type guaranteed to be large enough to hold a pointer. If you don't have it yet, `typedef` it to be a `long` and you'll probably be OK. Just to be extra safe, stick some code in your program somewhere to test that `sizeof(void *)` is equal to `sizeof(long)`. (If you're not familiar with the notation `#var`, read up on your C—it makes whatever `var` expands to into a string.)
- `ec_rv` is similar to `ec_cmp`, but it's for functions that return a non-zero error code to indicate an error and which don't use `errno` itself. However, the codes it returns are `errno` values, so they can be passed directly to `ec_push`.
- `ec_ai` is similar to `ec_rv`, but the error codes it deals with aren't `errno` values. The last argument to `ec_push` becomes `EC_EAI` to indicate this. (Only a couple of functions, both in Chapter 8, use this approach.)
- The macros `ec_neg1`, `ec_null`, `ec_false`, and `ec_eof` call `ec_cmp` with the appropriate arguments, and `ec_nzero` does its own checking. They cover the most common cases, and we can just use `ec_cmp` directly for the others.

- `EC_FAIL` is used when an error condition arises from a test that doesn't use the macros in the previous paragraph.
- `EC_CLEANUP` is used when we just want to jump to the cleanup code.
- `EC_FLUSH` is used when we just want to display the error information, without waiting to exit. It's handy in interactive programs that need to keep going. (The argument isn't used.)

The various service functions called from the macros won't be shown here, since they don't illustrate much about UNIX system calls (they just use Standard C), but you can go to the AUP Web site [AUP2003] to see them along with an explanation of how they work. Here's a summary:

- `ec_push` pushes the error and context information passed to it (by the `ec_cmp` macro, say) onto a stack.
- There's a function registered with `atexit` that prints the information on the stack when the program exits:

```
static void ec_atexit_fcn(void)
{
    ec_print();
}
```

- `ec_print` walks down the stack to print the trace and error information.
- `ec_reinit` erases what's on the stack, so error-checking can start with a fresh trace.
- `ec_warn` is called from the `EC_CLEANUP_BGN` code if we accidentally fall into it.

All the functions are thread-safe so they can be used from within multithreaded programs. More on what this means in Section 5.17.

1.4.3 Using C++ Exceptions

Before you spend too much time and energy deciding whether you like the "ec" macros in the previous section and coming up with some improvements, you might ask yourself whether you'll even be programming in C. These days it's much more likely that you'll use C++. Just about everything in this book works fine in a C++ program, after all. C is still often preferred for embedded systems, operating systems (e.g., Linux), compilers, and other relatively low-level software, but those kinds of systems are likely to have their own, highly specialized, error-handling mechanisms.

C++ provides an opportunity to handle errors with exceptions, built into the C++ language, rather than with the combination of `goto`s and `return` statements that

we used in C. Exceptions have their own pitfalls, but if used carefully they're easier to use and more reliable than the "ec" macros, which don't protect you from, say, using `ec_null` when you meant to use `ec_neg1`.

As the library that contains the system-call wrapper code is usually set up just for C, it won't throw exceptions unless someone's made a special version for C++. So, to use exceptions you need another layer of wrapping, something like this for the `close` system call:

```
class syscall_ex {
public:
    int se_errno;

    syscall_ex(int n)
        : se_errno(n)
        { }
    void print(void)
        {
            fprintf(stderr, "ERROR: %s\n", strerror(se_errno));
        }

};

class syscall {
public:
    static int close(int fd)
        {
            int r;
            if ((r = ::close(fd)) == -1)
                throw(syscall_ex(errno));
            return r;
        }
};
```

Then you just call `syscall::close` instead of plain `close`, and it throws an exception on an error. You probably don't want to type in the code for the other 1100 or so UNIX functions, but perhaps just the ones your application uses.

If you want the exception information to include location information such as file and line, you need to define *another* wrapper, this time a macro, to capture the preprocessor data (e.g., via `__LINE__`),[13] so here's an even fancier couple of classes:

13. We need the macro because if you just put `__LINE__` and the others as direct arguments to the `syscall_ex` constructor, you get the location in the definition of `class syscall`, which is the wrong place.

```
class syscall_ex {
public:
    int se_errno;
    const char *se_file;
    int se_line;
    const char *se_func;

    syscall_ex(int n, const char *file, int line, const char *func)
        : se_errno(n), se_file(file), se_line(line), se_func(func)
        { }
    void print(void)
        {
            fprintf(stderr, "ERROR: %s [%s:%d %s()]\n",
              strerror(se_errno), se_file, se_line, se_func);
        }

};

class syscall {
public:
    static int close(int fd, const char *file, int line, const char *func)
        {
            int r;
            if ((r = ::close(fd)) == -1)
                throw(syscall_ex(errno, file, line, func));
            return r;
        }
};

#define Close(fd) (syscall::close(fd, __FILE__, __LINE__, __func__))
```

This time you call `Close` instead of `close`.

You can goose this up with a call-by-call trace, as we did for the "ec" macros if you like, and probably go even further than that.

There's an example C++ wrapper, Ux, for all the system calls in this book that's described in Appendix B.

1.5 UNIX Standards

Practically speaking, what you'll mostly find in the real world is that the commercial UNIX vendors follow the Open Group branding (Section 1.2), and the open-source distributors claim only conformance to POSIX1990, although, with few exceptions, they don't bother to actually run the certification tests. (The tests have now been made freely available, so future certification is now more likely.)

1.5.1 Evolution of the API Standards

Enumerating all the various POSIX and Open Group standards, guides, and specifications would be enormously complicated and confusing. For most purposes, the relevant developments can be simplified by considering them as a progression of standards relevant to the UNIX API, of which only the eight shown in Table 1.2 are important for the purposes of this book.

Table 1.2 POSIX and Open Group API Standards

Name	Standard	Comments
POSIX1988	IEEE Std 1003.1-1988 (198808L*)	First standard
POSIX1990	IEEE Std 1003.1-1990/ ISO 9945-1:1990 (199009L)	Minor update of POSIX1988
POSIX1993	IEEE Std 1003.1b-1993 (199309L)	POSIX1990 + real-time
POSIX1996	IEEE Std 1003.1-1996/ISO 9945-1:1996 (199506L)	POSIX1993 + threads + fixes to real-time
XPG3	X/Open Portability Guide	First widely distributed X/Open guide
SUS1	Single UNIX Specification, Version 1	POSIX1990 + all commonly used APIs from BSD, AT&T System V, and OSF; also known as Spec 1170; certified systems branded UNIX 95†
SUS2	Single UNIX Specification, Version 2	SUS1 updated to POSIX1996 + 64-bit, large file, enhanced multibyte, and Y2K; UNIX 98 brand
SUS3	Single UNIX Specification, Version 3 (200112L)	Update to SUS2; API part identical to IEEE Std 1003.1-2001 (POSIX fully merged with Open Group); UNIX 03 brand

* These huge numbers are the year and month of approval by IEEE. They're used in feature testing as explained in the next section.
† Spec 1170 gets its name from the total number of APIs, headers, and commands.

Actually, there is also a POSIX2001 (IEEE Std POSIX.1-2001), but as it's contained in SUS3, we don't need a separate row for it.

Even with this over-simplified list, we can already see that the loose statement that an OS "conforms to POSIX" is meaningless and probably indicates that whoever is making it either doesn't really know POSIX (or SUS) or is assuming that the listener doesn't. What POSIX? And, since newer features like real time and threads are optional, what options? Generally, here's how to interpret what you're hearing:

- If all you hear is POSIX, it probably means POSIX1990, which was the first and last OS standard that most people bothered to become aware of.
- Unless you know that the OS has passed certification tests established by standards organizations, all you can be sure of is that the OS developers took the POSIX standard into account somehow.
- You have to investigate the status of optional features separately (more on that shortly).

1.5.2 Telling the System What You Want

In this and the next section I'll describe what you're *supposed* to do according to the standards to ensure that your application conforms and to check what version of the standard the OS conforms to. Then you can decide how much of this you want to do and how much is just mumbo-jumbo that you can skip.

First of all, your application can state what version of the standard it expects by defining a preprocessor symbol before including any POSIX headers, such as unistd.h. This limits the standard headers (e.g., stdio.h, unistd.h, errno.h) to only those symbols that are defined in the standard. To limit yourself to classic POSIX1990, you do this:

```
#define _POSIX_SOURCE
#include <sys/types.h>
#include <unistd.h>
```

For a newer POSIX standard you have to be more specific, with another symbol, like this:

```
#define _POSIX_SOURCE
#define _POSIX_C_SOURCE 199506L
#include <sys/types.h>
#include <unistd.h>
```

Note that unlike `_POSIX_SOURCE`, `_POSIX_C_SOURCE` has a value which is normally a long integer formed from the year and month when the standard was approved (June 1995 in the example). The other possible long integer is `200112L`. But for POSIX1990, you set it to `1`, not to `199009L`.[14]

If you're interested in going beyond POSIX into SUS, which you will be if you're using a commercial UNIX system, there's a separate symbol for that, `_XOPEN_SOURCE`, and two special numbers for it: 500 for SUS2, and 600 for SUS3. So, if you want to use what's in a SUS2, UNIX 98-branded system, you do this:

```
#define _POSIX_SOURCE
#define _POSIX_C_SOURCE 199506L
#define _XOPEN_SOURCE 500
#include <sys/types.h>
#include <unistd.h>
```

For SUS1, you define `_XOPEN_SOURCE` without a value, and also define `_XOPEN_SOURCE_EXTENDED` as 1:

```
#define _POSIX_SOURCE
#define _POSIX_C_SOURCE 2
#define _XOPEN_SOURCE
#define _XOPEN_SOURCE_EXTENDED 1
#include <unistd.h>
```

Technically, you don't need to define the first two POSIX symbols if you really have a SUS2 system, since they will be defined automatically. But, the source you're writing may be used on an older system, and you need to tell it that you want POSIX in terms it will understand. `_POSIX_C_SOURCE` should use 2 when `_XOPEN_SOURCE` is defined with no value, `199506L` when it's 500, and `200112L` when it's 600.

Are you lost? Too bad, because it gets much worse. To help a little, here's a summary of what to do:

- Decide what level of standardization your program is written to expect.
- If it's only POSIX1990, define only `_POSIX_SOURCE`.
- If it's POSIX1993 or POSIX1996, define both `_POSIX_SOURCE` and `_POSIX_C_SOURCE`, and use the appropriate number from Table 1.2 in Section 1.5.1.

14. Or you can set it to 2 if you also want the C bindings from 1003.2-1992.

- If it's SUS1, SUS2, or SUS3, define _XOPEN_SOURCE (to nothing, to 500, or to 600) and also use the two POSIX symbols with the appropriate numbers.
- For SUS1, also define _XOPEN_SOURCE_EXTENDED as 1.
- Forget about XPG3 (and XPG4, which I haven't even mentioned)—you have enough to worry about already.

It's easy to write a header file that sets the various request symbols based on a single definition that you set before including it. Here's the header suvreq.h;[15] I'll show how it's used in the next section:

```
/*
    Header to request specific standard support. Before including it, one
    of the following symbols must be defined (1003.1-1988 isn't supported):

        SUV_POSIX1990    for 1003.1-1990
        SUV_POSIX1993    for 1003.1b-1993 - real-time
        SUV_POSIX1996    for 1003.1-1996
        SUV_SUS1         for Single UNIX Specification, v. 1 (UNIX 95)
        SUV_SUS2         for Single UNIX Specification, v. 2 (UNIX 98)
        SUV_SUS3         for Single UNIX Specification, v. 3 (UNIX 03)
*/
#if defined(SUV_POSIX1990)
#define _POSIX_SOURCE
#define _POSIX_C_SOURCE 1

#elif defined(SUV_POSIX1993)
#define _POSIX_SOURCE
#define _POSIX_C_SOURCE 199309L

#elif defined(SUV_POSIX1996)
#define _POSIX_SOURCE
#define _POSIX_C_SOURCE 199506L

#elif defined(SUV_SUS1)
#define _POSIX_SOURCE
#define _POSIX_C_SOURCE 2
#define _XOPEN_SOURCE
#define _XOPEN_SOURCE_EXTENDED 1

#elif defined(SUV_SUS2)
#define _POSIX_SOURCE
#define _POSIX_C_SOURCE 199506L
#define _XOPEN_SOURCE 500
#define _XOPEN_SOURCE_EXTENDED 1
```

15. It's pronounced "S-U-V wreck." SUV is for Standard UNIX Version. What did you think it was?

```
#elif defined(SUV_SUS3)
#define _POSIX_SOURCE
#define _POSIX_C_SOURCE 200112L
#define _XOPEN_SOURCE 600
#define _XOPEN_SOURCE_EXTENDED 1
#endif
```

Limiting your program to only what's in a standard is great if you're trying to write a highly portable application, because it helps ensure that you don't accidentally use anything nonstandard. But, for other applications, that's too limiting. For example, FreeBSD claims only to conform to POSIX1990, yet it does implement System V messages, which are at the SUS1 level. If you need messages, or any other post-POSIX1990 stuff that FreeBSD offers, you don't want to define `_POSIX_SOURCE`. So, don't—it's entirely optional. That's what I had to do to get the example code in this book to run on FreeBSD.[16]

Hmmm. We fried your brain explaining these symbols and all the funny numbers and then told you that you probably don't want to use them. Welcome to the world of standards!

1.5.3 Asking the System What It Has

Just because you've asked for a certain level with the `_POSIX_SOURCE`, `_POSIX_C_SOURCE`, and `_XOPEN_SOURCE` symbols, that doesn't mean that that's what you're going to get. If the system is only POSIX1993, that's all it can be. If that's all you need, then fine. If not, you need to refuse to compile (with an `#error` directive), disable an optional feature of your application, or enable an alternative implementation.

The way you find out what the OS has to offer, after you've included unistd.h, is to check the `_POSIX_VERSION` symbol and, if you're on a SUS system, the `_XOPEN_UNIX` and `_XOPEN_VERSION` symbols. The four "SOURCE" symbols in the previous section were you talking to the system, and now the system is talking back to you.

16. Also, most compilers and operating systems have additional symbols to bring in useful nonstandard features, such as `_GNU_SOURCE` for gcc or `__EXTENSIONS__` for Solaris. Check your system's documentation.

Of course, these symbols may not even be defined, although that would be strange for _POSIX_VERSION, since you've already successfully included unistd.h. So, you have to test carefully, and the way to do it is with a preprocessor command like this:

```
#if _POSIX_VERSION >= 199009L
    /* we have some level of POSIX
#else
#error "Can be compiled only on a POSIX system!"
#endif
```

If _POSIX_VERSION is undefined, it will be replaced by 0.

_POSIX_VERSION takes on the numbers (e.g., 199009L) from the table in the previous section. If you have at least a SUS1 system, _XOPEN_UNIX will be defined (with no value), but it's better to check against _XOPEN_VERSION, which will be equal to 4, 500, or 600, or perhaps someday an even larger number.

Let's write a little program that prints what the headers are telling us after we've requested SUS2 compatibility (the header suvreq.h is in the previous section):

```
#define SUV_SUS2

#include "suvreq.h"
#include <unistd.h>
#include <stdio.h>

int main(void)
{
    printf("Request:\n");
#ifdef _POSIX_SOURCE
    printf("\t_POSIX_SOURCE defined\n");
    printf("\t_POSIX_C_SOURCE = %ld\n", (long)_POSIX_C_SOURCE);
#else
    printf("\t_POSIX_SOURCE undefined\n");
#endif

#ifdef _XOPEN_SOURCE
    #if _XOPEN_SOURCE +0 == 0
        printf("\t_XOPEN_SOURCE defined (no value)\n");
    #else
        printf("\t_XOPEN_SOURCE = %d\n", _XOPEN_SOURCE);
    #endif
#else
    printf("\t_XOPEN_SOURCE undefined\n");
#endif
```

```
#ifdef _XOPEN_SOURCE_EXTENDED
    printf("\t_XOPEN_SOURCE_EXTENDED defined\n");
#else
    printf("\t_XOPEN_SOURCE_EXTENDED undefined\n");
#endif

    printf("Claims:\n");
#ifdef _POSIX_VERSION
    printf("\t_POSIX_VERSION = %ld\n", _POSIX_VERSION);
#else
    printf("\tNot POSIX\n");
#endif

#ifdef _XOPEN_UNIX
    printf("\tX/Open\n");
    #ifdef _XOPEN_VERSION
        printf("\t_XOPEN_VERSION = %d\n", _XOPEN_VERSION);
    #else
        printf("\tError: _XOPEN_UNIX defined, but not "
          "_XOPEN_VERSION\n");
    #endif
#else
    printf("\tNot X/Open\n");
#endif
    return 0;
}
```

When we ran this on a Solaris 8 system, this was the output:

```
Request:
        _POSIX_SOURCE defined
        _POSIX_C_SOURCE = 199506
        _XOPEN_SOURCE = 500
        _XOPEN_SOURCE_EXTENDED defined
Claims:
        _POSIX_VERSION = 199506
        X/Open
        _XOPEN_VERSION = 500
```

So, Solaris is willing to be SUS2. And, if we change the first line to define `SUV_SUS1` instead, the output is now

```
Request:
        _POSIX_SOURCE defined
        _POSIX_C_SOURCE = 2
        _XOPEN_SOURCE defined (no value)
        _XOPEN_SOURCE_EXTENDED defined
Claims:
        _POSIX_VERSION = 199506
        X/Open
        _XOPEN_VERSION = 4
```

which means that Solaris is also set up to behave like a SUS1 system, hiding any features newer than SUS1.

Running the program on FreeBSD 4.6 produced this output:

```
Request:
        _POSIX_SOURCE defined
        _POSIX_C_SOURCE = 2
        _XOPEN_SOURCE defined (no value)
        _XOPEN_SOURCE_EXTENDED defined
Claims:
        _POSIX_VERSION = 199009
        Not X/Open
```

This means that FreeBSD is claiming only to be POSIX1990, despite what we'd like it to be.

As I said, you probably don't want to make this request of FreeBSD because that just cuts out too much that they've included, some of which probably conforms to standards later than POSIX1990. When we reran our program without the suvreq.h header at all, this is what we got:

```
Request:
        _POSIX_SOURCE undefined
        _XOPEN_SOURCE undefined
        _XOPEN_SOURCE_EXTENDED undefined
Claims:
        _POSIX_VERSION = 199009
        Not X/Open
```

Still claiming to be POSIX1990, of course. Interestingly, when we do the same thing on Solaris, it decides it's going to be XPG3, so we might call that its default level. Why the default isn't SUS2, especially since Solaris is branded as UNIX 98, is a mystery. Here's the Solaris output:

```
Request:
        _POSIX_SOURCE undefined
        _XOPEN_SOURCE undefined
        _XOPEN_SOURCE_EXTENDED undefined
Claims:
        _POSIX_VERSION = 199506
        X/Open
        _XOPEN_VERSION = 3
```

On SuSE Linux 8.0 (Linux version 2.4, glibc 2.95.3), when we defined `SUV_SUS2` we got this:

```
Request:
        _POSIX_SOURCE defined
        _POSIX_C_SOURCE = 199506
        _XOPEN_SOURCE = 500
        _XOPEN_SOURCE_EXTENDED defined
Claims:
        _POSIX_VERSION = 199506
        X/Open
        _XOPEN_VERSION = 500
```

So Linux is willing to be a SUS2 system. But its default is SUS1, since we got this when we didn't include suvreq.h at all:

```
Request:
        _POSIX_SOURCE defined
        _POSIX_C_SOURCE = 199506
        _XOPEN_SOURCE undefined
        _XOPEN_SOURCE_EXTENDED undefined
Claims:
        _POSIX_VERSION = 199506
        X/Open
        _XOPEN_VERSION = 4
```

Compare this output to the output from Solaris when we didn't include the header—this time Linux went ahead and defined `_POSIX_SOURCE` for us when we included unistd.h. That probably makes some sense, since defining it has always been a POSIX requirement.

Darwin is really modest—it claims only POSIX1988 (with suvreq.h included):

```
Request:
        _POSIX_SOURCE defined
        _POSIX_C_SOURCE = 199506
        _XOPEN_SOURCE = 500
        _XOPEN_SOURCE_EXTENDED defined
Claims:
        _POSIX_VERSION = 198808
        Not X/Open
```

Anyway, for all the example code in this book we define `SUV_SUS2` before including suvreq.h, except for FreeBSD and Darwin, where we don't make any request at all. Which leads to the question, how do we know we're on FreeBSD or Darwin? (Answer in Section 1.5.8.)

1.5.4 Checking for Options

I said that POSIX1996 (IEEE Std 1003.1-1996) included real-time and threads, but that doesn't mean that systems that conform to it are required to support real-time and threads, because those are optional features. It means that if they don't, they have to say so in a standard way, as I'm about to explain.

It turns out that as complex as the symbology has been already, it's much too coarse to tell us about options. For that, a whole bunch of additional symbols have been defined for option groups (e.g., threads, POSIX messages) and, for some of them, suboptions. I won't explain each one here, but just indicate how to check, in general, whether an option is supported. My explanation is somewhat simplified, as the option-checking rules have evolved from POSIX1990 to SUS1 to SUS3 into something pretty complicated. (See the Web page [Dre2003] for an in-depth discussion of the options, cross-referenced with the affected functions.)

In practice, I've encountered this unpleasant state of affairs:

- The principal commercial systems, such as Solaris, support all the options. They also handle the option-checking symbols correctly, although, since the options are supported, not checking would also work.
- The open-source systems, Linux, FreeBSD, and Darwin, tend not to support many options, so it's important to do the option checking. However, they don't handle the option checking correctly, so the checking code you put in often doesn't work.

Clearly, no amount of work by the standards groups will improve the situation if the OS developers don't follow the current standards.

Anyway, each option has a preprocessor symbol defined in unistd.h that can be tested to see if the option is supported, such as `_POSIX_ASYNCHRONOUS_IO` for the Asynchronous Input and Output option, which includes the system calls `aio_read`, `aio_write`, and a few others (more on them in Chapter 3). If your program uses this option, you first need to test the symbol, like this:

```
#if _POSIX_ASYNCHRONOUS_IO <= 0
    /* substitute code, or message, or compile failure with #error */
#else
    /* code that uses the feature */
#endif
```

What this means is that you can't use those system calls and you can't include the associated header (aio.h) if the symbol is undefined or defined with a value of 0 or

less. If the symbol is defined with a value greater than 0, you can assume the feature is always supported.[17]

Some options depend on what file you're using, and there the rules are different. For example, if `_POSIX_ASYNCHRONOUS_IO` is supported, you then have to check the suboption `_POSIX_ASYNC_IO` to see if asynchronous I/O is supported on the file you want to use it on. The rules for suboptions vary a bit from option to option, but in most cases if the option is file related, you first check to see whether it's defined at all. If it is, a value of –1 means it isn't supported for any file, and any other value means it's supported for all files. If it's undefined, you have to call `pathconf` or `fpathconf` to test it (Section 1.5.6).

The file-dependent option rules I just explained were for SUS2. For pre-SUS2—POSIX1993 and POSIX1996—the rules were different: You only had to check the file if the top-level option (`_POSIX_ASYNCHRONOUS_IO` in our example) was undefined; if it was defined with a value other than –1, you could assume support on all files.

Linux is a little confused: It claims to be SUS2, yet it follows the pre-SUS2 rules.

One way to deal with this option mess is by encapsulating the option-checking into a function for each group of optional system calls that you use. For example, here's one to test to see if the asynchronous I/O functions are supported, complete with the adjustment for Linux (Section 1.5.6 explains `pathconf`):

```
typedef enum {OPT_NO = 0, OPT_YES = 1, OPT_ERROR = -1} OPT_RETURN;

OPT_RETURN option_async_io(const char *path)
{
#if _POSIX_ASYNCHRONOUS_IO <= 0
    return OPT_NO;
#elif _XOPEN_VERSION >= 500 && !defined(LINUX)
    #if !defined(_POSIX_ASYNC_IO)
        errno = 0;
        if (pathconf(path, _PC_ASYNC_IO) == -1)
            if (errno == 0)
                return OPT_NO;
            else
                EC_FAIL
        else
            return OPT_YES;
```

17. The test as I've coded it is oversimplified because it doesn't treat the undefined or 0 case as special. For some symbols and some versions of SUS, you get headers and function stubs so you can test at run-time with `sysconf`. But some systems, notably Linux, omit the stubs in the undefined case. So the test as we've shown it fails to account for systems where the symbol is zero, but `sysconf` will report that the feature is supported.

```
    EC_CLEANUP_BGN
        return OPT_ERROR;
    EC_CLEANUP_END
    #elif _POSIX_ASYNC_IO == -1
        return OPT_NO;
    #else
        return OPT_YES;
    #endif /* _POSIX_ASYNC_IO */
#elif _POSIX_VERSION >= 199309L
    return OPT_YES;
#else
    errno = EINVAL;
    return OPT_ERROR;
#endif /* _POSIX_ASYNCHRONOUS_IO */
}
```

Note that you can't use the function to find out whether code that uses an option will compile. For that you have to use the preprocessor, and hence the option symbol (e.g., `_POSIX_ASYNCHRONOUS_IO`) directly. So you still have to do something like this:

```
#if _POSIX_ASYNCHRONOUS_IO <= 0
    /* substitute code, or message, or compile failure with #error */
#else
    /*
        code that calls option_async_io and uses the
        option if it returns OPT_YES
    */
#endif
```

Just to recap in case you got lost: The option `_POSIX_ASYNCHRONOUS_IO` is for testing whether the system supports asynchronous I/O at all; on a SUS2 system, to find out whether it's supported for the particular file we want to use it on, we have to then check `_POSIX_ASYNC_IO`, possibly at run-time with a call to `pathconf` or `fpathconf`.

Don't even try to learn the symbols for all the options—there are too many of them, most of which you'll never use, and there are too many irregularities in how they work. Instead, for most optional system calls that I cover in this book, I'll indicate what symbols have to be checked.[18] But I usually won't show the actual option-checking in my example programs.

18. Don't confuse optional, which is still standardized, with nonstandard. I cover only a handful of nonstandard calls in this book.

1.5.5 `sysconf` System Call

You use the `sysconf` system call not only to check at run-time for support of optional features (Section 1.5.4), but also to see what various implementation-dependent limits are, such as the number of files you can have open at once. You can see a list of things it can query from the man page or from [SUS2002].

sysconf—get system option or limit

```
#include <unistd.h>

long sysconf(
    int name           /* option or limit name */
);
/* Returns option/limit value or -1 (sets errno on error) */
```

The only error you can make with `sysconf`, assuming your code compiles, is to use a symbol it doesn't know about, in which case it returns –1 and sets `errno` to `EINVAL`. Otherwise, if `errno` wasn't set, a –1 return just means that the option isn't supported or, if you're testing a limit, that there is no limit. So you have to set `errno` to zero before calling `sysconf`. Also, don't use the `ec_neg1` macro, because it thinks all –1 returns are errors. In the following example that checks a limit, we do the checking ourselves and use `EC_FAIL` to trigger the error. (The "ec" stuff was explained in Section 1.4.2.)

```
    long value;

#if defined(_SC_ATEXIT_MAX)
    errno = 0;
    if ((value = sysconf(_SC_ATEXIT_MAX)) == -1)
        if (errno == 0)
            printf("max atexit registrations: unlimited\n");
        else
            EC_FAIL
    else
        printf("max atexit registrations: %ld\n", value);
#else
    printf("_SC_ATEXIT_MAX undefined\n");
#endif
```

On Linux we got:

```
max atexit registrations: 2147483647
```

which is the biggest value a `long` can have (the same as the Standard C symbol `LONG_MAX`). It probably just means that Linux keeps the registrations in a linked list rather than a fixed-sized array. But on FreeBSD we got:

```
_SC_ATEXIT_MAX undefined
```

which is also OK, as the symbol isn't part of POSIX1990, which is all FreeBSD claims to conform to. It just illustrates the importance of checking to see whether the symbol is defined. (Since Standard C says that the minimum value for `atexit` is 32, we can assume that the number for FreeBSD is 32, although we got that from the C Standard, not from `sysconf`.)

1.5.6 `pathconf` and `fpathconf` System Calls

`sysconf` is only for checking system-wide options and limits. Others depend on what file you're talking about, and for that there are two very similar functions named `fpathconf` and `pathconf`:

pathconf—get system option or limit by path

```
#include <unistd.h>

long pathconf(
      const char *path,      /* pathname */
      int name               /* option or limit name */
);
/* Returns option/limit value or -1 (sets errno on error) */
```

fpathconf—get system option or limit by file descriptor

```
#include <unistd.h>

long fpathconf(
      int fd,                /* file descriptor */
      int name               /* option or limit name */
);
/* Returns option/limit value or -1 (sets errno on error) */
```

These two functions take the same second argument and return the same result; you use `fpathconf` when you have an open file, and `pathconf` if you have a path name to a file that may or may not be open. The valid values for `name` are in [SUS2002] or on your system's `man` page.

The interpretation of the return value is the same as for `sysconf` (previous section). In particular, a –1 means an error only if `errno` changed, so you have to set it to zero before making the call. We showed code using `pathconf` in Section 1.5.4.

1.5.7 `confstr` System Call

The call `confstr` is used like `sysconf`, but it's for string values, rather than numeric ones:

confstr—get configuration string

```
#include <unistd.h>

size_t confstr(
      int name,          /* option or limit name */
      char *buf,         /* returned string value */
      size_t len         /* size of buf */
);
/* Returns size of value or 0 on error (sets errno on error) */
```

When you make the call, you set `len` to the size of the buffer you pass in as the `buf` argument; on output it fills the buffer with a NUL-terminated string, truncating it if there isn't room for the whole thing. The size that would be needed for the whole string is returned as the value. If 0 is returned, `errno` indicates an error if it was changed. If `errno` wasn't changed, it means that name was invalid. So, as with the previous few functions, you have to set `errno` to zero before the call.

1.5.8 Checking for a Specific OS

Checking for features the POSIX/SUS way is best, but sometimes you really do need to know whether you're on Solaris, HP/UX, AIX, FreeBSD, Darwin, Linux, or whatever. Sometimes you even need to know the major and minor version numbers. We already saw one case where this is legitimate: We don't want to define `_POSIX_SOURCE` on a FreeBSD or Darwin system. Other cases are where an OS has bugs, or has a feature (e.g., System V messages) that goes beyond the standard they're claiming to conform to (see next section).

There's no portable way to check for the OS and, on some systems (e.g., Solaris), no nonportable way, either. On most systems a symbol may be set once you include a particular header, but we want the indication very early, before we've even included unistd.h, which should usually be the first POSIX or Standard C header to be included. So, we set a symbol at compile time, making sure it's different for each system, like this:

```
$ gcc -DSOLARIS -c xyz.c
```

Actually, our command lines are more complicated than that, and they're inside makefiles, but you get the idea.

1.5.9 Bonus Features

Systems like FreeBSD and Darwin are pretty conservative in their claims of POSIX conformance, yet they're much more complete than a pure POSIX 1988 or POSIX1990 system, which included neither socket-style networking nor System V IPC. FreeBSD has those features, which mostly work just as they're supposed to, and lots more.[19]

It isn't possible for an OS to set a POSIX level and also an option symbol to indicate that it has bonus features, because (1) some features, like those I listed, were never in any POSIX standard prior to SUS3 (POSIX2001, if you like) and (2) in SUS3, they are not optional, so there's no symbol for them. FreeBSD can't solve the problem by claiming to conform to, say, SUS1, because it doesn't.

At the same time, we certainly don't want to put tests for the FREEBSD symbol all over our code, because that's not what we mean: We mean "sockets" or "System V IPC." Gratuitous OS dependencies make porting the code, especially by a different programmer, difficult because the porter has no idea what point you're making with the OS-dependent test. Somebody porting to, say, Linux may be an expert on Linux but be clueless about the peculiarities of FreeBSD.

So, we need to make up even more symbols for what we call bonus features: those that are standard someplace, but not in the standard to which the OS claims to conform. I'll introduce these as we go along and you'll sometimes see them in the example programs, with code something like this:

```
#if (defined(_XOPEN_SOURCE) && _XOPEN_SOURCE >= 4) || defined(BSD_DERIVED)
#define HAVE_SYSVMSG
#endif
```

Our common header (see next section) defines `BSD_DERIVED` if `FREEBSD` or `DARWIN` is defined, as explained in Section 1.5.8.

19. Darwin 6.6, the version in Mac OS X 10.2.6, is missing System V messages, although it has the other System V IPC features.

You might think of what we're doing as formalizing the idea of "partial conformance," an idea that makes the standards people wince, but one that OS implementors and application writers find essential.

1.6 Common Header File

All of the examples in this book include the common header file defs.h that contains our request for SUV_SUS2 and the include for suvreq.h, which I talked about in Section 1.5.3. It also has many of the standard includes that we often need; we don't worry about including any extras if a program doesn't need them all. Other, less common, headers we include as we need them. I won't show it here, but defs.h also includes the prototypes for utility functions like `syserrmsg` (Section 1.4.1). Here's most of defs.h, or at least what it looked like when this was being written; the latest version is on the Web site [AUP2003]:

```
#if defined(FREEBSD) || defined(DARWIN)
#define BSD_DERIVED
#endif

#if !defined(BSD_DERIVED) /* _POSIX_SOURCE too restrictive */
#define SUV_SUS2
#include "suvreq.h"
#endif

#ifdef __GNUC__
#define _GNU_SOURCE /* bring GNU as close to C99 as possible */
#endif

#include <unistd.h>

#ifndef __cplusplus
#include <stdbool.h> /* C99 only */
#endif
#include <sys/types.h>
#include <time.h>
#include <limits.h>
#ifdef SOLARIS
#define _VA_LIST /* can't define it in stdio.h */
#endif
#include <stdio.h>
#ifdef SOLARIS
#undef _VA_LIST
#endif
#include <stdarg.h> /* this is the place to define _VA_LIST */
```

```
#include <stdlib.h>
#include <stddef.h>
#include <string.h>
#include <ctype.h>
#include <errno.h>
#include <fcntl.h>
#include <assert.h>
#include "ec.h"
```

The `#ifdef SOLARIS` stuff about two-thirds of the way down is a perfect example of why you sometimes need OS-dependent code. There's disagreement about whether the "va" macros (used for variable argument lists in C) go in stdio.h or stdarg.h, and gcc on Solaris does it differently from the other systems. So, I used a little trick to effectively disable the definitions in stdio.h. Not pretty, and worth some effort to avoid, but sometimes there's just no other way, and you need to get on with your work.

The order of the includes is a little strange, and probably could be partially alphabetized, but in a few cases I came upon order dependencies and had to juggle things around a bit to get a clean compile on all the systems I tried. What you see is where things ended up. There aren't supposed to be order dependencies because a header should include other headers it depends on, but there are.

1.7 Dates and Times

Two kinds of time are commonly used in UNIX: *calendar time* and *execution time*. This section includes system calls and functions for getting the time and manipulating it. There are also interval timers that you can set; they're discussed in Section 9.7.

1.7.1 Calendar Time

Calendar time is used for the access, modification, and status-change times of files, for recording when a user logged in, for displaying the current date and time, and so on.

Calendar time is usually in one of four forms:

- An arithmetic type, `time_t`, which is a count of the number of seconds since the *epoch,* which is traditionally set at midnight, January 1, 1970

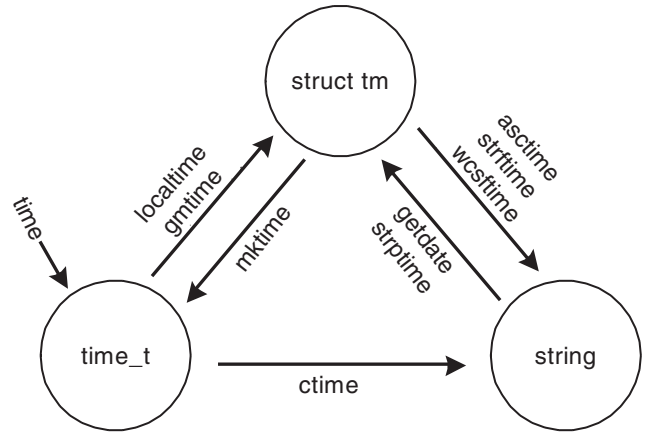


Figure 1.1 Time-conversion functions.

UTC.[20] It's a good way to store time in files and to pass it around among functions, because it's compact and time-zone-independent.

- A structure type, `struct timeval`, which holds a time in seconds and microseconds. (Used for both calendar and execution times.)
- A structure type, `struct tm`, which has time broken down into year, month, day, hour, minute, second, and a few other things.
- A string, such as `Tue Jul 23 09:44:17 2002`.

There's a full set of library functions, most from Standard C, to convert between the three forms. The function names are a crazy collection that make no sense at all (nothing simple like `cvt_tm_to_time`); Figure 1.1 provides a map that shows what the nine conversion functions do. The tenth function, `time`, is for getting the current time.

Here are the `tm` and `timeval` structures[21] followed by the synopses of the primary calendar-time functions:[22]

20. Universal Coordinated Time. Formerly known as GMT, or Greenwich Mean Time.
21. There's one obsolete structure, `timeb`, which holds the time in seconds and milliseconds, and an obsolete function, `ftime`, to fill it. Use `gettimeofday` instead.
22. Some of the Standard C functions also have a re-entrant form (e.g., `ctime_r`) that doesn't use a static buffer, but these aren't covered in this book.

struct tm—structure for broken-down time

```
struct tm {
     int tm_sec;            /* second [0,61] (up to 2 leap seconds) */
     int tm_min;            /* minute [0,59] */
     int tm_hour;           /* hour [0,23] */
     int tm_mday;           /* day of month [1,31] */
     int tm_mon;            /* month  [0,11] */
     int tm_year;           /* years since 1900 */
     int tm_wday;           /* day of week [0,6] (0 = Sunday) */
     int tm_yday;           /* day of year [0,365] */
     int tm_isdst;          /* daylight-savings flag */
};
```

struct timeval—structure for gettimeofday

```
struct timeval {
     time_t tv_sec;         /* seconds */
     suseconds_t tv_usec;   /* microseconds */
};
```

time—get current date and time as time_t

```
#include <time.h>

time_t time(
     time_t *t              /* NULL or returned time */
);
/* Returns time or -1 on error (errno not defined) */
```

gettimeofday—get current date and time as timeval

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

int gettimeofday(
     struct timeval *tvalbuf,      /* returned time */
     void *dummy                   /* always NULL */
);
/* Returns 0 on success or -1 (maybe) on error (may set errno) */
```

localtime—convert time_t to local broken-down time

```
#include <time.h>

struct tm *localtime(
     const time_t *t        /* time */
);
/* Returns broken-down time or NULL on error (sets errno) */
```

gmtime—convert time_t to UTC broken-down time

```
#include <time.h>

struct tm *gmtime(
     const time_t *t        /* time */
);
/* Returns broken-down time or NULL on error (sets errno) */
```

mktime—convert local broken-down time to time_t

```
#include <time.h>

time_t mktime(
     struct tm *tmbuf           /* broken-down time */
);
/* Returns time or -1 on error (errno not defined) */
```

ctime—convert time_t to local-time string

```
#include <time.h>

char *ctime(
     const time_t *t            /* time */
);
/* Returns string or NULL on error (errno not defined) */
```

asctime—convert broken-down time to local-time string[23]

```
#include <time.h>

char *asctime(
     const struct tm *tmbuf     /* broken-down time */
);
/* Returns string or NULL on error (errno not defined) */
```

strftime—convert broken-down time to string with format

```
#include <time.h>

size_t strftime(
     char *buf,                 /* output buffer */
     size_t bufsize,            /* size of buffer */
     const char *format,        /* format */
     const struct tm *tmbuf     /* broken-down time */
);
/* Returns byte count or 0 on error (errno not defined) */
```

wcsftime—convert broken-down time to wide-character string with format

```
#include <wchar.h>
size_t wcsftime(
     wchar_t *buf,              /* output buffer */
     size_t bufsize,            /* size of buffer */
     const wchar_t *format,     /* format */
     const struct tm *tmbuf     /* broken-down time */
);
/* Returns wchar_t count or 0 on error (errno not defined) */
```

23. The SUS doesn't say that `asctime` returns NULL on an error, but it's a good idea to check anyway.

getdate—convert string to broken-down time with rules

```
#include <time.h>

struct tm *getdate(
     const char *s              /* string to convert */
);
/* Returns broken-down time or NULL on error (sets getdate_err) */
```

strptime—convert string to broken-down time with format

```
#include <time.h>

char *strptime(
     const char *s,             /* string to convert */
     const char *format,        /* format */
     struct tm *tmbuf           /* broken-down time (output) */
);
/* Returns pointer to first unparsed char or NULL on error (errno not
defined) */
```

According to the standard, none of these functions set `errno`, and most don't even return an error indication, although it's a good idea to test any returned pointers against `NULL` anyway. `getdate` is really weird: On an error it sets the variable or macro `getdate_err`, used nowhere else, to a number from 1 to 8 to indicate the nature of the problem; see [SUS2002] for details.

Most UNIX systems implement `time_t` as a `long` integer, which is often only 32 bits (signed); that's enough seconds to take us to 19-Jan-2038.[24] At some point it needs more bits, and it will be changed to something that's guaranteed to be 64 bits.[25] To prepare, always use a `time_t` (rather than, say, a `long`) and don't make any assumptions about what it is. A `time_t` is already inadequate for historical times, as on many systems it goes (with a negative number) only back to 1901.

Without making any assumptions about what a `time_t` is, it's difficult to subtract two of them, which is a common need, so there's a function just for that:

difftime—subtract two time_t values

```
#include <time.h>

double difftime(
     time_t time1,              /* time */
     time_t time0               /* time */
);
/* Returns time1 - time0 in seconds (no error return) */
```

24. If we keep to our aggressive 19-year revision cycle, this should be about when the fourth edition of this book will appear.
25. Clearly that's overkill, but with computers the number after 32 is often 64.

`gettimeofday` is a higher-resolution alternative to `time`, but none of the other functions take a `struct timeval` as an argument. They'll take the `tv_sec` field by itself, however, since it's a `time_t`, and you can deal with the microseconds separately. All you can safely assume about the type `tv_usec` is that it's a signed integer of some size. However, there is no reason for it to be any wider than 32 bits, as there are only 1,000,000 microseconds in a second, so casting the `tv_usec` field to a `long` will work fine. You can time an execution interval by bracketing it with two calls to `gettimeofday` and then use `difftime` on the `tv_sec` fields and naked subtraction on the `tv_usec` fields.

Some implementations (FreeBSD, Darwin, and Linux, but not Solaris) use the second argument to `gettimeofday` to return time-zone information, but that's nonstandard. Most implementations indicate an error with a –1 return, and even set `errno`. Those that don't, return zero always, so you can safely check the error return in all cases.

A `time_t` is always in terms of UTC, but broken-down time (`struct tm`) and strings can be in UTC or in local time. It's convenient to use local time when times are entered or printed out, and this complicates things. There must be some way for the computer to know what time zone the user is in and whether it is currently standard or daylight time. Worse, when printing past times, the computer must know whether it was standard or daylight time back *then*. The various functions concerned with local time deal with this problem fairly well.

The user's time zone is recorded in the environment variable `TZ` or as a system default if `TZ` isn't set. To provide portability, there's one function and three global variables to deal with time-zone and daylight-time settings:

tzset—set time zone information

```
#include <time.h>

extern int daylight;       /* DST? (not in FreeBSD/Darwin) */
extern long timezone;      /* timezone in secs (not in FreeBSD/Darwin) */
extern char *tzname[2];    /* timezone strings (not in FreeBSD/Darwin) */

void tzset(void);
```

An application can call `tzset` to somehow get the time zone (from `TZ` or wherever) and set the three globals, as follows:

- `daylight` is set to zero (`false`) if Daylight Savings Time should never be applied to the time zone, and non-zero (`true`) if it should. The switchover periods are built into the library functions in an implementation-defined way.
- `timezone` is set to the difference in seconds between UTC and local standard time. (Divide it by 3,600 to get the difference in hours.)
- `tzname` is an array of two strings, `tzname[0]` to be used to designate the local time zone during standard time, and `tzname[1]` for Daylight Savings Time.

The three globals were not in POSIX earlier than POSIX2002 and are not defined on FreeBSD or Darwin. But usually you don't need to call `tzset` or interrogate the globals directly. The standard functions do that as needed.

Here's an example anyway:

```
tzset();
printf("daylight = %d; timezone = %ld hrs.; tzname = %s/%s\n",
  daylight, timezone / 3600, tzname[0], tzname[1]);
```

with this output:

```
daylight = 1; timezone = 7 hrs.; tzname = MST/MDT
```

For all the details of how all the calendar-time functions are used, especially the ones that involve format strings, see [SUS2002] or a good book on Standard C, such as [Har2002]. Here are some miscellaneous points to keep in mind:

- Ignoring the possibility of an error return, `ctime(t)` is defined as `asctime(localtime(t))`.
- The `tm_sec` field of the `tm` structure can go to 61 (instead of just 59), to allow for occasional leap seconds.
- The `tm_year` field of the `tm` structure is an offset from 1900. It's still perfectly Y2K compatible, just strange (2002 is represented as 102).
- Just as strangely, `ctime` and `asctime` put a newline at the end of the output string.
- Several functions take a pointer to a `time_t`, when a call-by-value `time_t` would work as well. This is a holdover from the earliest days of UNIX when the C language didn't have a `long` data type and you had to pass in an array of two 16-bit `int`s.

- The `time_t` pointer argument to `time` is totally unnecessary and should just be `NULL`. It looks like it supplies a buffer for `time` to use, but no buffer is needed, as the returned `time_t` is an arithmetic type.
- All of the functions presented here that return a pointer to a `tm` structure use a static buffer internally on most implementations, so use the buffer or copy it before you call another such function. There are re-entrant forms with an `_r` suffix that you can use in an multithreaded program if you need to.
- The functions that use formats, `strftime`, `wcsftime`, `getdate`, and `strptime` are very complicated, but very powerful, and are well worth learning. The first two are in Standard C; the last two are in the SUS. `getdate` is not in FreeBSD or Darwin, although there is a version of it in the GNU C library that you can install if you want.
- Use `getdate` to get dates and times that users type in, and `strptime` or `getdate` when reading data files containing dates. The reason is that `strptime` takes only a single format, which makes it too difficult for users to format the date and time correctly. `getdate` uses a whole list of formats and tries them one-by-one looking for a match.

1.7.2 Execution Time

Execution time is used to measure timing intervals and process-execution times, and for accounting records. The kernel automatically records each process's execution time. Interval times come in various subsecond units ranging from nanoseconds to hundredths of a second.

The execution-time types and functions are just as confusing as those for calendar time. First, here are the principal types:

- `clock_t`, in Standard C, an arithmetic type (typically a `long`, but don't assume so) which is a time interval in units of `CLOCKS_PER_SEC` or clock ticks
- `struct timeval`, which holds an interval in seconds and microseconds (also used for calendar time, and explained in detail in the previous section)
- `struct timespec`, which holds an interval in seconds and nanoseconds

Here's the structure for `timespec` (we already showed `timeval`), which is defined in the header <time.h>:

struct timespec—structure for time in seconds and nanoseconds

```
struct timespec {
     time_t tv_sec;          /* seconds */
     long tv_nsec;           /* nanoseconds */
};
```

There are three primary functions for measuring a time interval, `gettimeofday`, detailed in the previous section, `clock`, and `times`. The latter two and the structure used by `times` are:

clock—get execution time

```
#include <time.h>

clock_t clock(void);
/* Returns time in CLOCKS_PER_SEC or -1 on error (errno not defined) */
```

times—get process and child-process execution times

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

clock_t times(
     struct tms *buffer     /* returned times */
);
/* Returns time in clock ticks or -1 on error (sets errno) */
```

struct tms—structure for times system call

```
struct tms {
     clock_t tms_utime;     /* user time */
     clock_t tms_cutime;    /* user time of terminated children */
     clock_t tms_stime;     /* sys time */
     clock_t tms_cstime;    /* sys time of terminated children */
};
```

`times` returns the time elapsed since some point in the past (usually since the system was booted), sometimes called the *wall-clock* or *real* time. This is different from the time returned from the functions in the previous section, which returned the time since the epoch (normally 1-Jan-1970).

`times` uses units of clock ticks; you get the number of clock ticks per second with `sysconf` (Section 1.5.5) like this:

```
clock_ticks = sysconf(_SC_CLK_TCK);
```

Additionally, `times` loads the `tms` structure with more specific information:

- `tms_utime` (user time) is the time spent executing instructions from the process's user code. It's CPU time only and doesn't include time spent waiting to run.
- `tms_stime` (system time) is the CPU time spent executing system calls on behalf of the process.
- `tms_cutime` (child user time) is the total of user CPU times for all the process's child processes that have terminated and for which the parent has issued a `wait` system call (explained fully in Chapter 5).
- `tms_cstime` (child system time) is the total of system CPU times for terminated and waited-for child processes.

Although `clock` returns the same type as `times`, the value is CPU time *used* since the process started, not real time, and the units are given by the macro `CLOCKS_PER_SEC`, not clock ticks. It's equivalent (except for the different units) to `tms_utime` + `tms_stime`.

`CLOCKS_PER_SEC` is defined in the SUS as 1,000,000 (i.e., microseconds), but non-SUS systems may use a different value, so always use the macro. (FreeBSD defines `CLOCKS_PER_SEC` as 128; Darwin as 100.)

If `clock_t` is a 32-bit signed integer, which it often is, and uses units of microseconds, it wraps after about 36 minutes, but at least `clock` starts timing when the process starts. `times`, operating in clock ticks, can run longer, but it also starts earlier—much earlier if the UNIX system has been running for weeks or months, which is very common. Still, a 32-bit signed `clock_t` operating in units of ticks won't wrap for about 250 days, so a semiannual reboot, if you can arrange for one, solves the problem.

However, you can't assume that `clock_t` is a 32-bit signed integer, or that it's signed, or even that it's an integer. It could be an `unsigned long`, or even a `double`.

We'll use `times` throughout this book to time various programs because it gives us the real time and the user and system CPU times all at once. Here are two handy functions that we'll use—one to start timing, and one to stop and print the results:

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

static struct tms tbuf1;
static clock_t real1;
static long clock_ticks;
```

```
void timestart(void)
{
    ec_neg1( real1 = times(&tbuf1) )
    /* treat all -1 returns as errors */
    ec_neg1( clock_ticks = sysconf(_SC_CLK_TCK) )
    return;

EC_CLEANUP_BGN
    EC_FLUSH("timestart");
EC_CLEANUP_END
}

void timestop(char *msg)
{
    struct tms tbuf2;
    clock_t real2;

    ec_neg1( real2 = times(&tbuf2) )
    fprintf(stderr,"%s:\n\t\"Total (user/sys/real)\", %.2f, %.2f, %.2f\n"
      "\t\"Child (user/sys)\", %.2f, %.2f\n", msg,
      ((double)(tbuf2.tms_utime + tbuf2.tms_cutime) -
      (tbuf1.tms_utime + tbuf1.tms_cutime)) / clock_ticks,
      ((double)(tbuf2.tms_stime + tbuf2.tms_cstime) -
      (tbuf1.tms_stime + tbuf1.tms_cstime)) / clock_ticks,
      (double)(real2 - real1) / clock_ticks,
      (double)(tbuf2.tms_cutime - tbuf1.tms_cutime) / clock_ticks,
      (double)(tbuf2.tms_cstime - tbuf1.tms_cstime) / clock_ticks);
    return;

EC_CLEANUP_BGN
    EC_FLUSH("timestop");
EC_CLEANUP_END
}
```

When we want to time something, we just call `timestart` at the beginning of the interval and `timestop` at the end, as in this example, which also includes calls to `gettimeofday` and `clock` for a comparison:

```
#define REPS 1000000
#define TV_SUBTRACT(t2, t1)\
    (double)(t2).tv_sec + (t2).tv_usec / 1000000.0 -\
    ((double)(t1).tv_sec + (t1).tv_usec / 1000000.0)

int main(void)
{
    int i;
    char msg[100];
    clock_t c1, c2;
    struct timeval tv1, tv2;
```

```
    snprintf(msg, sizeof(msg), "%d getpids", REPS);
    ec_neg1( c1 = clock() )
    gettimeofday(&tv1, NULL);
    timestart();
    for (i = 0; i < REPS; i++)
        (void)getpid();
    (void)sleep(2);
    timestop(msg);
    gettimeofday(&tv2, NULL);
    ec_neg1( c2 = clock() )
    printf("clock(): %.2f\n", (double)(c2 - c1) / CLOCKS_PER_SEC);
    printf("gettimeofday(): %.2f\n", TV_SUBTRACT(tv2, tv1));
    exit(EXIT_SUCCESS);

EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
EC_CLEANUP_END
}
```

This was the output on Linux (Solaris, FreeBSD, and Darwin gave similar results):

```
1000000 getpids:
        "Total (user/sys/real)", 0.72, 0.44, 3.19
        "Child (user/sys)", 0.00, 0.00
clock(): 1.16
gettimeofday(): 3.19
```

So `clock` measured 1.16,[26] exactly the sum of .72 and .44, and `gettimeofday` measured the same elapsed time as `times`. That's good, or we would have more explaining to do.

If you want more resolution than `times` can give you, use `getrusage` instead (Section 5.16).

1.8 About the Example Code

There's lots of example code in this book, and you're free to use it for your own educational purposes or even in commercial products, as long as you give credit where you normally would, such as in an About dialog box or the credits page of a manual or book. For details, see the Web page *www.basepath.com/aup/copyright.htm*.

26. In the 1985 edition of this book, we got a time of 1.43 seconds for only 1000 `getpids`!

Sometimes the examples in this book are abridged to save space (as in the error-handling code earlier in this chapter and the code in defs.h, but the complete files are on the Web at *www.basepath.com/aup* [AUP2003]. (There's lots of other information there, too, such as errata.) The code on the Web will be updated, as bugs are found after this book's publication, so if you're confused by something you think is wrong in the book, check the newer code first. Then, if you still think there's a bug, please email me at aup@basepath.com and let me know. If you have a different view of something I've said or a better way to do things, or even if you just want to say that this is the best book you've ever read, I'd love to get those emails, too.

The examples are written in C99, but the new stuff (e.g., `intptr_t`, `bool`) shouldn't present a problem if your compiler isn't at that level yet, as you can define your own macros and `typedef`s. Most of the code has been tested with the gcc compiler on SuSE Linux 8.0 (based on Linux 2.4), FreeBSD 4.6, and Solaris 8, all running on Intel CPUs, and Darwin 6.6 (the UNIX part of Mac OS X 10.2.6) on a PowerPC-based iMac. As time goes on, more testing on more systems will be done, and if there are any changes they'll be posted on the Web site.

1.9 Essential Resources

Here's a list of the essential resources (besides this book!) you'll want to have handy while you're programming:

- A good **reference book for your language**. For C, the best is *C: A Reference Manual*, by Harbison and Steele [Har2002]; for C++, perhaps *The C++ Programming Language*, by Bjarne Stroustrup [Str2000]. The idea is to have one book with all the answers, even if finding them is a bit of work. After you know the book, it's no longer any work, and you save lots of time and desk space with only one book instead of several.
- The **Open Group SUS** (Single UNIX Specification) at:

  www.unix.org/version3

  This is a fantastic site. You click on a function name in the left frame, and the spec shows in the right frame. It's also available on CD and on paper, but the Web version is free.
- The Usenet newsgroup **comp.unix.programmer**. Post almost any UNIX question, no matter how difficult, and you'll get an answer. It may not be

correct, but usually it will at least point you in the right direction. (To get to newsgroups, you need a mail client that can handle them and a news feed. Your ISP may provide one or, if not, you can pay a fee to *www.supernews.com* or *www.giganews.com*. Or, access them through Google Groups at *http://groups.google.com/grphp.*) There are some Web forums, too, such as *www.unix.com*, but they're not nearly as good as comp.unix.programmer. Stick to UNIX (or Linux or FreeBSD) questions—you'll get treated abruptly if you post a C or C++ question. There are separate newsgroups for those subjects (tens of thousands of newsgroups altogether). Google Groups also allows you to search through the Usenet archives to see if maybe someone in your parents' generation asked the same question 20 years ago.

- **Google**, the Web search site. Suppose the `sigwait` system call seems to be misbehaving on FreeBSD, or you're not sure whether Linux implements asynchronous I/O. You can get the answers the methodical way (from the FreeBSD or Linux sites, for example), or you can google around for the answer. About half the time, I get what I'm looking for not from the official source, but from some obscure forum post or private Web site that Google just happened to index.
- **GNU C Library**, a great source for coding examples, available for download at:

  `www.gnu.org/software/libc`

- The **`man` pages** for your system. Somewhat inconvenient to navigate through (the `apropos` command helps a little), but usually excellent once you've found the right page. It's a good idea to spend some time learning your way around so you'll be able to use them efficiently when the time comes.

Exercises

1.1. Write a program in C that displays "Hello Word" using the include file defs.h and at least one of the error-checking macros (`ec_neg1`, say). The purpose of this exercise is to get you to install all of the code for this book on your system, to make sure your C compiler is running OK, and to ensure the tools you'll need, such as a text editor and `make`, are in place. If you don't have access to a UNIX or Linux system, dealing with that problem is part of this exercise, too.

1.2. Write a program that displays what version of POSIX and the SUS your system conforms to. Do it for various requests (e.g., `SUV_SUS2`).

1.3. Write a program that displays all the `sysconf` values for your system. Figure out how to produce all the lines containing `sysconf` calls without having to type every symbol one-by-one.

1.4. Same as 1.3, but for `pathconf`. Design the program so that you can enter a path as an argument.

1.5. Same as 1.3, but for `confstr`.

1.6. Write a version of the `date` command, with no command-line options. For extra credit, handle as many options as you can. Base it on the SUS or one of the POSIX standards. No fair peeking at Gnu source or any other similar examples.

2 Basic File I/O

2.1 Introduction to File I/O

In this chapter we'll explore basic I/O on regular files. The I/O story continues in Chapter 3 with more advanced I/O system calls. I/O on special files is in Chapter 4, I/O on pipes in Chapter 6, I/O on named pipes in Chapter 7, and I/O on sockets in Chapter 8.

To get started I'll show a simple example that uses four system calls you may already be familiar with: `open`, `read`, `write`, and `close`. This function copies one file to another (like the `cp` command):

```
#define BUFSIZE 512

void copy(char *from, char *to)  /* has a bug */
{
    int fromfd = -1, tofd = -1;
    ssize_t nread;
    char buf[BUFSIZE];

    ec_neg1( fromfd = open(from, O_RDONLY) )
    ec_neg1( tofd = open(to, O_WRONLY | O_CREAT | O_TRUNC,
      S_IRUSR | S_IWUSR) )
    while ((nread = read(fromfd, buf, sizeof(buf))) > 0)
        if (write(tofd, buf, nread) != nread)
            EC_FAIL
    if (nread == -1)
        EC_FAIL
    ec_neg1( close(fromfd) )
    ec_neg1( close(tofd) )
    return;

EC_CLEANUP_BGN
    (void)close(fromfd); /* can't use ec_neg1 here! */
    (void)close(tofd);
EC_CLEANUP_END
}
```

Try to find the bug in this function (there's a clue in Section 1.4.1). If you can't, I'll point it out in Section 2.9.

I'll say just a few quick words about this function now; there will be plenty of time to go into the details later. The first call to `open` opens the input file for reading (as indicated by `O_RDONLY`) and returns a file descriptor for use in subsequent system calls. The second call to `open` creates a new file (because of `O_CREAT`) if none exists, or truncates an existing file (`O_TRUNC`). In either case, the file is opened for writing and a file descriptor is returned. The third argument to `open` is the set of permission bits to use if the file is created (we want read and write permission for only the owner). `read` reads the number of bytes given by its third argument into the buffer pointed to by its second argument. It returns the number of bytes read, zero on end-of-file, or –1 on error. `write` writes the number of bytes given by its third argument from the buffer given by its second argument. It returns the number of bytes written, which we treat as an error if it isn't equal to the number of bytes we asked to be written. Finally, `close` closes the file descriptors.

Rather than use `if` statements, `fprintf` calls, and `goto`s to deal with errors, I used the convenience macros `ec_neg1`, which leaves the function with an error if its argument is –1, and `EC_FAIL`, which always just leaves with an error. (Actually, they jump to the cleanup code, of which there is none in this case, delimited by `EC_CLEANUP_BGN` and `EC_CLEANUP_END`.) These were introduced back in Section 1.4.2.

2.2 File Descriptors and Open File Descriptions

Each UNIX process has a bunch of file descriptors at its disposal, numbered 0 through N, where N is the maximum. N depends on the version of UNIX and its configuration, but it's always at least 16, and much greater on most systems. To find out the actual value of N at run-time, you call `sysconf` (Section 1.5.5) with an argument of `_SC_OPEN_MAX`, like this:

```
printf("_SC_OPEN_MAX = %ld\n", sysconf(_SC_OPEN_MAX));
```

On a Linux 2.4 system we got 1024, on FreeBSD, 957, and on Solaris, 256. There probably aren't any current systems with just 16, except maybe for some embedded systems.

2.2.1 Standard File Descriptors

By convention, the first three file descriptors are already open when the process begins. File descriptor 0 is the *standard input,* file descriptor 1 is the *standard output,* and file descriptor 2 is the *standard error output,* which is usually open to the controlling terminal. Instead of numbers, it's better to use the symbols `STDIN_FILENO`, `STDOUT_FILENO`, and `STDERR_FILENO`.

A UNIX filter would read from `STDIN_FILENO` and write to `STDOUT_FILENO`; that way the shell can use it in pipelines. `STDERR_FILENO` should be used for important messages, since anything written to `STDOUT_FILENO` might go off down a pipe or into a file and never be seen should output be redirected, which is very commonly done from the shell.

Any of these standard file descriptors could be open to a file, a pipe, a FIFO, a device, or even a socket. It's best to program in a way that's independent of the type of source or destination, but this isn't always possible. For example, a screen editor probably won't work at all if the standard output isn't a terminal device.

The three standard file descriptors are ready to be used immediately in `read` and `write` calls. The other file descriptors are available for files, pipes, etc., that the process opens for itself. It's possible for a parent process to bequeath more than just the standard three file descriptors to a child process, and we'll see exactly that in Chapter 6 when we connect processes with pipes.

2.2.2 Using File Descriptors

Generally, UNIX uses file descriptors for anything that behaves in some way like a file, in that you can `read` it or `write` it or both. File descriptors aren't used for less-file-like communication mechanisms, such as message queues, which you can't `read` and `write` (there are specialized calls for the purpose).

There are only a few ways to get a fresh open file descriptor. We're not ready to dig into all of them right now, but it's helpful at least to list them:

- `open`, used for most things that have a path name, including regular and special files and named pipes (FIFOs)
- `pipe`, which creates and opens an un-named pipe (Chapter 6)
- `socket`, `accept`, and `connect`, which are used for networking (Chapter 8)

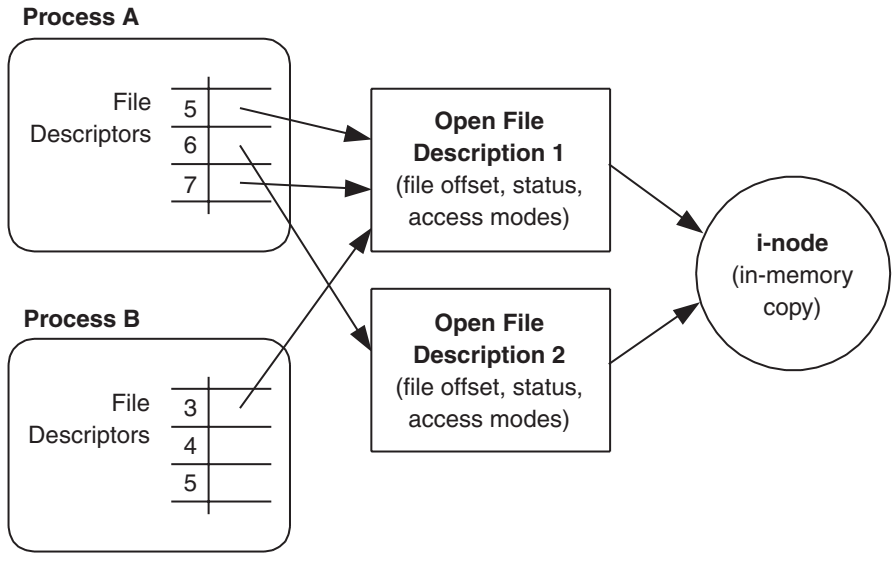


Figure 2.1 File descriptors, open file descriptions, and i-nodes.

There isn't any specific C type for a file descriptor (as there is, say, for a process ID), so we just use a plain `int`.[1]

2.2.3 Open File Descriptions and Sharing

A file descriptor is, as I said in Chapter 1, just an index into a per-process table. Each table entry points to a system-wide open file description (also known as a file-table entry), which in turn points to the file's data (via an in-memory copy of the i-node). Several file descriptors, even from different processes, can point to the same file description, as shown in Figure 2.1.

Each `open` or `pipe` system call creates a new open file description and a new file descriptor. In the figure, Process A opened the same file twice, getting file descriptors 5 and 6, and creating open file descriptions 1 and 2. Then, through a mechanism called file descriptor duplication (the `dup`, `dup2`, and `fork` system

1. I thought about introducing the type `fid_t` just for this book to make the examples a little more readable but then decided not to because in real code you'll see plain `int`, so why not get used to it.

calls do it), Process A got file descriptor 7 as a duplicate of 5, which means it points to the same open file description. Process B, a child of A, also got a duplicate of 5, which is its file descriptor 3.

We'll come back to Figure 2.1 from time to time because it tells us a lot about how things behave. For example, when we explain file offsets in Section 2.8, we'll see that Process A's file descriptors 5 and 7 and Process B's file descriptor 3 all share the same file offset because they share the same open file description.

2.3 Symbols for File Permission Bits

Recall from Section 1.1.5 that a file has 9 permission bits: read, write, and execute for owner, group, and others. We see them all the time in the output of the `ls` command:

```
-rwxr-xr-x    1 marc     users       29808 Aug  4 13:45 hello
```

Everyone thinks of the 9 bits as being together and in a certain order (owner, group, others), but that's not a requirement—only that there be 9 individual bits. So, from POSIX1988 on there have been symbols for the bits that are supposed to be used instead of octal numbers, which had been the traditional way to refer to them. The symbols have the form `S_I`**pwww** where **p** is the permission (`R`, `W`, or `X`) and **www** is for whom (`USR`, `GRP`, or `OTH`). This gives 9 symbols in all.

For instance, for the previous file, instead of octal 755, we write

```
S_IRUSR | S_IWUSR | S_IXUSR | S_IRGRP | S_IXGRP | S_IROTH | S_IXOTH
```

There are separate symbols for when a USR, GRP, or OTH has all three permissions, which are of the form `S_IRWX`**w**. This time **w** is just the first letter of the "whom," either `U`, `G`, or `O`. So the permissions could instead be written like this:

```
S_IRWXU | S_IRGRP | S_IXGRP | S_IROTH | S_IXOTH
```

The symbols are necessary to give implementors the freedom to do what they will with the bit positions, even if they're less readable and much more error-prone than just octal.[2] As any application you write is likely to use only a few combinations (e.g., one or two for data files it creates, and maybe another if it creates directories), it's a good idea to define macros for them just once, rather than using

2. One of the technical reviewers pointed out that even if octal were used, the kernel could map it to whatever the file system used internally.

long sequences of the S_I* symbols all over the place. For this book, we'll use just these, which are defined in defs.h (Section 1.6):

```
#define PERM_DIRECTORY   S_IRWXU
#define PERM_FILE        (S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)
```

Notice that we named the macros according to how we're going to use them, not according to the bits. Therefore, it's PERM_FILE, not something like PERM_RWURGO, because the whole point is being able to change the application's permissions policy by changing only one macro.

2.4 open and creat System Calls

open—open or create file

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

int open(
      const char *path,      /* pathname */
      int flags,             /* flags */
      mode_t perms           /* permissions (when creating) */
);
/* Returns file descriptor or -1 on error (sets errno) */
```

You use open to open an existing file (regular, special, or named pipe) or to create a new file, in which case it can only be a regular one. Special files are created with mknod (Section 3.8.2) and named pipes with mkfifo (Section 7.2.1). Once the file is opened, you can use the file descriptor you get back with read, write, lseek, close, and a bunch of other calls that I'll discuss in this and the next chapter.

2.4.1 Opening an Existing File

Let's talk first about opening an existing file, specified by path. If flags is O_RDONLY, it is opened for reading; if O_WRONLY, for writing; and if O_RDWR, for both reading and writing.[3]

3. Why three flags? Can't we scrap O_RDWR and just use O_RDONLY | O_WRONLY? No, because implementations have always defined O_RDONLY as zero, rather than as some bit.

The process needs read, or write, or both, kinds of permission to open the file, using the algorithm that was explained in Section 1.1.5. For example, if the effective user-ID of the process matches the owner of the file, and file's owner read and/or write permission bits have to be set.

For an existing file, the `perms` argument isn't used and is normally omitted entirely, so `open` is called with only two arguments.

The file offset (where reads and writes will occur) is positioned at the first byte of the file. More about this in Section 2.8.

Here's some code that opens an existing file:

```
int fd;

ec_neg1( fd = open("/home/marc/oldfile", O_RDONLY) )
```

There are lots of reasons why `open` can fail, and for many of them it pays to tell the user what the specific problem is. A path to a nonexistent file (`ENOENT`) requires a different solution from wrong permissions (`EACCES`). Our normal error-checking and reporting handles this very well (the "ec" macros were explained in Section 1.4.2).

The file descriptor returned by a successful `open` is the lowest-numbered one available, but normally you don't care what the number is. This fact is sometimes useful, however, when you want to redirect one of the standard file descriptors, 0, 1, or 2 (Section 2.2.1): You close the one you want to redirect and then open a file, which will then use the number (1, say) that you just made available.

2.4.2 Creating a New File

If the file doesn't exist, `open` will create it for you if you've ORed the `O_CREAT` flag into the flags. You can certainly create a new file opened only for reading, although that makes no sense, as there would be nothing to read. Normally, therefore, either `O_WRONLY` or `O_RDWR` are combined with `O_CREAT`. Now you do need the `perms` argument, as in this example:

```
ec_neg1( fd = open("/home/marc/newfile", O_RDWR | O_CREAT, PERM_FILE) )
```

The `perms` argument is only used if the file gets created. It has no effect on anything if the file already exists.

The permission bits that end up in a newly created file are formed by ANDing the ones in the system call with the complement of process's file mode creation mask, typically set at login time (with the `umask` command), or with the `umask` system call (Section 2.5). The "ANDing the complement" business just means that if a bit is set in the mask, it's cleared in the permissions. Thus, a mask of 002 would cause the `S_IWOTH` bit (write permission for others) to be cleared, even if the flag `S_IWOTH` appeared in the call to `open`. As a programmer, however, you don't usually think about the mask, since it's something users do to *restrict* permissions.

What if you create a file with the `O_WRONLY` or `O_RDWR` flags but the permission bits don't allow writing? As the file is brand new, it is *still* opened for writing. However, the next time it's opened it will already exist, so then the permission bits will control access as described in the previous section.

Sometimes you always want a fresh file, with no data in it. That is, if the file exists you want its data to be thrown away, with the file offset set at zero. The `O_TRUNC` flag does that:

```
ec_neg1( fd = open("/home/marc/newfile", O_WRONLY | O_CREAT | O_TRUNC,
  PERM_FILE) )
```

Since `O_TRUNC` destroys the data, it's allowed on an existing file only if the process has write permission, as it is a form of writing. Similarly, it can't be used if the `O_RDONLY` flag is also set.

For a *new* file (i.e., `O_CREAT` doing its thing), you need write permission in the parent directory, since a new link will be added to it. For an *existing* file, the permissions on the directory don't matter; it's the file's permissions that count. The way to think of this is to ask yourself, "What needs to be written to complete this operation?"

I might also mention that you need search (execute) permission on the intermediate directories in the path (i.e., `home` and `marc`). That, however, universally applies to any use of a path, and we won't say it every time.

`O_TRUNC` doesn't have to be used with `O_CREAT`. By itself it means "truncate the file to zero if it exists, and just fail if it doesn't exist." (Maybe there's a logging feature that gets turned on by creating a log file; no log file means no logging.)

The combination `O_WRONLY | O_CREAT | O_TRUNC` is so common ("create or truncate a file for writing") that there's a special system call just for it:

creat—create or truncate file for writing

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

int creat(
     const char *path,      /* pathname */
     mode_t perms           /* permissions */
);
/* Returns file descriptor  or -1 on error (sets errno) */
```

`open` for an existing file used only the first and second arguments (`path` and `flags`); `creat` uses just the first and third. In fact, `creat` could be just a macro:

```
#define creat(path, perms) open(path, O_WRONLY | O_CREAT | O_TRUNC, perms)
```

Why not just forget about `creat` and use `open` all the time—one system call to remember instead of two, and the flags always stated explicitly? Sounds like a great idea, and that's what we'll do in this book.[4]

We've skipped one important part of creating a new file: Who owns it? Recall from Section 1.1.5 that every file has an owner user-ID and an owner group-ID, which we just call owner and group for short. Here's how they're set for a new file:

- The owner is set from the effective user-ID of the process.
- The group is set to either the group-ID of the parent directory or the effective group-ID of the process.

Your application can't assume which method has been used for the group-ID, although it could find out with the `stat` system call (Section 3.5.1). Or, it could use the `chown` system call (Section 3.7.2) to force the group-ID to what it wants. But it's a rare application that cares about the group-ID at all.

There's another flag, `O_EXCL`, that's used with `O_CREAT` to cause failure if the file already exists. If `open` without the `O_CREAT` flag is "open if existent, fail if nonexistent," then `open` with `O_CREAT | O_EXCL` is the exact opposite, "create if nonexistent, fail if existent."

There's one interesting use for `O_EXCL`, using a file as a lock, which I'll show in the next section. Another use might be for a temporary file that's supposed to be deleted when the application exits. If the application finds it already existing, it

4. If you're interested in the history, `creat` is actually a very old system call that was important when an earlier form of `open` had only two arguments.

means the previous invocation terminated abnormally, so there's some cleanup or salvaging to do. You want the creation to fail in this case, and that's exactly what the O_EXCL flag will do for you:

```
int fd;

while ((fd = open("/tmp/apptemp", O_RDWR | O_CREAT | O_TRUNC | O_EXCL,
  PERM_FILE)) == -1) {
    if (errno == EEXIST) {
        if (cleanup_previous_run())
            continue;
        errno = EEXIST; /* may have been reset */
    }
    EC_FAIL /* some other error or can't cleanup */
}
/* file is open; go ahead with rest of app */
```

We don't want to use the ec_neg1 macro on open because we want to investigate errno for ourselves. The value EEXIST is specifically for the O_EXCL case. We call some function named cleanup_previous_run (not shown) and try again, which is why the while loop is there. If cleanup doesn't work, notice that we reset errno, as it's pretty volatile, and we have no idea what cleanup_previous_run may have done—it could be thousands of lines of code. (We could have used a for loop with only two iterations instead of the while, to catch that case when cleanup_previous_run returns true, keeping us in the loop, but has failed to unlink the file. But you get the idea.)

Our example messes up if the application is being run concurrently, maybe by two different users. In that case the temporary being around makes sense, and unlinking it would be a crime. If concurrent execution is allowed, we really need to redesign things so that each execution has a unique temporary file, and I'll show how to do that in Section 2.7. If we want to prevent concurrent execution entirely, we need some sort of locking mechanism, and that answer is in the next section.

2.4.3 Using a File as a Lock

Processes that want exclusive access to a resource can follow this protocol: Before accessing the resource, they try to create a file (with an agreed-upon name) using O_EXCL. Only one of them will succeed; the other processes' opens will fail. They can either wait and try later or just give up. When the successful pro-

cess finishes with the resource, it unlinks the file. One of the unsuccessful processes' `open`s will then work, and it can safely proceed.

For this to work, the checking to see if the file exists (with `access`, say, which is in Section 3.8.1) and the creating of it have to be atomic (indivisible)—no other process can be allowed to execute in the interim, or it might create that very file *after* the first process has already checked. So don't do this:

```
if (access( … ) == 0)        /* file does not exist */
    … open(… ) …             /* create it */
```

We need a more reliable method that guarantees atomicity.

A simple exclusivity mechanism like this is called a *mutex* (short for mutual exclusion), a *binary semaphore* (can count only up to 1), or a *lock*. We'll hit these things several times throughout this book; see Section 1.1.7 for a quick rundown. In the UNIX world, the word "mutex" is more often used in the context of threads and the word "semaphore" suggests something that uses the UNIX semaphore system calls, so we'll just call it a "lock" in this section.

The protocol is best encapsulated into two functions, `lock` and `unlock`, to be used like this:

```
if  (lock("accounts")) {
    … manipulate accounts …
    unlock ("accounts");
}
else
    … couldn't obtain lock …
```

The lock name "accounts" is abstract; it doesn't necessarily have anything to do with an actual file. If two or more processes are concurrently executing this code, the lock will prevent them from simultaneously executing the protected section ("manipulate accounts," whatever that means). Remember that if a process doesn't call `lock`, though, then there is no protection. They're *advisory* locks, not *mandatory*. (See Section 7.11.5 for more on the distinction between the two.)

Here is the code for `lock`, `unlock`, and a little function `lockpath`:

```
#define LOCKDIR "/tmp/"
#define MAXTRIES 10
#define NAPLENGTH 2

static char *lockpath(char *name)
{
    static char path[100];
```

```
    if (snprintf(path, sizeof(path), "%s%s", LOCKDIR, name) > sizeof(path))
        return NULL;
    return path;
}

bool lock(char *name)
{
    char *path;
    int fd, tries;

    ec_null( path = lockpath(name) )
    tries = 0;
    while ((fd = open(path, O_WRONLY | O_CREAT | O_EXCL, 0)) == -1 &&
      errno == EEXIST) {
        if (++tries >= MAXTRIES) {
            errno = EAGAIN;
            EC_FAIL
        }
        sleep(NAPLENGTH);
    }
    if (fd == -1)
        EC_FAIL
    ec_neg1( close(fd) )
    return(true);

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}

bool unlock(char *name)
{
    char *path;

    ec_null( path = lockpath(name) )
    ec_neg1( unlink(path) )
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

The function `lockpath` generates an actual file name to use as the lock. We put it in the directory `/tmp` because that directory is on every UNIX system and it's writable by everyone. Note that if `snprintf` returns a number too big, it doesn't overrun the buffer passed to it; the number represents what *might* have occurred.

`lock` places a lock by trying to create the file with `O_EXCL`, as explained in the previous section, and we distinguish the `EEXIST` case from the other error cases, just as we did there.

We try to create the file up to `MAXTRIES` times, sleeping for `NAPLENGTH` seconds between tries. (`sleep` is in Standard C, and also in Section 9.7.2.) We close (Section 2.11) the file descriptor we got from `open` because we aren't interested in actually writing anything to the file—its existence or nonexistence is all we care about. We even created it with no permissions. As an enhancement, we could write our process number and the time on the file so other processes waiting on it could see who they're waiting for and how long it's been busy. If we did this, we'd want the permissions to allow reading.

All `unlock` has to do is remove the file. The next attempt to create it will succeed. I'll explain the system call `unlink` in Section 2.6.

The little test program is also interesting:

```
void testlock(void)
{
    int i;

    for (i = 1; i <= 4; i++) {
        if (lock("accounts")) {
            printf("Process %ld got the lock\n", (long)getpid());
            sleep(rand() % 5 + 1); /* work on the accounts */
            ec_false( unlock("accounts") )
        }
        else {
            if (errno == EAGAIN) {
                printf("Process %ld tired of waiting\n", (long)getpid());
                ec_reinit(); /* forget this error */
            }
            else
                EC_FAIL /* something serious */
        }
        sleep(rand() % 5 + 5); /* work on something else */
    }
    return;

EC_CLEANUP_BGN
    EC_FLUSH("testlock")
EC_CLEANUP_END
}
```

It cycles four times through an acquire/work/release pattern, with the "work" just sleeping for some random number of seconds between 1 and 5. If it doesn't get the lock, it prints a complaint and keeps going. Then it does something else not involving the accounts for between 5 and 9 seconds, and cycles again. The `printf` calls use a system call to get the process ID that I didn't explain yet, `getpid` (Section 5.13). I ran three of these little guys at once:

```
$ tst & tst & tst &
```

and this was the output:

```
Process 9232 got the lock
Process 9233 got the lock
Process 9234 got the lock
Process 9232 got the lock
Process 9233 got the lock
Process 9232 got the lock
Process 9233 got the lock
Process 9234 got the lock
Process 9232 got the lock
Process 9233 got the lock
Process 9234 got the lock
Process 9234 got the lock
```

It started off in a predictable way but then got more interesting on line 6. At the end, process 9234 had to stay late after the others had gone home.

Let's talk about the pros and cons of using files as locks. The pros are that they're easy to code (we're still early in Chapter 2 of this book), being files they can contain some data, and they stay around as long as files do, which is useful when a long-duration lock is wanted. That last point also appears in the list of cons: If a process terminates without clearing its lock, even rebooting won't clear it unless the `/tmp` directory is cleared. Also, they're really slow because creating a file, even if it fails, is a giant operation—maybe OK for a few times per application-execution, but not for the kind of fast locking that a database or a real-time program might need.

I haven't mentioned the worst problem, however: If a process can't get the lock, it keeps sleeping and trying until it can, which is called *polling*. This is a lot of work for the CPU to do just to answer a simple question: "Is it my turn yet?" And, the lock might become available while the process is still asleep, but it won't find out until it wakes up, which is a waste.

Fortunately, all the built-in UNIX facilities for locking use what's called *blocking,* which means that the process sleeps until the event it's waiting for occurs. I'll get to that in Section 7.11.

2.4.4 Summary of `open` Flags

There are more `open` flags besides the ones I've explained so far, but it makes more sense to talk about them later when I've had a chance to place them in the appropriate context. For example, `O_NOCTTY` has to do with terminals, so I'll talk about it in Chapter 4. Table 2.1 shows all the flags defined by SUS3[5] with a quick description of each and a cross reference to the sections where they're discussed in detail.

Table 2.1 `open` Flags

Flag	Comment
`O_RDONLY`*	Open for reading only (Section 2.4.1).
`O_WRONLY`	Open for writing only (Section 2.4.1).
`O_RDWR`	Open for both (Section 2.4.1).
`O_APPEND`	All writes occur at the end of the file (Section 2.8).
`O_CREAT`	Create if nonexistent (Section 2.4.2).
`O_DSYNC`	Set synchronized I/O behavior (Section 2.16.3).
`O_EXCL`	Fail if exists; must be used with `O_CREAT` (Section 2.4.2).
`O_NOCTTY`	Don't make device the controlling terminal (Section 4.10.1).
`O_NONBLOCK`†	Don't wait for named pipe or special file to become available (Sections 4.2.2 and 7.2).
`O_RSYNC`	Set synchronized I/O behavior (Section 2.16.3).
`O_SYNC`	Set synchronized I/O behavior (Section 2.16.3).
`O_TRUNC`	Truncate to zero bytes (Section 2.4.2).

* One of the first three is required.
† Formerly called `O_NDELAY`, with somewhat different semantics.

5. Single UNIX Specification, Version 3; see Section 1.5.1.

2.5 `umask` System Call

We mentioned a process's file mode creation mask in Section 2.4.2. It's set by the `umask` system call, which is seldom used by anything other than the `umask` command:

umask—set and get file mode creation mask

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

mode_t umask(
    mode_t cmask           /* new mask */
);
/* Returns previous mask (no error return) */
```

Since every process has a mask, and since every combination of nine bits is legal, `umask` can never give an error return. It always returns the old mask. To find out what the old mask is without changing it requires two calls to `umask`: one to get the old value, with an argument of anything at all, and a second call to restore the mask to the way it was.

2.6 `unlink` System Call

unlink—remove directory entry

```
#include <unistd.h>

int unlink(
    const char *path       /* pathname */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

The `unlink` system call removes a link from a directory, reducing the link count in the i-node by one. If the resulting link count is zero, the file system will discard the file: All disk space that it used will be made available for reuse (added to the "free list"). The i-node will become available for reuse, too. The process must have write permission in the directory containing the link.

Any kind of file (regular, socket, named pipe, special, etc.) can be unlinked, but only a superuser can unlink a directory, and on some systems even the superuser can't. In any case, the `rmdir` system call (Section 3.6.3) should be used for unlinking a directory, not `unlink`.

If the link count goes to zero while some process still has the file open, the file system will delay discarding the file until it's closed, to avoid disrupting a running process. This feature is frequently used to make temporary files that are needed only while a program is running, like this:

```
ec_neg1( fd = open("temp", O_RDWR | O_CREAT | O_TRUNC | O_EXCL, 0) )
ec_neg1( unlink("temp") )
```

There are two advantages to this technique: First, if the process terminates for any reason, the file will be discarded. There's no need to register a function with `atexit` (Section 1.3.4), for example, to make sure the file is unlinked. Second, since the link is removed from the current directory right away with `unlink`, there's less danger of a second process accidentally using the same temporary file and failing in the `open` because of the `O_EXCL` flag. It could still happen, though, if the second process executes its `open` between the first process's `open` and `unlink`.

One way to fix the problem is with a lock (Section 2.4.3):

```
ec_false( lock("opentemp") )
ec_neg1(  fd = open("temp", O_RDWR | O_CREAT | O_TRUNC | O_EXCL, 0) )
ec_neg1(  unlink("temp") )
ec_false( unlock("opentemp") )
```

Our fix is pretty good, but still has a defect: The lock is only advisory, and the temporary-file name is rather unimaginative. Thus, it's possible for another application (not using the lock) to use the same name, and one of the processes would then fail to get its temporary file (failing in `open`, because of `O_EXCL`). A better fix is making the temporary file name unique, but that's a bit tricky, as we'll see in Section 2.7, where I talk more about temporary files.

You might think there is another problem: Since the file name is always `temp`, do two processes both running the code read and write the same temporary file, making a mess of things? No. If you made a file named `myfile` today, removed it, and made another file tomorrow with the same name, you wouldn't expect them to be the same file. That the first process still has the file open (with its data intact) is irrelevant because the i-node it's using is completely different from the i-node that the second process will use. Think of it this way: `unlink` makes the directory entry go away even if the file is still open, and the i-node then becomes anonymous and therefore completely isolated from access by a new process.

2.7 Creating Temporary Files

The approach we used to create a temporary file in the previous section, using a fixed name (`temp`) and a lock to prevent two processes from executing the same code at the same time, is cumbersome, and so it's more typical for a UNIX program to avoid any possible clash by using a name that's guaranteed to be unique. The Standard C function `tmpnam` seems to do what we want:

```
char *pathname;

ec_null( pathname = tmpnam(NULL) )
ec_neg1( fd = open(pathname, O_RDWR | O_CREAT | O_TRUNC | O_EXCL, 0) )
ec_neg1( unlink(pathname) )
```

It's guaranteed that what `tmpnam` returns is unique because it makes sure no file by that name exists. But the file doesn't get created until the `open`, so there's a small possibility that another process executing `tmpnam` at the same time could get the same name. (We have to keep reminding ourselves that the time between two lines of code can be arbitrarily long, and other processes can execute.) Because of the `O_EXCL` flag, one would fail to actually create and open it, so there's no danger of an I/O mix-up; however, we're only a little better off than we were with a fixed name. The probability of a clash is reduced, but still not zero. Not good enough.

Here's the answer:

mkstemp—create and open file with unique name

```
#include <stdlib.h>

int mkstemp(
    char *template        /* template for file name */
);
/* Returns open file descriptor or -1 on error (may not set errno) */
```

`mkstemp` absolutely guarantees that the file will be created with a unique name; there's no race-condition problem. You give it a template to use for the name that ends in six `X`s, which it replaces with whatever it takes to make the name unique. It then goes further than `tmpnam`: It actually creates and opens the file for reading and writing. You can't assume what permissions have been used, as the standard (SUS3) doesn't say, although most implementations will probably restrict reading and writing to just the owner (`S_IRUSR | S_IWUSR`).

For a portable program, it's not clear what to do about `errno` if `mkstemp` returns –1. The standard doesn't define any error codes, but it's a pretty good bet that all implementations will return a valid `errno`. So we choose to use the `ec_neg1` macro (which records `errno`, recall), even though the synopsis says `errno` may not be set. To try to prevent a misleading error message if `errno` is not set on an error, we try to remember to set it to zero prior to the call.

`mkstemp` is in SUS1, Linux, FreeBSD, and Darwin (it originated with BSD), so you can count on its being available just about everywhere.

Here's an example; just for fun we'll print the file name:

```
char pathname[] = "/tmp/dataXXXXXX";

errno = 0; /* mkstemp may not set it on error */
ec_neg1( fd = mkstemp(pathname) )
ec_neg1( unlink(pathname) )
printf("%s\n", pathname);
```

Output:

```
/tmp/dataKdBy0u
```

You don't have to unlink the file right away, but if you don't you'll have to arrange to unlink it later (in an `atexit`-registered function, say), or it will be left around. Of course, you can't unlink it immediately if you need to pass the pathname to another part of the program or an external program that needs a name, as in this example:

```
int status;
char cmd[100];

ec_neg1( fd = mkstemp(pathname) )
/* code to write text lines to fd (not shown) */
snprintf(cmd, sizeof(cmd), "sort %s", pathname);
ec_neg1( status = system(cmd) )
```

The comment ("code to write…") substitutes for code that writes the file. `system` is a Standard C function that invokes an external program; we'll see how it works in Chapter 5.

By the way, there's a Standard C function that you might want to look up named `tmpfile` that works just as well as `mkstemp`, but it returns a `FILE` pointer instead of a file descriptor.

There's one more function you may have heard of named `mktemp` that's closer to `tmpnam` than to `mkstemp`, in that it returns a name but doesn't create the file. It has all the problems of `tmpnam` and isn't in Standard C, so don't use it.

A quick summary:

- Good: `mkstemp` and `tmpfile`
- Bad: `mktemp` and `tmpnam`

2.8 File Offsets and `O_APPEND`

This section explains the `O_APPEND` flag, which we first saw in Section 2.4.4, and then introduces some properties of the `read`, `write`, `lseek`, `pread`, and `pwrite` system calls, which are explained more fully in the rest of this chapter.

A *file offset* is a position in a regular file that marks where the next `read` or `write` will occur. That's its only purpose. Other types of files—directories, sockets, named pipes, and symbolic links—don't have file offsets. Special files may or may not have them, depending on their implementation (Section 3.2).

Before UNIX came along (hint: the Beatles hadn't broken up yet), most operating systems had both "sequential" and "random" data files. UNIX had just one type of data file, with a movable file offset to handle random access. It was an important innovation in its day, even if it seems normal and obvious today.

You get an independent file offset each time you open a file, because, as Figure 2.1 showed, you get a new open file description. This means that in this example

```
int fd1, fd2, fd3;

ec_neg1( fd1 = open("myfile", O_WRONLY | O_CREAT | O_TRUNC, PERM_FILE) )
ec_neg1( fd2 = open("myfile", O_RDONLY) )
ec_neg1( fd3 = open("yourfile", O_RDWR | O_CREAT | O_TRUNC, PERM_FILE) )
```

`fd1` and `fd2` each have their own file offset so writes on `fd1` and reads on `fd2` are independent, but there is only one file offset for both reads and writes on `fd3`.

Absent the `O_APPEND` flag, the file offset starts out at zero on a freshly opened file and is automatically bumped by `read` and `write` by the amount they read or wrote. So, unless something is done to deliberately change the file offset, `read`s and `write`s are sequential. You read some, and then the next `read` reads some more, and so on. Ditto for `write`s.

Assuming the file offsets are starting at zero, if we write 100 bytes on `fd1` and then read 100 bytes on `fd2`, we get the 100 bytes we just wrote. But if we write on `fd3` and then read on `fd3`, we get whatever is *after* the written data, and an end-of-file if there wasn't anything because a file description has only a single offset, used by both `read` and `write`.

You can find out where the file offset is and/or set it to a new value with `lseek` (Section 2.13) on a file descriptor. The new value then affects the next `read` or `write` on that file descriptor.

Later, in Chapter 6, we're going to see how to duplicate an open file descriptor. By "duplicate" I don't mean copying, like

```
fd1 = fd2;
```

I mean using a system call that duplicates, such as `dup`. Anyway, for now the important point is that if a file descriptor is a duplicate of another, they share the same file offset because they share the same open file description.

If the file is opened with the `O_APPEND` flag set, all writes with the `write` system call are preceded by an implicit `lseek` to the end of the file, so writing occurs at the end, atomically. Even if several processes with the file opened with `O_APPEND` are writing at once, each of their writes will go at the end as it is that instant, and they won't overwrite each other or intermingle their data. You can't do the equivalent thing with `lseek` followed by a `write` (`O_APPEND` not set) because as we've seen in other situations, there's a gap between the two system calls that could result in this:

1. Process A seeks its file offset to the end (position 1000, say).
2. Process B seeks its file offset to the end (also position 1000).
3. Process B writes 200 bytes (at position 1000).
4. Process A writes 200 bytes (at position 1000—overwriting B). Ouch!

We could use a lock (Section 2.4.3) to fix this, but there's a much better way: If `O_APPEND` was set when process A and B opened the file, you are guaranteed to get this:

1. Process A seeks its file offset to the end (position 1000, say) and writes.
2. Process B seeks its file offset to the end (position 1200) and writes.

So O_APPEND is perfect for log files or other situations when you want to accumulate output from several processes.

You can also read and write by just specifying the position in the system call itself, without first calling lseek; that's what pread and pwrite do (Section 2.14). They don't use the file offset, and they don't change it, either.

Since you probably already know pretty much how read and write work, if you like you can skip to Section 2.13 to read about lseek and see some interesting examples and then loop back to Section 2.9 to continue in sequence.

2.9 write System Call

write—write to file descriptor

```
#include <unistd.h>

ssize_t write(
    int fd,             /* file descriptor */
    const void *buf,    /* data to write */
    size_t nbytes       /* amount to write */
);
/* Returns number of bytes written or -1 on error (sets errno) */
```

We've talked so much about write, don't you think it's time we got properly introduced?

write writes the nbytes bytes pointed to by buf to the open file represented by fd. The write starts at the current position of the file offset, and, after the write, the file offset is incremented by the number of bytes written. The number of bytes written, or –1 if there was an error, is returned.

Recall that if the O_APPEND flag is set, the file offset is set automatically to the end of the file prior to the write.

write is used to write to pipes, special files, and sockets, too, but its semantics are somewhat different in these cases. One important difference is that such writes can block, which means that they're waiting for some unpredictable event, such as data being available. If a write is blocked, it can be interrupted by the arrival of a signal (Section 9.1.4), in which case it returns –1 with errno set to EINTR. I'll postpone the rest of the discussion of writes to other than regular files until Chapters 4, 6, and 8, when I talk about other types of files.

`write` is deceptively simple. It seems that it writes the data and then returns, but a little experimentation will convince you that this is impossible—it's too fast. It must be cheating!

Indeed, it does cheat. When you issue a `write` system call, it does not perform the write and then return. It just transfers the data to a buffer cache in the kernel and then returns, claiming nothing more than this:

> I've taken note of your request, and rest assured that your file descriptor is OK. I've copied your data successfully, and there's enough disk space. Later, when it's convenient for me, and if I'm still alive, I'll try to put your data on the disk where it belongs. If I discover an error then I'll try to print something on the console, but I won 't tell you about it (indeed, you may have terminated by then). If you, or any other process, tries to read this data before I've written it out, I'll give it to you from the buffer cache, so, if all goes well, you'll never be able to find out when and if I've completed your request. You may ask no further questions. Trust me, and thank me for the speedy reply—I figured that's all you really cared about.

If all does go well, delayed writing is fantastic. The semantics are the same as if the writing actually took place, but it's much faster. However, if there is a disk error, or if the kernel stops for any reason, then the game is up. We discover that the data we "wrote" isn't on the disk at all.

In addition to the uncertainty about when the physical write occurs, there are two other problems with delayed writes. First, a process initiating a write cannot be informed of write errors. Indeed, file system buffers aren't owned by any single process; if several processes write to the same block of the same file at the same time, their data will be transferred to the same buffer. Of course, one could conceive of a scheme in which a "write error" signal would be sent to every process that wrote a particular buffer, but what is a process supposed to do about it at that late date? And how does the kernel notify processes that have already terminated?

The second problem is that the *order* of physical writes can't be controlled. Order often matters. For example, in updating a linked-list structure on a file, it is better to write a new record and then update the pointer to it, rather than the reverse. This is because a record not pointed to is usually less of a problem than a pointer that points nowhere. Even if the `write` system calls are issued in a particular order, however, that doesn't mean that the buffers will be physically written to disk in that order. So *careful replacement* techniques, of which this is but one example, are not as advantageous as they might be. They guard against the process itself terminating at an inopportune time, but not against disk errors or kernel crashes.

Fortunately, you can force synchronized writes, and I'll explain how in Section 2.16.

These problems with `write` should not be overemphasized. Considering how reliable computers are today, and how reliable UNIX implementations usually are, kernel crashes are quite rare. Most users are pleased to benefit from the quick response provided by the buffer cache and never find out that the kernel is cheating.

Now let's look once again at the file copy example at the beginning of this chapter. The bug is in the check for a write error:

```
if (write(tofd, buf, nread) != nread)
    EC_FAIL
```

It is *not* an error if the count returned by `write` is less than the requested count. Maybe the count is short because the write is to a pipe that's momentarily full, or maybe it's a regular file that's reached its size limit. It will be the *next* call to write that will produce the error.[6]

So the bug is that the `EC_FAIL` macro will record a meaningless value for `errno`, which is set *only* when the return value is –1. We could recode the function to withstand partial writes and to keep trying until a real error occurs:

```
#define BUFSIZE 512

void copy2(char *from, char *to)
{
    int fromfd = -1, tofd = -1;
    ssize_t nread, nwrite, n;
    char buf[BUFSIZE];

    ec_neg1( fromfd = open(from, O_RDONLY) )
    ec_neg1( tofd = open(to, O_WRONLY | O_CREAT | O_TRUNC,
      S_IRUSR | S_IWUSR) )
    while ((nread = read(fromfd, buf, sizeof(buf))) > 0) {
        nwrite = 0;
        do {
            ec_neg1( n = write(tofd, &buf[nwrite], nread - nwrite) )
            nwrite += n;
        } while (nwrite < nread);
    }
```

6. Although, even then, it's possible that whatever caused the short count, such as being out of space, resolved itself before the next call, so it won't return an error. That is, there's no 100% reliable way of finding out why a partial `write` or `read` was short.

```
    if (nread == -1)
        EC_FAIL
    ec_neg1( close(fromfd) )
    ec_neg1( close(tofd) )
    return;

EC_CLEANUP_BGN
    (void)close(fromfd); /* can't use ec_neg1 here! */
    (void)close(tofd);
EC_CLEANUP_END
}
```

Realistically, this seems like too much trouble for regular files, although we will use a technique similar to this later on, when we do I/O to terminals and pipes, which sometimes simply require additional attempts. Perhaps this simple solution makes more sense when we know we're writing to a regular file:

```
if ((nwrite = write(tofd, buf, nread)) != nread) {
    if (nwrite != -1)
        errno = 0;
    EC_FAIL
}
```

or you might prefer to code it like this:

```
errno = 0;
ec_false( write(tofd, buf, nread) == nread )
```

We set `errno` to zero so that the error report will show an error (along with the line number), but not what could be a misleading error code. The one thing you definitely don't want to do is ignore the short count completely:

```
ec_neg1( write(tofd, buf, nread) ) /* wrong */
```

Here's a handy function, `writeall`, that encapsulates the "keep trying" approach in the `copy2` example. We'll use it later in this book (Sections 4.10.2 and 8.5) when we want to make sure everything got written. Note that it doesn't use the "ec" macros because it's meant as a direct replacement for `write`:

```
ssize_t writeall(int fd, const void *buf, size_t nbyte)
{
    ssize_t nwritten = 0, n;

    do {
        if ((n = write(fd, &((const char *)buf)[nwritten],
          nbyte - nwritten)) == -1) {
            if (errno == EINTR)
                continue;
```

```
            else
                return -1;
        }
        nwritten += n;
    } while (nwritten < nbyte);
    return nwritten;
}
```

We've treated an `EINTR` error specially, because it's not really an error. It just means that `write` was interrupted by a signal before it got to write anything, so we keep going. Signals and how to deal with interrupted system calls are dealt with more thoroughly in Section 9.1.4. There's an analogous `readall` in the next section.

2.10 `read` System Call

read—read from file descriptor

```
#include <unistd.h>

ssize_t read(
    int fd,             /* file descriptor */
    void *buf,          /* address to receive data */
    size_t nbytes       /* amount to read */
);
/* Returns number of bytes read or -1 on error (sets errno) */
```

The `read` system call is the opposite of `write`. It reads the `nbytes` bytes pointed to by `buf` from the open file represented by `fd`. The `read` starts at the current position of the file offset, and then the file offset is incremented by the number of bytes read. `read` returns the number of bytes read, or 0 on an end-of-file, or –1 on an error. It isn't affected by the `O_APPEND` flag.

Unlike `write`, the `read` system call can't very well cheat by passing along the data and then reading it later. If the data isn't already in the buffer cache (due to previous I/O), the process just has to wait for the kernel to get it from disk. Sometimes, the kernel tries to speed things up by noticing access patterns suggestive of sequential reading of consecutive disk blocks and then reading ahead to anticipate the process's needs. If the system is lightly loaded enough for data to remain in buffers a while, and if reads are sequential, read-ahead is quite effective.

There's the same problem in getting information about partial reads as there was with partial writes: Since a short count isn't an error, `errno` isn't valid, and you

have to guess what the problem is. If you really need to read the whole amount, it's best to call `read` in a loop, which is what `readall` does (compare it to `writeall` in the previous section):

```
ssize_t readall(int fd, void *buf, size_t nbyte)
{
    ssize_t nread = 0, n;

    do {
        if ((n = read(fd, &((char *)buf)[nread], nbyte - nread)) == -1) {
            if (errno == EINTR)
                continue;
            else
                return -1;
        }
        if (n == 0)
            return nread;
        nread += n;
    } while (nread < nbyte);
    return nread;
}
```

We'll see `readall` in use in Section 8.5.

As with `write`, a `read` from a pipe, special file, or socket can block, in which case it may be interrupted by a signal (Section 9.1.4), causing it to return –1 with `errno` set to `EINTR`.

2.11 `close` System Call

close—close file descriptor

```
#include <unistd.h>

int close(
    int fd              /* file descriptor */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

The most important thing to know about the `close` system call is that it does practically nothing. It does *not* flush any kernel buffers; it just makes the file descriptor available for reuse. When the last file descriptor pointing to an open file description (Section 2.2.3) is closed, the open file description can be deleted as well. And, in turn, when the last open file description pointing to an in-memory i-node is deleted, the in-memory i-node can be deleted. There's one more step, too: If all

the links to the actual i-node have been removed, the i-node on disk and all its data are deleted (explained in Section 2.6).

Since `close` doesn't flush the buffer cache, or even accelerate flushing, there's no need to close a file that you've written before reading it. It's guaranteed that you'll get the data you wrote. To say it another way, the kernel buffering in no way affects the semantics of `read`, `write`, `lseek`, or any other system call.

In fact, if the file descriptor isn't needed again, there's no requirement to call `close` at all, as the file descriptors will be reclaimed when the process terminates. It's still a good idea, however, to free-up kernel structures and to indicate to readers of your program that you're finished with the file. If you're consistently checking for errors, it also keeps you from accidentally using the file descriptor later on by mistake.

There are some additional side-effects of calling `close` that apply to pipes and other irregular files, but I'll talk about them when we get into the details of those types of files.

2.12 User Buffered I/O

2.12.1 User vs. Kernel Buffering

Recall from Chapter 1 that the UNIX file system is built on top of a block special file, and therefore all kernel I/O operations and all kernel buffering are in units of the block size. Anything is possible, but in practice it's always a multiple of 512, with numbers like 1024, 2048, and 4096 fairly typical. It's not necessarily the same on all devices, and it can even vary depending on how the disk partition was created. We'll get to figuring out what it is shortly.

Reads and writes in chunks equal to the block size that occur on a block-sized boundary are faster than any smaller unit. To demonstrate this we recompiled `copy2` from Section 2.9 with a user buffer size of 1 by making this change:

```
#define BUFSIZE 1
```

We timed the two versions of copy on a 4MB file[7] with the results shown in Table 2.2 (times are in seconds).

Table 2.2 Block-Sized I/O vs. Character-at-a-Time

Method	User	System	Total
512-byte buffer	0.07	0.58	0.65
1-byte buffer	18.43	204.98	223.41

(The times shown were on Linux; on FreeBSD they were about the same.)

User time is the time spent executing instructions in the user process. System time is the time spent executing instructions in the kernel on behalf of the process.

The performance penalty for I/O with regular files in such small chunks is so drastic that one simply never does it, unless the program is just for occasional and casual use, or the situation is quite unusual (reading a file backward, for example; see Section 2.13). Most of the penalty is simply from the larger number of system calls (by a factor of 512). To test the penalty from a poor choice of I/O buffer size, the tests were rerun with `BUFSIZE` set to 1024, a "good" size, and then 1100, a larger, but "bad" size. This time the difference was less on Linux, but still about 75% worse for 1100 (7.4 sec. total time vs. 4.25 sec.). On FreeBSD and Solaris the differences were much closer, only 10–20% off.

But the problem is that rarely is the block size a natural fit for what the program really wants to do. Lines of varying lengths and assorted structures are more typical. The solution is to pack the odd pieces of data into blocks in user space and to write a block only when it's full. On input, one does the reverse: unpacking the data from blocks as they're read. That is, one does *user* buffering in addition to *kernel* buffering. Since a piece of data can span a block boundary, it's somewhat tricky to program, and I'll show how to do it in the next section.

First, one nagging question: How do you know what the block size is? Well, you could use the actual number from the file system that's going to hold the files

7. The 1985 edition of this book used a *4000-byte* file, with similar times!

you're working with,[8] but some experimentation has shown that, within reason, larger numbers are better than smaller ones, and here's why: If the number is less than the actual block size, but divides evenly into it, the kernel can very efficiently pack the writes into a buffer and then schedule the buffer to be written when it's full. If the number is bigger and a multiple of the buffer size, it's still very efficient for the kernel to fit the data into buffers *and* there are fewer `write` system calls, which tends to be the overwhelming factor. A similar argument holds for `read`s.

So, by far the easiest thing to do is to use the macro `BUFSIZ`, defined by Standard C, which is what the standard I/O functions (e.g., `fputs`) use. It's not optimized for the actual file systems, being constant, but the experiments showed that that doesn't matter so much. If space is at a premium, you can even use 512 and you'll be fine.

2.12.2 Functions for User Buffering

It's convenient to use a set of functions that do reads, writes, seeks, and so on, in whatever units the caller wishes. These subroutines handle the buffering automatically and never stray from the block model. An exceptionally fine example of such a package is the so-called "standard I/O library," described in most books on C, such as [Har2002].

To show the principles behind a user-buffering package, I'll present a simplified one here that I call `BUFIO`. It supports reads and writes, but not seeks, in units of a single character. First, the header file bufio.h that users of the package must include (prototypes not shown):

```
typedef struct {
    int fd;                     /* file descriptor */
    char dir;                   /* direction: r or w */
    ssize_t total;              /* total chars in buf */
    ssize_t next;               /* next char in buf */
    unsigned char buf[BUFSIZ];  /* buffer */
} BUFIO;
```

Now for the implementation of the package (bufio.c):

8. You use the `stat` system call (Section 3.5.1).

```
BUFIO *Bopen(const char *path, const char *dir)
{
    BUFIO *b = NULL;
    int flags;

    switch (dir[0]) {
    case 'r':
        flags = O_RDONLY;
        break;
    case 'w':
        flags = O_WRONLY | O_CREAT | O_TRUNC;
        break;
    default:
        errno = EINVAL;
        EC_FAIL
    }
    ec_null( b = calloc(1, sizeof(BUFIO)) )
    ec_neg1( b->fd = open(path, flags, PERM_FILE) )
    b->dir = dir[0];
    return b;

EC_CLEANUP_BGN
    free(b);
    return NULL;
EC_CLEANUP_END
}

static bool readbuf(BUFIO *b)
{
    ec_neg1( b->total = read(b->fd, b->buf, sizeof(b->buf)) )
    if (b->total == 0) {
        errno = 0;
        return false;
    }
    b->next = 0;
    return true ;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}

static bool writebuf(BUFIO *b)
{
    ssize_t n, total;

    total = 0;
    while (total < b->next) {
        ec_neg1( n = write(b->fd, &b->buf[total], b->next - total) )
        total += n;
    }
```

```
    b->next = 0;
    return true ;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}

int Bgetc(BUFIO *b)
{
    if (b->next >= b->total)
        if (!readbuf(b)) {
            if (errno == 0)
                return -1;
            EC_FAIL
        }
    return b->buf[b->next++];

EC_CLEANUP_BGN
    return -1;
EC_CLEANUP_END
}

bool Bputc(BUFIO *b, int c)
{
    b->buf[b->next++] = c;
    if (b->next >= sizeof(b->buf))
        ec_false( writebuf(b) )
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}

bool Bclose(BUFIO *b)
{
    if (b != NULL) {
        if (b->dir == 'w')
            ec_false( writebuf(b) )
        ec_neg1( close(b->fd) )
        free(b);
    }
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

Finally, we recode our file-copy function to use the new package:

```
#include "bufio.h"

bool copy3(char *from, char *to)
{
    BUFIO *stfrom, *stto;
    int c;

    ec_null( stfrom = Bopen(from, "r") )
    ec_null( stto = Bopen(to, "w") )
    while ((c = Bgetc(stfrom)) != -1)
        ec_false( Bputc(stto, c) )
    if (errno != 0)
        EC_FAIL
    ec_false( Bclose(stfrom) )
    ec_false( Bclose(stto) )
    return true;

EC_CLEANUP_BGN
    (void)Bclose(stfrom);
    (void)Bclose(stto);
    return false;
EC_CLEANUP_END
}
```

You will notice a strong resemblance between BUFIO and a subset of the standard I/O library. Here's a version of copy using library functions:

```
bool copy4(char *from, char *to)
{
    FILE *stfrom, *stto;
    int c;

    ec_null( stfrom = fopen(from, "r") )
    ec_null( stto = fopen(to, "w") )
    while ((c = getc(stfrom)) != EOF)
        ec_eof( putc(c, stto) )
    ec_false( !ferror(stfrom) )
    ec_eof( fclose(stfrom) )
    ec_eof( fclose(stto) )
    return true;

EC_CLEANUP_BGN
    (void)fclose(stfrom);
    (void)fclose(stto);
    return false;
EC_CLEANUP_END
}
```

To see the great benefits of user buffering, Table 2.3 shows the times in seconds for the file copy using BUFIO, the standard I/O Library, and the straight-system-call method (shown in `copy2`).

Table 2.3 Comparison of Buffered and Unbuffered I/O

Method	User *	System	Total
BUFIO			
Solaris	1.00	0.51	1.51
Linux	1.00	0.28	1.28
FreeBSD	1.00	0.45	1.45
Standard I/O			
Solaris	0.57	0.24	0.81
Linux	11.32	0.15	11.48
FreeBSD	1.02	0.20	1.22
BUFSIZ buffer			
Solaris	0.00	0.52	0.52
Linux	0.00	0.23	0.23
FreeBSD	0.01	0.37	0.38

* All times in seconds and normalized within systems so that user BUFIO time on that system is 1.00.

With user buffering we've almost got the best of both worlds: We process the data as we like, even 1 byte at a time, yet we achieve a system time about the same as the `BUFSIZ` buffer method. So user buffering is definitely the right approach. Except for Linux, the standard I/O times are the best; it does what our `BUFIO` functions do, except faster and with more flexibility. The standard I/O user time for Linux (11.32) sticks out. A bit of research showed that while we used the gcc

compiler for all the tests, Linux uses the gcc version of stdio.h, FreeBSD uses one based on BSD, and Solaris uses one based on System V. Looks like the gcc version needs some attention.[9]

The wide acceptance of the standard I/O library is ironic. The ability to do I/O in arbitrary units on regular files has always been one of the really notable features of the UNIX kernel, yet in practice, the feature is usually too inefficient to use.

2.13 `lseek` System Call

The `lseek`[10] system call just sets the file offset for use by the next `read`, `write`, or `lseek`. No actual I/O is performed, and no commands are sent to the disk controller (remember, there's normally a buffer cache in the way anyhow).

lseek—set and get file offset

```
#include <unistd.h>

off_t lseek(
    int fd,             /* file descriptor */
    off_t pos,          /* position */
    int whence          /* interpretation */
);
/* Returns new file offset or -1 on error (sets errno) */
```

The argument `whence` can be one of:

`SEEK_SET` The file offset is set to the `pos` argument.

`SEEK_CUR` The file offset is set to its current value plus the `pos` argument, which could be positive, zero, or negative. Zero is a way to find out the current file offset.

`SEEK_END` The file offset is set to the size of the file plus the `pos` argument which could be positive, zero, or negative. Zero is a way to set the file offset to the end of the file.

9. The problem is that it's checking for thread locks on each `putc`, but the other systems were smart enough to realize that we weren't multithreading. This may be fixed by the time you read this.

10. It was called "seek" back in the days before C had a `long` data type (this goes *way* back), and to get to a byte past 65,535 required a seek to the block, and then a second seek to the byte in the block. The extra letter was available for the new system call, as `creat` was one letter short.

The resulting file offset may have any non-negative value at all, even greater than the size of the file. If greater, the next `write` stretches the file to the necessary length, effectively filling the interval with bytes of zero. A `read` with the file offset set at or past the end generates a zero (end-of-file) return. A `read` of the stretched interval caused by a `write` past the end succeeds, and returns bytes of zero, as you would expect.

When a `write` beyond the end of a file occurs, most UNIX systems don't actually store the intervening blocks of zeros. Thus, it is possible for a disk with, say, 3,000,000 available blocks to contain files whose combined lengths are greater than 3,000,000 blocks. This can create a serious problem if files are backed up file-by-file and then restored; as the files have to be read to transfer them to the backup device, more than 3,000,000 blocks will be written and then read back in! Users who create many files with holes in them usually hear about it from their system administrator, unless the backup program is smart enough to recognize the holes.

Of all the possible ways to use `lseek`, three are the most popular. First, `lseek` may be used to seek to an absolute position in the file:

```
ec_neg1( lseek(fd, offset, SEEK_SET) )
```

Second, `lseek` may be used to seek to the end of the file:

```
ec_neg1( lseek(fd, 0, SEEK_END) )
```

Third, `lseek` may be used to find out where the file offset currently is:

```
off_t where;
ec_neg1( where = lseek(fd, 0, SEEK_CUR) )
```

Other ways of using `lseek` are much less common.

As I said in Section 2.8, most seeks done by the kernel are implicit rather than as a result of explicit calls to `lseek`. When `open` is called, the kernel seeks to the first byte. When `read` or `write` is called, the kernel increments the file offset by the number of bytes read or written. When a file is opened with the `O_APPEND` flag set, a seek to the end of the file precedes each write.

To illustrate the use of `lseek`, here is a function `backward` that prints a file backward, a line at a time. For example, if the file contains:

```
dog
bites
man
```

then `backward` will print:

```
man
bites
dog
```

Here is the code:

```
void backward(char *path)
{
    char s[256], c;
    int i, fd;
    off_t where;

    ec_neg1( fd = open(path, O_RDONLY) )
    ec_neg1( where = lseek(fd, 1, SEEK_END) )
    i = sizeof(s) - 1;
    s[i] = '\0';
    do {
        ec_neg1( where = lseek(fd, -2, SEEK_CUR) )
        switch (read(fd, &c, 1)) {
        case 1:
            if (c == '\n') {
                printf("%s", &s[i]);
                i = sizeof(s) - 1;
            }
            if (i <= 0) {
                errno = E2BIG;
                EC_FAIL
            }
            s[--i] = c;
            break;
        case -1:
            EC_FAIL
            break;
        default: /* impossible */
            errno = 0;
            EC_FAIL
        }
    } while (where > 0);
    printf("%s", &s[i]);
    ec_neg1( close(fd) )
    return;

EC_CLEANUP_BGN
    EC_FLUSH("backward");
EC_CLEANUP_END
}
```

There are two tricky things to observe in this function: First, since `read` implicitly seeks the file forward, we have to seek backward by *two* bytes to read the previous byte; to get things started we set the file offset to one byte past the end. Second, there's no such thing as a "beginning-of file" return from `read`, as there is an end-of-file return, so we have to watch the file offset (the variable `where`) and stop after we've read the first byte. Alternatively, we could wait until `lseek` tries to make the file pointer negative; however, this is unwise because the error code in this case (`EINVAL`) is also used to indicate other kinds of invalid arguments, and because as a general rule it's just better not to use error returns to implement an algorithm.

2.14 `pread` and `pwrite` System Calls

pread—read from file descriptor at offset

```
#include <unistd.h>

ssize_t pread(
    int fd,             /* file descriptor */
    void *buf,          /* address to receive data */
    size_t nbytes,      /* amount to read */
    off_t offset        /* where to read */
);
/* Returns number of bytes read or -1 on error (sets errno) */
```

pwrite—write to file descriptor at offset

```
#include <unistd.h>

ssize_t pwrite(
    int fd,             /* file descriptor */
    const void *buf,    /* data to write */
    size_t nbytes,      /* amount to write */
    off_t offset        /* where to write */
);
/* Returns number of bytes written or -1 on error (sets errno) */
```

`pread` and `pwrite` are almost exactly like `read` and `write` preceded with a call to `lseek`, except:

- The file offset isn't used, as the position for reading and writing is supplied explicitly by the `offset` argument.
- The file offset isn't set, either. In fact, it's completely ignored.

The `O_APPEND` flag (Section 2.8) does affect `pwrite`, making it behave exactly like `write` (as I said, the `offset` argument is ignored).

One call instead of two is convenient, but more importantly, `pread` and `pwrite` avoid the problem of the file offset being changed by another process or thread between an `lseek` and the following `read` or `write`. Recall that this could happen, as threads could use the same file descriptor, and processes could have duplicates that share a file offset, as explained in Section 2.2.3. This is the same problem that `O_APPEND` avoids (Section 2.8). As neither `pread` nor `pwrite` even use the file offset, they can't get the position wrong.

To show `pread` in action, here's another version of `backward` (from the previous section). It's more straightforward—easier to write and debug—because the funny business with calling `lseek` to decrement the file offset by –2 is gone:

```
void backward2(char *path)
{
    char s[256], c;
    int i, fd;
    off_t file_size, where;

    ec_neg1( fd = open(path, O_RDONLY) )
    ec_neg1( file_size = lseek(fd, 0, SEEK_END) )
    i = sizeof(s) - 1;
    s[i] = '\0';
    for (where = file_size - 1; where >= 0; where--)
        switch (pread(fd, &c, 1, where)) {
        case 1:
            if (c == '\n') {
                printf("%s", &s[i]);
                i = sizeof(s) - 1;
            }
            if (i <= 0) {
                errno = E2BIG;
                EC_FAIL
            }
            s[--i] = c;
            break;
        case -1:
            EC_FAIL
            break;
        default: /* impossible */
            errno = 0;
            EC_FAIL
        }
    printf("%s", &s[i]);
    ec_neg1( close(fd) )
    return;
```

```
EC_CLEANUP_BGN
    EC_FLUSH("backward2");
EC_CLEANUP_END
}
```

2.15 `readv` and `writev` System Calls

readv—scatter read

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

ssize_t readv(
     int fd,                    /* file descriptor */
     const struct iovec *iov,   /* vector of data buffers */
     int iovcnt                 /* number of elements */
);
/* Returns number of bytes read or -1 on error (sets errno) */
```

writev—gather write

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

ssize_t writev(
     int fd,                    /* file descriptor */
     const struct iovec *iov,   /* vector of data buffers */
     int iovcnt                 /* number of elements */
);
/* Returns number of bytes written or -1 on error (sets errno) */
```

`readv` and `writev` are like `read` and `write`, except that instead of using one data address, they can take data from or put data to several memory addresses at once. They're sometimes called *scatter* read and *gather* write. The data is still contiguous on the file, pipe, socket, or whatever `fd` is open to—it's the process's memory that can be scattered.

In other words, if you have three structures to be written, instead of using three `write`s to do it, you can write all three at once with `writev`; however, looking at the weird second argument, you can readily see that these functions are a pain to use, so they better be worth it, right? I'll get to that question shortly; I'll show how to use them first.

Before you call either function you have to set up the `iov` array (of `iovcnt` elements) so that each element contains a pointer to data and a size—in effect, the second and third arguments to an equivalent call of `read` or `write`. Think of `struct iovec` as being defined like this (it may have additional, nonstandard, members):

struct iovec—structure for readv and writev[11]

```
struct iovec {
    void *iov_base;     /* base address of data */
    size_t iov_len;     /* size of this piece */
};
```

In your program, you can declare an array of `struct iovec`s, or allocate memory with `malloc`, or whatever you like, as long as there's enough of it and it's properly initialized. The maximum number of elements you can have varies with the system, but it's always at least 16. On SUS systems the symbol `IOV_MAX`, if defined, tells you the actual maximum; if it's undefined you can call `sysconf` (Section 1.5.5) with an argument of `_SC_IOV_MAX`. But, really, for what `readv` and `writev` were designed for, 16 is a lot.

Here are some fragments from an example program that uses `writev` to write a header structure followed by two data structures. First, the structure declarations:

```
#define VERSION 506
#define STR_MAX 100

struct header {
    int h_version;
    int h_num_items;
} hdr, *hp;
struct data {
    enum {TYPE_STRING, TYPE_FLOAT} d_type;
    union {
        float d_val;
        char d_str[STR_MAX];
    } d_data;
} d1, d2, *dp;
struct iovec v[3];
```

(The version is just some number to identify this header type.)

Next, we initialize the header, two data structures, and the vector:

```
hdr.h_version = VERSION;
hdr.h_num_items = 2;
d1.d_type = TYPE_STRING;
strcpy(d1.d_data.d_str, "Some data to write");
d2.d_type = TYPE_FLOAT;
d2.d_data.d_val = 123.456;
```

11. And also `sendmsg` and `recvmsg`; see Section 8.6.3.

```
v[0].iov_base = (char *)&hdr; /* iov_base is sometimes char * */
v[0].iov_len = sizeof(hdr);
v[1].iov_base = (char *)&d1;
v[1].iov_len = sizeof(d1);
v[2].iov_base = (char *)&d2;
v[2].iov_len = sizeof(d2);
```

and then write all three structures with a single call to `writev`:

```
ec_neg1( n = writev(fd, v, sizeof(v) / sizeof(v[0])) )
```

Note, in the initialization of `iov_base`, the cast, which we ordinarily wouldn't need, as the SUS declares it as a `void` pointer. But FreeBSD (and probably other systems) define it a `char` pointer. The cast suits both.

To show that the data really is contiguous on the file, we read it back and print it out. We don't read the data into the original structures, but into one big anonymous buffer that we point into with pointers of the appropriate types (`hp` and `dp`). Note the call to `lseek` to rewind the file, which was opened `O_RDWR` (not shown):

```
ec_null( buf = malloc(n) )
ec_neg1( lseek(fd, 0, SEEK_SET) )
ec_neg1( read(fd, buf, n) )
hp = buf;
dp = (struct data *)(hp + 1);
printf("Version = %d\n", hp->h_version);
for (i = 0; i < hp->h_num_items; i++) {
    printf("#%d: ", i);
    switch (dp[i].d_type) {
    case TYPE_STRING:
        printf("%s\n", dp[i].d_data.d_str);
        break;
    case TYPE_FLOAT:
        printf("%.3f\n", dp[i].d_data.d_val);
        break;
    default:
        errno = 0;
        EC_FAIL
    }
}
ec_neg1( close(fd) )
free(buf);
```

This is the output:

```
Version = 506
#0: Some data to write
#1: 123.456
```

Aside from the obvious—being able to do in one system call what would otherwise take several—how much do `readv` and `writev` buy you? Table 2.4 shows some timing tests based on using `writev` to write 16 data items of 200 bytes each 50,000 times vs. substituting 16 calls to `write` for each call to `writev`. Thus, each test wrote a 160MB file. As the different UNIX systems were on different hardware, we've normalized the times to show the `writev` system time as 50 sec., which happens to be about what it was on the Linux machine.[12]

Table 2.4 Speed of writev vs. write

System Call	User	System
Solaris		
`writev`	1.35	50.00
`write`	8.45	67.61
Linux		
`writev`	.17	50.00
`write`	1.83	27.67
FreeBSD		
`writev`	.39	50.00
`write`	4.90	209.89

For Solaris and, especially, FreeBSD, it looks like `writev` really does win over `write`. On Linux the system time is actually *worse*. Reading through the Linux code to see why, it turns out that for files, Linux just loops through the vector, calling `write` for each element! For sockets, which are what `readv` and `writev` were designed for, Linux does much better, carrying the vector all the way down to some very low-level code. We don't have timing tests to report, but it's apparent from the code that on sockets `writev` is indeed worthwhile.

12. As you read the table, remember that you can't conclude anything about the relative speeds of, say, Linux vs. FreeBSD, because each was separately normalized. We're only interested in the degree to which each system provides an advantage of `writev` over `write`. Also, Linux may not be as bad as my results show; see www.basepath.com/aup/writer.htm.

2.16 Synchronized I/O

This section explains how to bypass kernel buffering, which was introduced in Section 2.12.1, and, if you don't want to go that far, how to control when buffers are flushed to disk.

2.16.1 Synchronized vs. Synchronous

In English the words "synchronized" and "synchronous" have nearly the same meaning, but in UNIX I/O they mean different things:

- *Synchronized* I/O means that a call to `write` (or its siblings, `pwrite` and `writev`) doesn't return until the data is flushed to the output device (disk, typically). Normally, as I indicated in Section 2.9, `write`, unsynchronized, returns leaving the data in the kernel buffer cache. If the computer crashes in the interim, the data will be lost.
- *Synchronous* I/O means that `read` (and its siblings) doesn't return until the data is available, and `write` (and its siblings) doesn't return until the data has been at least written to the kernel buffer, and all the way to the device if the I/O is also synchronized. The I/O calls we've described so far—`read`, `pread`, `readv`, `write`, `pwrite`, and `writev`—all operate synchronously.

Normally, therefore, UNIX I/O is *unsynchronized* and *synchronous*. Actually, reads and writes are asynchronous to some extent because of the way the buffer cache works; however, as soon as you make writes synchronized (forcing the buffers out on every call), actually waiting for a write to return would slow down the program too much, and that's when you really want writes to operate asynchronously. You want to say, "initiate this write and I'll go off and do something useful while it's writing and ask about what happened later when I feel like it." Reads, especially nonsequential ones, are always somewhat synchronized (the kernel can't fake it if the data's not in the cache); you'd rather not wait for them, either.

Getting `read` and `write` to be synchronized involves setting some `open` flags or executing system calls to flush kernel buffers, and it's the subject of this section. Asynchronous I/O uses a completely different group of system calls (e.g., `aio_write`) that I'll talk about in Section 3.9, where I'll distinguish between synchronized and synchronous even more sharply.

2.16.2 Buffer-Flushing System Calls

The oldest, best known, and least effective I/O-synchronizing call is `sync`:

sync—schedule buffer-cache flushing

```
#include <unistd.h>

void sync(void);
```

All `sync` does is tell the kernel to flush the buffer cache, which the kernel immediately adds to its list of things to do. But `sync` returns right away, so the flushing happens sometime later. You're still not sure when the buffers got flushed. `sync` is also heavy-handed—*all* the buffers that have been written are flushed, not just those associated with the files you care about.

The main use of this system call is to implement the `sync` command, run when UNIX is being shut down or before a removable device is unmounted. There are better choices for applications.

The next call, `fsync`, behaves, at a minimum, like `sync`, but just for those buffers written on behalf of a particular file. It's supported on SUS2 systems and on earlier systems if the option symbol `_POSIX_FSYNC` is defined.

fsync—schedule or force buffer-cache flushing for one file

```
#include <unistd.h>

int fsync(
    int fd              /* file descriptor */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

We said "at a minimum" because if the synchronized I/O option (`_POSIX_SYNCHRONIZED_IO`) is supported (Section 1.5.4), the guarantee is much stronger: It doesn't return until the buffers have been physically written to the device controller, or until an error has been detected.[13]

Here's the function `option_sync_io` to check if the option is supported; see Section 1.5.4 for an explanation of what it's doing.

13. The data still might be only in the controller's cache, but it will get to the storage medium very rapidly from there, usually even if UNIX crashes or locks up, as long as the hardware is still powered and functional.

```
OPT_RETURN option_sync_io(const char *path)
{
#if _POSIX_SYNCHRONIZED_IO <= 0
    return OPT_NO;
#elif _XOPEN_VERSION >= 500 && !defined(LINUX)
    #if !defined(_POSIX_SYNC_IO)
        errno = 0;
        if (pathconf(path, _PC_SYNC_IO) == -1)
            if (errno == 0)
                return OPT_NO;
            else
                EC_FAIL
        else
            return OPT_YES;

    EC_CLEANUP_BGN
        return OPT_ERROR;
    EC_CLEANUP_END
    #elif _POSIX_SYNC_IO == -1
        return OPT_NO;
    #else
        return OPT_YES;
    #endif /* _POSIX_SYNC_IO */
#elif _POSIX_VERSION >= 199309L
    return OPT_YES;
#else
    errno = EINVAL;
    return OPT_ERROR;
#endif /* _POSIX_SYNCHRONIZED_IO */
}
```

The last syncing call, `fdatasync`, is a slightly faster form of `fsync` because it forces out only the actual data, not control information such as the file's modification time. For most critical applications, this is enough; the normal buffer-cache writing will take care of the control information later. `fdatasync` is only available as part of the synchronized I/O option (i.e., there's no `sync`-like behavior).

fdatasync—force buffer-cache flushing for one file's data

```
#include <unistd.h>

int fdatasync(
    int fd              /* file descriptor */
);
/* Returns 0 or -1 on error (sets errno) */
```

With the synchronized I/O option, both `fsync` and `fdatasync` can return genuine I/O errors, for which `errno` will be set to `EIO`.

One last point on these functions: If synchronized I/O is not supported, the implementation isn't required to do anything when you call `sync` or `fsync` (`fdatasync` won't be present); they might be no-ops. Think of them as mere requests. If the option is supported, however, the implementation is required to provide a high level of data integrity, although what actually happens depends on the device driver and the device.

2.16.3 `open` Flags for Synchronization

Typically you call `fsync` or `fdatasync` when it's necessary for the application to know that the data has been written, such as before reporting to the user that a database transaction has been committed. For even more critical applications, however, you can arrange, via `open` flags, for an implicit `fsync` or `fdatasync` on *every* `write`, `pwrite`, and `writev`.

I first mentioned these flags, `O_SYNC`, `O_DSYNC`, and `O_RSYNC` in the table in Section 2.4.4. They're only available if synchronized I/O is supported. Here's what they do:

`O_SYNC` causes an implicit `fsync` (full-strength variety) after every `write`.

`O_DSYNC` causes an implicit `fdatasync` after every `write`.

`O_RSYNC` causes `read`, as well as `write`, synchronization; must be used with `O_SYNC` or `O_DSYNC`.

(When we say `write`, we also mean `pwrite` and `writev`, and similarly for `read`.)

The only thing that `O_RSYNC` really does is ensure that the access time in the i-node is updated in a synchronized manner, which means that with `O_DSYNC` it probably doesn't do anything at all. Even with `O_SYNC`, the access time is rarely critical enough to require synchronized updating. It's also possible on some systems for `O_RSYNC` to disable read-ahead.

Here's an example program using `O_DSYNC` that compares synchronized and unsynchronized writing:

```
#define SYNCREPS 5000
#define PATHNAME "tmp"
```

```
void synctest(void)
{
    int i, fd = -1;
    char buf[4096];

#if !defined(_POSIX_SYNCHRONIZED_IO) || _POSIX_SYNCHRONIZED_IO == -1
    printf("No synchronized I/O -- comparison skipped\n");
#else
    /* Create the file so it can be checked */
    ec_neg1( fd = open("tmp", O_WRONLY | O_CREAT, PERM_FILE) )
    ec_neg1( close(fd) )
    switch (option_sync_io(PATHNAME)) {
    case OPT_YES:
        break;
    case OPT_NO:
        printf("sync unsupported on %s\n", PATHNAME);
        return;
    case OPT_ERROR:
        EC_FAIL
    }

    memset(buf, 1234, sizeof(buf));

    ec_neg1( fd = open(PATHNAME, O_WRONLY | O_TRUNC | O_DSYNC) )
    timestart();
    for (i = 0; i < SYNCREPS; i++)
        ec_neg1( write(fd, buf, sizeof(buf)) )
    ec_neg1( close(fd) )
    timestop("synchronized");

    ec_neg1( fd = open(PATHNAME, O_WRONLY | O_TRUNC) )
    timestart();
    for (i = 0; i < SYNCREPS; i++)
        ec_neg1( write(fd, buf, sizeof(buf)) )
    ec_neg1( close(fd) )
    timestop("unsynchronized");
#endif
    return;

EC_CLEANUP_BGN
    EC_FLUSH("backward");
    (void)close(fd);
EC_CLEANUP_END
}
```

The functions `timestart` and `timestop` are used to get the timings; we showed them in Section 1.7.2. Note the call to `option_sync_io` to check the pathname, which required us to first create the file to be used. The options in the three open

calls are a little unusual: The first uses `O_CREAT` without `O_TRUNC` because we just want to ensure that the file is there (we close it right away); the last two use `O_TRUNC` without `O_CREAT`, since we know it already exists.

On Linux, we got the results in Table 2.5 (Solaris times were similar; FreeBSD doesn't support the option).

Table 2.5 Synchronized vs. Unsynchronized I/O

Test	User*	System	Real
Synchronized	.03	5.57	266.45
Unsynchronized	.02	1.13	1.15

* Times in seconds.

(Real time is total elapsed time, including time waiting for I/O to complete.)

As you can see, synchronization is pretty costly. That's why you want to do it asynchronously, and I'll explain how in Section 3.9.[14]

2.17 `truncate` and `ftruncate` System Calls

truncate—truncate or stretch file by path

```
#include <unistd.h>

int truncate(
    const char *path,  /* pathname */
    off_t length       /* new length */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

ftruncate—truncate or stretch file by file descriptor

```
#include <unistd.h>

int ftruncate(
    int fd,            /* file descriptor */
    off_t length       /* new length */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

14. If this paragraph makes no sense to you, reread Section 2.16.1.

Stretching a file—making it bigger without actually writing lots of data—is easy, and I already showed how to do it: `lseek` to someplace beyond the end and `write` something. `truncate` and `ftruncate` can do that, but they're most useful in truncating (shrinking) a file, which was impossible on UNIX until they came along. (You used to have to write a completely new file and then rename it to the old name.)

Here's a rather contrived example. Note the unusual error checking for `write`, which I explained at the end of Section 2.9:

```
void ftruncate_test(void)
{
    int fd;
    const char s[] = "Those are my principles.\n"
      "If you don't like them I have others.\n"
      "\t--Groucho Marx\n";

    ec_neg1( fd = open("tmp", O_WRONLY | O_CREAT | O_TRUNC, PERM_FILE) )
    errno = 0;
    ec_false( write(fd, s, sizeof(s)) == sizeof(s) )
    (void)system("ls -l tmp; cat tmp");
    ec_neg1( ftruncate(fd, 25) )
    (void)system("ls -l tmp; cat tmp");
    ec_neg1( close(fd) )
    return;

EC_CLEANUP_BGN
    EC_FLUSH("ftruncate_test");
EC_CLEANUP_END
}
```

Here's the output:

```
-rw-r--r--   1 marc     sysadmin      80 Oct  2 14:03 tmp
Those are my principles.
If you don't like them I have others.
        --Groucho Marx
-rw-r--r--   1 marc     sysadmin      25 Oct  2 14:03 tmp
Those are my principles.
```

`ftruncate` is also used to size shared memory, as explained in Section 7.14.

Exercises

2.1. Change `lock` in Section 2.4.3 to store the login name in the lock file (use `getlogin`; Section 3.5.2). Add an argument to be used when `lock` returns `false` that provides the login name of the user who has acquired the lock.

2.2. Write a program that opens a file for writing with the `O_APPEND` flag and then writes a line of text on the file. Run several concurrent processes executing this program to convince yourself that the text lines won't get intermixed. Then recode the program without `O_APPEND` and use `lseek` to seek to the end before each `write`. Rerun the concurrent processes to see if the text gets intermixed now.

2.3. Rerun the buffered I/O timing tests in Section 2.12.2 with buffer sizes of 2, 57, 128, 256, 511, 513, and 1024. Try some other interesting numbers if you wish. If you have access to several versions of UNIX, run the experiment on each version and assemble the results into a table.

2.4. Enhance the BUFIO package (Section 2.12.2) to open a file for both reading and writing.

2.5. Add a `Bseek` function to the BUFIO package.

2.6. Write a `cat` command that takes no options. For extra credit, implement as many options as you can. Use the SUS as a specification.

2.7. Same as Exercise 2.6, but for the `tail` command.

This page intentionally left blank

3 Advanced File I/O

3.1 Introduction

This chapter picks up where Chapter 2 left off. First I'll extend the use of the I/O system calls already introduced to work on disk special files. I'll use that to look inside file systems. Then I'll introduce additional system calls that allow us to link to existing files; create, remove, and read directories; and obtain or modify file status information.

You're much less likely to use the advanced features covered in this chapter than those in Chapter 2. You will still benefit from knowing how to use them, however, because that will give you a more complete understanding of how file I/O works.

The program examples in this chapter are more extensive than those that have appeared so far. Careful study of these will be well worth your time, since they illustrate details not covered explicitly in the text.

3.2 Disk Special Files and File Systems

The section explains how to do I/O on disk special files, which is then used to access the internals of a UNIX file system. It also explains how mounting and unmounting work.

3.2.1 I/O on Disk Special Files

Until now we've done I/O exclusively through the kernel file system, using relatively high-level abstractions like file, directory, and i-node. As discussed in Section 2.12, the file system is implemented on top of the block I/O system,

which uses the buffer cache.[1] The block I/O system is accessed via a block special file, or block device, that interfaces directly to the disk. The disk is treated as a sequence of blocks whose size is a multiple of the sector size, which is usually 512.

There may be several physical disks, and each physical disk may be divided into pieces, each of which is called a *volume*, *partition*, or *file system*. (The term *file system* is confusing because it also describes part of the kernel. The context in which we use the term will make our intended meaning clear.)

Each volume corresponds to a special file whose name is typically formed from a device name, such as "hd," followed by numbers and letters that indicate which section of which physical disk it occupies. These special files are usually linked into the /dev directory, although they don't have to be. For example, on Linux the file /dev/hdb3 refers to the third partition of the second physical hard disk.

In principle, a disk special file may be operated on with I/O system calls just as if it were a regular file. It may be opened for reading and/or writing, read or written (in arbitrary units), seeked (to any byte boundary), and closed. The buffer cache is used (since it is a block special file), but within the volume, there are no directories, files, i-nodes, permissions, owners, sizes, times, and so on. One just deals with a giant array of numbered blocks.

In practice, most users can't perform any of these operations because permission is denied to them. Reading a disk that contains other users' files compromises their privacy, even though the disk appears haphazard when viewed as a special file (blocks are assigned to UNIX files and directories in no obvious order). Writing to a disk without going through the kernel file system would create even worse havoc.

On the other hand, if a volume is reserved for a user, and not used for other users' files and directories, then there is no conflict. Users implementing database managers or data acquisition systems may indeed want to have a volume set aside so they can access it as a block special file. A limitation, of course, is that there are only so many disk special files to go around. Usually when a disk special file is used by an application, that application is the only one, or certainly the main one, on the computer.

Potentially even faster than block disk special files are *raw* disk special files. These device drivers deal with the same areas of disk. However, these raw spe-

1. Newer systems actually use the virtual-memory system rather than the buffer cache, but the cache is still a useful abstract model for how the kernel handles files.

cial files are *character* devices, not block devices. That means they do not follow the block model, and they do not use the buffer cache. They do something even better.

When a read or write is initiated on a raw special file, the process is locked into memory (prevented from swapping) so no physical data addresses can change. Then, if the hardware and driver support it, the disk is ordered to transfer data using DMA. Data flows directly between the process's data segment and the disk controller, without going through the kernel at all. The size of the transfer may be more than a block at a time.

Usually, I/O on raw devices is less flexible than it is with regular files and block special files. I/O in multiples of a disk sector is required. The DMA hardware may require the process's buffer address to be on a particular boundary. Seeks may be required to be to a block boundary only. These restrictions don't bother designers of database-oriented file systems, since they find it convenient to view a disk as being made of fixed-size pages anyhow.

So far, we have seen that UNIX features vary somewhat from version to version. Here, however, we have a variation that depends also on the hardware, the device drivers, and even the installation. Since computers vary enormously in their I/O hardware, device drivers vary accordingly. Also, the goals of the implementation effort affect the importance of making raw I/O fast. On a general-purpose desktop system, for example, it may be of very little importance.

Like synchronized I/O (Section 2.16), raw I/O has one tremendous advantage and one tremendous disadvantage. The tremendous advantage is that, since the process waits for a write to complete, and since there is no question of which process owns the data, the process can be informed about physical write errors via a –1 return value and an `errno` code. There is a code defined (`EIO`), but whether it is ever passed on is up to the device driver implementor. You'll have to check the system-specific documentation or even look in the code for the device driver to be sure.

The tremendous disadvantage of raw I/O is that, since the process waits for a read or write to complete, and since the design of UNIX allows a process to issue only a single `read` or `write` system call at a time, the process does I/O very fast but not very often. For example, in a multi-user database application with one centralized database manager process (a common arrangement), there will be a colossal I/O traffic jam if raw I/O is used, since only one process does the I/O for everyone. The solutions are multiple database processes, multiple threads (Section 5.17), or asynchronous I/O (Section 3.9).

A minor disadvantage of raw I/O that we can dispose of immediately is that while the process is locked in memory, other processes may not be able to run because there is no room to swap them in. This would be a shame, because DMA does not use the CPU, and these processes could otherwise actually execute. The solution is just to add some more memory. At today's prices there is no excuse for not having enough memory to avoid most, if not all, swapping, especially on a computer that runs an application important enough to be doing raw I/O.

Table 3.1 summarizes the functional differences between I/O on regular files, on block disk devices, and on raw disk devices.

Table 3.1 I/O Feature Comparison

Features	Regular File	Block Disk Device	Raw Disk Device
directories, files, i-nodes, permissions, etc.	yes	no	no
buffer cache	yes	yes	no
I/O error returns, DMA	no	no	yes

To show the speed differences, on Solaris we timed 10,000 reads of 65,536 bytes each. We read a regular file, a block disk device (/dev/dsk/c0d0p0), and a raw disk device (/dev/rdsk/c0d0p0). As I did in Chapter 2, in Table 3.2 I report system time, user time, and real time, in seconds. Since reading a raw disk device provides features somewhat similar to reading a regular file with the `O_RSYNC` flag, as explained in Section 2.16.3, I included times for that as well.

Table 3.2 I/O Timing Comparison

I/O Type	User	System	Real
Regular file	0.40	21.28	441.75
Regular file, `O_RSYNC | O_DSYNC`	0.25	25.50	412.05
Block disk device	0.38	22.50	562.78
Raw disk device	0.10	2.87	409.70

As you can see, the speed advantages of raw I/O on Solaris are enormous, and I got similar results on Linux. I didn't bother running a comparison for writes, since I know that the two file types that use the buffer cache would have won hands down, and because I didn't have a spare volume to scribble on. Of course, it wouldn't have been a fair comparison, since the times with the buffer cache wouldn't have included the physical I/O, and the raw I/O time would have included little else.

3.2.2 Low-Level Access to a File System

It's illuminating to look at the internal structure of a file system by reading it as a disk device (assuming you have read permission). You'll probably never do this in an application program, but that's what low-level file utilities such as `fsck` do.

There are more file-system designs than there are UNIX systems: UFS (UNIX File System) on Solaris, FFS (Fast File System) on FreeBSD (also called UFS), and ReiserFS and Ext2fs on Linux. What follows here is loosely based on the original (1970s) UNIX file system and on FFS.

The raw disk is treated as a sequence of blocks of a fixed size, say 2048 bytes. (The actual size doesn't matter for this discussion.) The first block or so is reserved for a boot program (if the disk is bootable), a disk label, and other such administrative information.

The file system proper starts at a fixed offset from the start of the disk with the *superblock;* it contains assorted structural information such as the number of i-nodes, the total number of blocks in the volume, the head of a linked list of free blocks, and so on. Each file (regular, directory, special, etc.) uses one i-node and must be linked to from at least one directory. Files that have data on disk—regular, directory, and symbolic link—also have data blocks pointed to from their i-nodes. I-nodes start at a location pointed to from the superblock.

I-nodes 0 and 1 aren't used.[2] I-node 2 is reserved for the root directory (/) so that when the kernel is given an absolute path, it can start from a known place. No other i-numbers have any particular significance; they are assigned to files as needed.

2. I-node numbering starts at 1, which allows 0 to be used to mean "no i-node" (e.g., an empty directory). Historically, i-node 1 was used to collect bad disk blocks.

The following program, which is very specific to FreeBSD's implementation of FFS, shows one way to access the superblock and an i-node by reading the disk device /dev/ad0s1g, which is a raw disk device that happens to contain the /usr directory tree. That this is so can be seen by executing the `mount` command (which knows FFS as type "ufs"):

```
$ mount
/dev/ad0s1a on / (ufs, NFS exported, local)
/dev/ad0s1f on /tmp (ufs, local, soft-updates)
/dev/ad0s1g on /usr (ufs, local, soft-updates)
/dev/ad0s1e on /var (ufs, local, soft-updates)
procfs on /proc (procfs, local)
```

Here's the program:

```
#ifndef FREEBSD
#error "Program is for FreeBSD only."
#endif

#include "defs.h"
#include <sys/param.h>
#include <ufs/ffs/fs.h>
#include <ufs/ufs/dinode.h>

#define DEVICE "/dev/ad0s1g"

int main(int argc, char *argv[])
{
    int fd;
    long inumber;
    char sb_buf[((sizeof(struct fs) / DEV_BSIZE) + 1) * DEV_BSIZE];
    struct fs *superblock = (struct fs *)sb_buf;
    struct dinode *d;
    ssize_t nread;
    off_t fsbo, fsba;
    char *inode_buf;
    size_t inode_buf_size;

    if (argc < 2) {
        printf("Usage: inode n\n");
        exit(EXIT_FAILURE);
    }
    inumber = atol(argv[1]);
```

```
    ec_neg1( fd = open(DEVICE, O_RDONLY) )
    ec_neg1( lseek(fd, SBLOCK * DEV_BSIZE, SEEK_SET) )
    switch (nread = read(fd, sb_buf, sizeof(sb_buf))) {
    case 0:
        errno = 0;
        printf("EOF from read (1)\n");
        EC_FAIL
    case -1:
        EC_FAIL
    default:
        if (nread != sizeof(sb_buf)) {
            errno = 0;
            printf("Read only %d bytes instead of %d\n", nread,
sizeof(sb_buf));
            EC_FAIL
        }
    }
    printf("Superblock info for %s:\n", DEVICE);
    printf("\tlast time written = %s", ctime(&superblock->fs_time));
    printf("\tnumber of blocks in fs = %ld\n", (long)superblock->fs_size);
    printf("\tnumber of data blocks in fs = %ld\n",
      (long)superblock->fs_dsize);
    printf("\tsize of basic blocks in fs = %ld\n",
      (long)superblock->fs_bsize);
    printf("\tsize of frag blocks in fs = %ld\n",
      (long)superblock->fs_fsize);
    printf("\tname mounted on = %s\n", superblock->fs_fsmnt);

    inode_buf_size = superblock->fs_bsize;
    ec_null( inode_buf = malloc(inode_buf_size) )

    fsba = ino_to_fsba(superblock, inumber);
    fsbo = ino_to_fsbo(superblock, inumber);

    ec_neg1( lseek(fd, fsbtodb(superblock, fsba) * DEV_BSIZE, SEEK_SET) )
    switch (nread = read(fd, inode_buf, inode_buf_size)) {
    case 0:
        errno = 0;
        printf("EOF from read (2)\n");
        EC_FAIL
    case -1:
        EC_FAIL
    default:
        if (nread != inode_buf_size) {
            errno = 0;
            printf("Read only %d bytes instead of %d\n",
              nread, inode_buf_size);
            EC_FAIL
        }
    }
```

```
    d = (struct dinode *)&inode_buf[fsbo * sizeof(struct dinode)];
    printf("\ninumber %ld info:\n", inumber);
    printf("\tmode = 0%o\n", d->di_mode);
    printf("\tlinks = %d\n", d->di_nlink);
    printf("\towner = %d\n", d->di_uid);
    printf("\tmod. time = %s", ctime((time_t *)&d->di_mtime));
    exit(EXIT_SUCCESS);

EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
EC_CLEANUP_END
}
```

The complicated declaration for `sb_buf`:

```
char sb_buf[((sizeof(struct fs) / DEV_BSIZE) + 1) * DEV_BSIZE];
```

is to round up its size to an even sector, as that's a requirement for reading raw disk devices on FreeBSD.

The first `lseek` is to `SBLOCK * DEV_BSIZE`, which is the location of the superblock. `SBLOCK` happens to be 16, and `DEV_BSIZE` is the sector size, which is 512, as is true for most disks. The superblock always starts 16 sectors from the start of the file system. The first group of `printf`s print a few fields from the very long `fs` structure. This is that output:

```
Superblock info for /dev/ad0s1g:
        last time written = Mon Oct 14 15:25:25 2002
        number of blocks in fs = 1731396
        number of data blocks in fs = 1704331
        size of basic blocks in fs = 16384
        size of frag blocks in fs = 2048
        name mounted on = /usr
```

Next the program finds the i-node supplied as its argument (`argv[1]`) using some macros that are in one of the header files, as the arithmetic is very complicated on an FFS disk. I got the i-number of a file with the `-i` option on the `ls` command:

```
$ ls -li x
383642 -rwxr-xr-x  1 marc  marc  11687 Sep 19 13:29 x
```

And this was the rest of the output for i-node 383642:

```
inumber 383642 info:
        mode = 0100755
        links = 1
        owner = 1001
        mod. time = Thu Sep 19 13:29:30 2002
```

I'm not going to spend any more time on low-level disk access—all I really wanted was to illustrate how regular files and special files are related. But you might enjoy modifying the program a bit to display other information on a FreeBSD system if you have one, or even modifying it for another UNIX system that you do have.

3.2.3 `statvfs` and `fstatvfs` System Calls

Reading the superblock with `read` after finding it on a block or raw device is fun, but there's a much better way to do it on SUS1-conforming systems (Section 1.5.1) using the `statvfs` or `fstatvfs` system calls:

statvfs—get file system information by path

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

int statvfs(
     const char *path,      /* pathname */
     struct statvfs *buf    /* returned info */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

fstatvfs—get file system information by file descriptor

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

int fstatvfs(
     int fd,                /* file descriptor */
     struct statvfs *buf    /* returned info */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

Usually, you'll use `statvfs`, which returns information about the file system that contains the path given by its first argument. If you have a file opened, you can use `fstatvfs` instead, which returns the same information.

The standard defines fields for the `statvfs` structure, but implementations are not required to support them all. Also, most implementations support additional fields and additional flags for the `f_flag` field. You'll have to check your system's `man` page for the specifics. If an implementation doesn't support a standard field, the field is still there, but it won't contain meaningful information.

struct statvfs—structure for statvfs and fstatvfs

```
struct statvfs {
    unsigned long f_bsize;    /* block size */
    unsigned long f_frsize;   /* fundamental (fblock) size */
    fsblkcnt_t f_blocks;      /* total number of fblocks */
    fsblkcnt_t f_bfree;       /* number of free fblocks */
    fsblkcnt_t f_bavail;      /* number of avail. fblocks */
    fsfilcnt_t f_files;       /* total number of i-numbers */
    fsfilcnt_t f_ffree;       /* number of free i-numbers */
    fsfilcnt_t f_favail;      /* number of avail. i-numbers */
    unsigned long f_fsid;     /* file-system ID */
    unsigned long f_flag;     /* flags (see below) */
    unsigned long f_namemax;  /* max length of filename */
};
```

Some comments about this structure:

- Most file systems allocate fairly large blocks to files, given by the `f_bsize` field, to speed up access, but then finish off the file with smaller fragment blocks to avoid wasting space. The fragment-block size is called the *fundamental* block size and is given by the `f_frsize` field; we call it an *fblock* for short. The fields of type `fsblkcnt_t` are in units of fblocks, so, to get the total space in bytes, you would multiply `f_blocks` by `f_frsize`.
- The types `fsblkcnt_t` and `fsfilcnt_t` are unsigned, but otherwise implementation defined. They're typically `long` or `long long`. If you need to display one and you have a C99 compiler, cast it to `uintmax_t` and use `printf` format `%ju`; otherwise, cast it to `unsigned long long` and use format `%llu`.
- The SUS standards define only two flags for the `f_flag` field that indicate how the file system was mounted:

  `ST_RDONLY` is set if it's read only, and
  `ST_NOSUID` is set if the set-user-ID-on-execution and set-group-ID-on-execution bits are to be ignored on executable files (a security precaution).
- The term "available," which applies to the `f_bavail` and `f_favail` fields, means "available to nonsuperuser processes." It might be less that the corresponding "free" fields, so as to reserve a minimum amount of free space on systems where performance suffers when free space gets tight.

FreeBSD, being pre-SUS, doesn't support the `statvfs` or `fstatvfs` functions, but it does have a similar function called `statfs` whose structure has a different name and mostly different fields. For the basic information, it's possible to code in a way that works with both `statvfs` and `statfs`, as the following program illustrates.

```
#if _XOPEN_SOURCE >= 4
#include <sys/statvfs.h>
#define FCN_NAME statvfs
#define STATVFS 1

#elif defined(FREEBSD)
#include <sys/param.h>
#include <sys/mount.h>
#define FCN_NAME statfs

#else
#error "Need statvfs or nonstandard substitute"
#endif

void print_statvfs(const char *path)
{
    struct FCN_NAME buf;

    if (path == NULL)
        path = ".";
    ec_neg1( FCN_NAME(path, &buf) )
#ifdef STATVFS
    printf("block size = %lu\n", buf.f_bsize);
    printf("fundamental block (fblock) size = %lu\n", buf.f_frsize);
#else
    printf("block size = %lu\n", buf.f_iosize);
    printf("fundamental block size = %lu\n", buf.f_bsize);
#endif
    printf("total number of fblocks = %llu\n",
      (unsigned long long)buf.f_blocks);
    printf("number of free fblocks = %llu\n",
      (unsigned long long)buf.f_bfree);
    printf("number of avail. fblocks = %llu\n",
      (unsigned long long)buf.f_bavail);
    printf("total number of i-numbers = %llu\n",
      (unsigned long long)buf.f_files);
    printf("number of free i-numbers = %llu\n",
      (unsigned long long)buf.f_ffree);
#ifdef STATVFS
    printf("number of avail. i-numbers = %llu\n",
      (unsigned long long)buf.f_favail);
    printf("file-system ID = %lu\n", buf.f_fsid);
    printf("Read-only = %s\n",
      (buf.f_flag & ST_RDONLY) == ST_RDONLY ? "yes" : "no");
    printf("No setuid/setgid = %s\n",
      (buf.f_flag & ST_NOSUID) == ST_NOSUID ? "yes" : "no");
    printf("max length of filename = %lu\n", buf.f_namemax);
#else
```

```
    printf("Read-only = %s\n",
      (buf.f_flags & MNT_RDONLY) == MNT_RDONLY ? "yes" : "no");
    printf("No setuid/setgid = %s\n",
      (buf.f_flags & MNT_NOSUID) == MNT_NOSUID ? "yes" : "no");
#endif
    printf("\nFree space = %.0f%%\n",
      (double)buf.f_bfree * 100 / buf.f_blocks);
    return;

EC_CLEANUP_BGN
    EC_FLUSH("print_statvfs");
EC_CLEANUP_END
}
```

Here's the output we got on Solaris for the file system that contains the directory /home/marc/aup (Linux produces similar output):

```
block size = 8192
fundamental block (fblock) size = 1024
total number of fblocks = 4473046
number of free fblocks = 3683675
number of avail. fblocks = 3638945
total number of i-numbers = 566912
number of free i-numbers = 565782
number of avail. i-numbers = 565782
file-system ID = 26738695
Read-only = no
No setuid/setgid = no
max length of filename = 255

Free space = 82%
```

On FreeBSD this was the output for /usr/home/marc/aup:

```
block size = 16384
fundamental block size = 2048
total number of fblocks = 1704331
number of free fblocks = 1209974
number of avail. fblocks = 1073628
total number of i-numbers = 428030
number of free i-numbers = 310266
Read-only = no
No setuid/setgid = no

Free space = 71%
```

The `statvfs` (or `statfs`) system call is the heart of the well-known `df` ("disk free") command (see Exercise 3.2). Here's what it reported on FreeBSD:

```
$ df /usr
Filesystem  1K-blocks   Used   Avail Capacity  Mounted on
/dev/ad0s1g   3408662 988696 2147274    32%    /usr
```

The reported number of 1k-blocks, 3408662, equates to the number of 2k-blocks that our `print_statvfs` program showed, 1704331.

There's also a standard way to read the information in an i-node, so you don't have to read the device file as we did in the previous section. We'll get to that in Section 3.5.

3.2.4 Mounting and Unmounting File Systems

A UNIX system can have lots of disk file systems (hard disks, floppies, CD-ROMs, DVDs, etc.), but they're all accessible within a single directory tree that starts at the root. Connecting a file system to the existing hierarchy is called *mounting,* and disconnecting it *unmounting*. Figure 3.1 shows a large file system that contains the root (i-node 2, with directory entries x and y) and a smaller unconnected file system, with its own root, also numbered 2. (Recall that 2 is always the i-number for a root directory.)

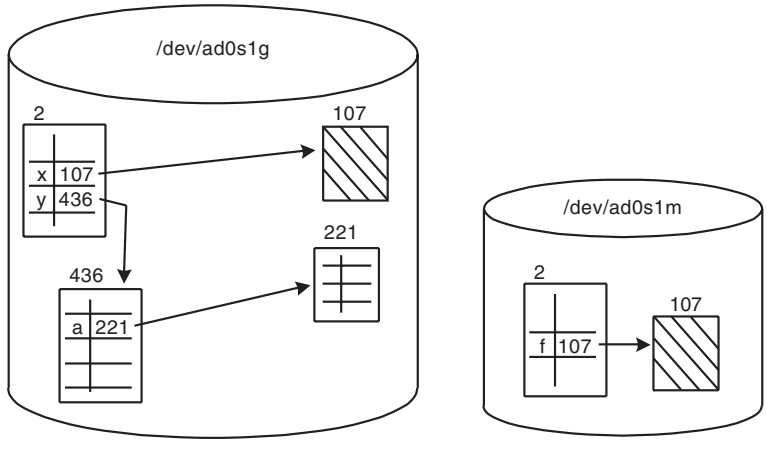


Figure 3.1 Two disconnected file systems.

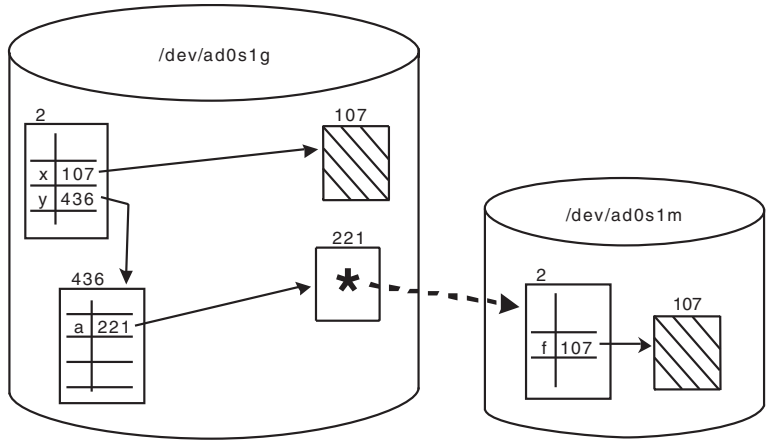


Figure 3.2 Mounted file system.

To mount the second file system, you need two things: Its device name, which is /dev/ad0s1m, and the directory where you want to connect it, for which we'll choose /y/a (i-node 221). Here's the command that creates the tree in Figure 3.2:

```
# mount /dev/ad0s1m /y/a
```

The old contents of directory /y/a are now hidden, and all references to that directory automatically become references to i-node ad0s1m:2, a notation that I'll use to mean "i-node 2 on device ad0s1m." Thus, the file ad0s1m:107 is now accessible as /y/a/f. When ad0s1m is unmounted, its contents are no longer accessible, and the old contents of ad0s1g:221 reappear.[3]

Every UNIX system has a superuser-only system call named `mount` and its undo-function `umount` (sometimes called `unmount`), but their arguments and exact behavior vary from system to system. Like other superuser-only functions, they're not standardized. In practice, these systems calls are used to implement the `mount` and `umount` commands and are almost never called

3. It's rare to actually have anything in a directory whose only purpose is to serve as a mount point, but it's allowed and occasionally used by system administrators to hide a directory.

directly. As an example, here are the Linux synopses, which I won't bother to explain completely:

mount—mount file system (nonstandard)

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

int mount(
     const char *source,        /* device */
     const char *target,        /* directory */
     const char *type,          /* type (e.g., ext2) */
     unsigned long flags,       /* mount flags (e.g., MS_RDONLY) */
     const void *data           /* file-system-dependent data */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

umount—unmount file system (nonstandard)

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

int umount(
     const char *target         /* directory */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

Normally, `umount` fails with the error `EBUSY` if any file or directory on it is in use. There's usually an alternative system call named `umount2` that has a flag as a second argument that forces the unmount, which is occasionally essential when the system has to be shut down.

From the application-programming perspective, except for one situation, you're normally not concerned with the fact that a given file or directory you're trying to access is on a file system separately mounted from some other file or directory, as long as you have a path name that works. In fact, *all* accessible file systems, even the root, had to be mounted at one time, perhaps during the boot sequence. The one situation has to do with links, which is what the next section is about.

3.3 Hard and Symbolic Links

An entry in a directory, consisting of a name and an i-number, is called a *hard link*. The other kind of link is a *symbolic link*, which I'll talk about in a moment. Figure 3.3 illustrates both. (The hex numbers in parentheses are device IDs, which I'll explain later.)

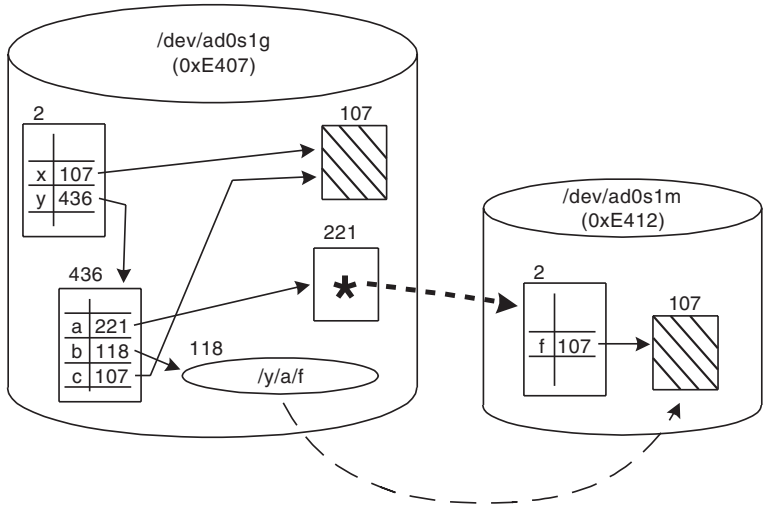


Figure 3.3 Hard and symbolic links.

3.3.1 Creating Hard Links (`link` System Call)

You get a hard link when a file of any type, including directory, is created. You can get additional hard links to nondirectories[4] with the `link` system call:

link—create hard link

```
#include <unistd.h>

int link(
     const char *oldpath,       /* old pathname */
     const char *newpath        /* new pathname */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

The first argument, `oldpath`, must be an existing link; it provides the i-number to be used. The second argument, `newpath`, indicates the name of the new link. The links are equal in every way, since UNIX has no notion of primary and secondary links. The process must have write permission on the directory that is to contain

4. On some systems the superuser can link to an existing directory, but doing so means the directory structure is no longer a tree, complicating system administration.

the new link. The link specified by the second argument must not already exist; `link` can't be used to change an existing link. In this case the old link must first be removed with `unlink` (Section 2.6), or the `rename` system call can be used (next section).

3.3.2 Renaming a File or Directory (`rename` System Call)

At first thought it seems easy to rename a file, if by that you mean "change the name in the directory entry." All you do is make a second hard link with `link` and remove the old link with `unlink`, but there are many complications:

- The file may be a directory, which you can't make a second hard link to.
- Sometimes you want the new name to be in a different directory, which is no problem if it's on the same file system, but a big problem if it isn't, because `link` only works within a file system. You can, of course, create a symbolic link (next section) in the new directory, but that's not what most people mean by "rename."
- If it's a directory that you want to "rename" ("move," really) between file systems, you have to move the whole subtree beneath it. Moving only empty directories is too restrictive.
- If there are multiple hard links and you're renaming the file within the same file system, they're going to be OK, because they reference the i-number, which won't change. But if you somehow manage to move the file to another file system, the old links will no longer be valid. What might happen, since you have to copy the file and then unlink the original, is that the old copy will stay around with all but the moved hard link still linked to it. Not good.
- If there are symbolic links to a path and you change that path, the symbolic links become dead ends. Also not good.

There are probably even more complications that we could list if we thought about it some more, but you get the idea, and we're already getting a headache. That's why the UNIX `mv` command, which is what you use to "rename" a file or directory, is a very complex piece of code. And it doesn't even deal with the last two problems.

Anyway, the `mv` command does pretty well, but to move a directory within a file system it needs one of the following alternative mechanisms.

- To always copy a directory and its subtree, followed by a deletion of the old directory, even if the directory is just renamed within its parent.
- A `link` system call that will work on directories, even if restricted to the superuser (The `mv` command can run with the set-user-ID bit on, to temporarily take on the privileges of the superuser.)
- A new system call that can rename a directory within a file system

The first alternative is bad—that's way too much work if all you want to do is change a few characters in the name! The second was ruled out by POSIX in favor of the third—a new system call, `rename`. Actually, it isn't new—it was in Standard C and had been in BSD. It's the only way to rename a directory without a full copy/unlink.

rename—rename file

```
#include <stdio.h>

int rename(
     const char *oldpath,       /* old pathname */
     const char *newpath        /* new pathname */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

`rename` is roughly equivalent to the sequence:

1. If `newpath` exists, remove it with `unlink` or `rmdir`.
2. `link(oldpath, newpath)`, even if `oldpath` is a directory.
3. Remove `oldpath` with `unlink` or `rmdir`.

What `rename` brings to the table are some additional features and rules:

- As I mentioned, step 2 works on directories, even if the process isn't running as the superuser. (You need write permission in `newpath`'s parent, though.)
- If `newpath` exists, it and `oldpath` have to be both files or both directories.
- If `newpath` exists and is a directory, it has to be empty. (Same rule as for `rmdir`.) In step 3, `oldpath`, if a directory, is removed even if nonempty, since its contents are also in `newpath`.
- If `rename` fails somewhere along the way, everything is left unchanged.

So you can see that while steps 1, 2, and 3 look like you could write them as a library function, there's no way to get it to work like the system call.

The `mv` command can do several things that the `rename` system call can't, such as move files and directories between file systems, and move groups of files and directories to a new parent directory. `rename` just supplies a small, but essential, part of `mv`'s functionality.

A final note: If `oldpath` is a symbolic link, `rename` operates on the symbolic link, not on what it points to; therefore, it might have been called `lrename`, but that would confuse the Standard C folks.

3.3.3 Creating Symbolic Links (`symlink` System Call)

As shown in Figure 3.3, if we wanted to create a second link to the file /x (ad0s1g:107) from directory /y, we could execute this call:

```
ec_neg1( link("/x", "/y/c") )
```

This makes /x and /y/c equivalent in every way; ad0s1g:107 isn't "in" either directory.

But now the transparency of the UNIX mounting mechanism breaks down: It's impossible to create a hard link from /y to /y/a/f (ad0s1m:107) because a directory entry has just an i-number to identify the linked-to object, not a device:i-number combination, which is what it takes to be unique. (I took the trouble in the figures to make 107 the i-number of two completely different files.) If you try to execute:

```
ec_neg1( link("/y/a/f", "/y/b") )
```

it will produce an `EXDEV` error.

The solution, as every experienced UNIX user knows, is to use a symbolic link. Unlike a hard link, where the i-number you want to link to goes right in the directory, a symbolic link is a small file containing the text of a path to the object you want to link to. The link we want is shown by the ellipse in Figure 3.3. It has an i-node of its own (ad0s1g:118), but normally any attempt to access it as /y/b causes the kernel to interpret the access as one to /y/a/f, which is on another file system.

A symbolic link can reference another symbolic link. Normally, such as when a symbolic link is passed to `open`, the kernel keeps dereferencing symbolic links until it finds something that isn't a symbolic link. This chaining can't happen with a hard link, as it directly references an i-node, which cannot be a hard link. (Although it can be a directory that *contains* a hard link.) In other words, if a path

leads to a hard link, the path is followed literally. If it leads to a symbolic link, the actual path followed depends on what that symbolic link references, until the end of the chain is reached.

Users create a symbolic link with the –s option of the `ln` command, but the system call that does the work is `symlink`:

symlink—create symbolic link

```
#include <unistd.h>

int symlink(
     const char *oldpath,       /* old pathname */
     const char *newpath        /* possible new pathname */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

Mostly, `symlink` works like `link`. In both cases, a hard link is created whose path is given by `newpath`; however, with `symlink`, that hard link is to a symbolic-link file that contains the string given by `newpath`.

The comment in the synopsis says "possible" because there is no validation of `newpath` at all—not to ensure that it's a valid path, or a valid anything else. You can even do this:

```
ec_neg1( symlink("lunch with Mike at 11:30", "reminder") )
```

and then see your reminder with the `ls` command (output wrapped to fit on the page):

```
$ ls -l reminder
lrwxrwxrwx   1 marc     sysadmin      24
Oct 16 12:04 reminder -> lunch with Mike at 11:30
```

This rather loose behavior is purposeful and leads to a useful feature of a symbolic link: What it references need not exist. Or, it may exist, but the file system that it's on may not be mounted.

The flip-side of the kernel's lack of interest in whether a symbolic-link target exists is that nothing is done to a symbolic link when its target is unlinked. By "unlinked," I mean, of course, "un-hard-linked," as the hard-link count going to zero is still what the kernel uses to determine if a file is no longer needed (i-node and data reclaimable). It could be impossible to do anything to adjust the symbolic links, because some of them could be on unmounted file systems. That's not possible with hard links, as they're internal to a file system.

So how do you get rid of a symbolic link? With `unlink` (refer to Figure 3.3):

```
ec_neg1( unlink("/y/b") )
```

This unlinks the symbolic link /y/b, and has no effect on the file it refers to, /y/a/f. Think of `unlink` as removing the hard link that its argument directly specifies.

OK, so how *do* you unlink a file that you want to refer to via its symbolic link? You could use another path, as the file has to be hard linked to *some* directory. But if all you have is the symbolic link path, you can read its contents with the `readlink` system call:

readlink—read symbolic link

```
#include <unistd.h>

ssize_t readlink(
     const char *path,          /* pathname */
     char *buf,                 /* returned text */
     size_t bufsize             /* buffer size */
);
/* Returns byte count or -1 on error (sets errno) */
```

`readlink` is unusual for a UNIX system call in that you can't assume that the returned string is NUL-terminated. You need to ensure that `buf` points to enough space to hold the contents of the symbolic link plus a NUL byte, and then pass its size less one as the third argument. Upon return, you'll probably want to force in a NUL byte, like this:

```
ssize_t n;
char buf[1024];

ec_neg1( n = readlink("/home/marc/mylink", buf, sizeof(buf) - 1) )
buf[n] = '\0';
```

If you want, you can now remove whatever /home/marc/mylink linked to and the symbolic link itself like this:

```
ec_neg1( unlink(buf) )
ec_neg1( unlink("/home/marc/mylink") )
```

`readlink` isn't the only case where we want to deal with a symbolic link itself, rather than what it references. Another is the `stat` system call (Section 3.5.1), for getting i-node information, which has a variant that doesn't follow symbolic links called `lstat`.

A problem with the `readlink` example code is the constant 1024, meant to represent a size large enough for the largest path name. Our code isn't going to blow up if the path is longer—we were too careful for that. It will simply truncate the returned path, which is still not good.

But if a file name in a directory is limited to, say, 255 bytes, and there's no limit on how deeply nested directories can get, how much space is enough? Certainly 1024 isn't the answer—that's only enough to handle four levels! Getting the answer is somewhat messy, so it gets a section all to itself. Read on.

3.4 Pathnames

This section explains how to determine the maximum length for a pathname and how to retrieve the pathname of the current directory.

3.4.1 How Long Is a Pathname?

If there were a fixed limit on the length of a pathname—a constant `_POSIX_MAX_PATH`, say—there would also be a limit on how deeply directories could nest. Even if the number were large, it would create problems, as it's pretty common for a UNIX system to effectively mount file systems on other computers via facilities like NFS. Thus, the limit has to be dynamically determined at runtime, and it has to be able to vary by file system.

That's exactly what `pathconf` and `fpathconf` are for (Section 1.5.6). On a system that conforms to POSIX1990—essentially all of them—you call it like this:

```
static long get_max_pathname(const char *path)
{
    long max_path;

    errno = 0;
    max_path = pathconf(path, _PC_PATH_MAX);
    if (max_path == -1) {
        if (errno == 0)
            max_path = 4096; /* guess */
        else
            EC_FAIL
    }
    return max_path + 1;
```

```
EC_CLEANUP_BGN
    return -1;
EC_CLEANUP_END
}
```

I added one to the value returned by `pathconf` because the documentation I've looked at is a little fuzzy as to whether the size includes space for a NUL byte. I'm inclined to assume it does not, figuring that the OS implementors are as unsure as I am.

I called this function for a locally mounted file system and for an NFS-mounted file system on each of my three test systems, and I got 1024 on FreeBSD and Solaris, and 4096 on Linux. But that was with my versions of those systems, and with my configuration. You need to use `pathconf` in your own code, rather than relying on my numbers.

Technically, for `_PC_PATH_MAX`, `pathconf` returns the size of the longest relative pathname from its argument path, not that of the longest absolute path; therefore, for complete accuracy you should call it with path of the root of the file system that you're interested in, and then add to that number the size of the path from the root to the root of that file system. But that's probably going too far—most everyone just calls it with an argument of `"/"` or `"."`.

The preceding was according to the POSIX and SUS standards. In practice, even systems that return some number from `pathconf` and `fpathconf` don't enforce that as a limit when creating directories, but they do enforce it when you try to pass a too-long path name to a system call that takes a path, such as `open` (the error is `ENAMETOOLONG`). On most systems, `getcwd` (next section) can return a string longer than the maximum returned by `pathconf` or `fpathconf`, if you give it a buffer that's big enough.

3.4.2 `getcwd` System Call

There's a straightforward system call for getting the path to the current directory. Like `readlink` (Section 3.3.3), the only tricky thing about it is knowing how big a buffer to pass in.

getcwd—get current directory pathname

```
#include <unistd.h>

char *getcwd(
     char *buf,             /* returned pathname */
     size_t bufsize         /* size of buf */
);
/* Returns pathname or NULL on error (sets errno) */
```

Here's a function that makes it even easier to call `getcwd`, as it automatically allocates a buffer to hold the path string. A call with a `true` argument tells it to free the buffer.

```
static char *get_cwd(bool cleanup)
{
    static char *cwd = NULL;
    static long max_path;

    if (cleanup) {
        free(cwd);
        cwd = NULL;
    }
    else {
        if (cwd == NULL) {
            ec_neg1( max_path = get_max_pathname(".") )
            ec_null( cwd = malloc((size_t)max_path) )
        }
        ec_null( getcwd(cwd, max_path) )
        return cwd;
    }
    return NULL;

EC_CLEANUP_BGN
    return NULL;
EC_CLEANUP_END
}
```

Here's some code that does what the standard `pwd` command does:

```
char *cwd;

ec_null( cwd = get_cwd(false) )
printf("%s\n", cwd);
(void) get_cwd(true);
```

When I ran it I got this on Solaris:

```
/home/marc/aup
```

There's enough functionality in this chapter to program `getcwd` ourselves, and I'll show the code in Section 3.6.4.

There's a similar system call, `getwd`, but it's obsolete and shouldn't be used in new programs.

3.5 Accessing and Displaying File Metadata

This section explains how to retrieve file metadata, such as the owner or modification time, and how to display it.

3.5.1 `stat`, `fstat`, and `lstat` System Calls

An i-node contains a a file's *metadata*—all the information about it other than its name, which really doesn't belong to it anyway, and its data, which the i-node points to. Reading an i-node straight from the disk, as we did in Section 3.2.2, is really primitive. Fortunately, there are three standardized system calls for getting at an i-node, `stat`, `fstat`, and `lstat`, and one very well-known command, `ls`.

stat—get file info by path

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

int stat(
    const char *path,      /* pathname */
    struct stat *buf       /* returned information */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

lstat—get file info by path without following symbolic link

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

int lstat(
    const char *path,      /* pathname */
    struct stat *buf       /* returned information */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

fstat—get file info by file descriptor

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

int fstat(
    int fd,                /* file descriptor */
    struct stat *buf       /* returned information */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

`stat` takes a path and finds the i-node by following it; `fstat` takes an open file descriptor and finds the i-node from the active i-node table inside the kernel. `lstat` is identical to `stat`, except that if the path leads to a symbolic link, the metadata is for the symbolic link itself, not for what it links to.[5] For all three, the same metadata from the i-node is rearranged and placed into the supplied `stat` structure.

Here's the `stat` structure, but be aware that implementations are free to add or rearrange fields, so make sure you include your local system's sys/stat.h header.

struct stat—structure for stat, fstat, and lstat

```
struct stat {
    dev_t st_dev;             /* device ID of file system */
    ino_t st_ino;             /* i-number */
    mode_t st_mode;           /* mode (see below) */
    nlink_t st_nlink;         /* number of hard links */
    uid_t st_uid;             /* user ID */
    gid_t st_gid;             /* group ID */
    dev_t st_rdev;            /* device ID (if special file) */
    off_t st_size;            /* size in bytes */
    time_t st_atime;          /* last access */
    time_t st_mtime;          /* last data modification */
    time_t st_ctime;          /* last i-node modification */
    blksize_t st_blksize;     /* optimal I/O size */
    blkcnt_t st_blocks;       /* allocated 512-byte blocks */
};
```

Some comments about this structure:

- A device ID (type `dev_t`) is a number that uniquely identifies a mounted file system, even when it's mounted with NFS. So, the combination `st_dev` and `st_ino` uniquely identifies the i-node. This is essentially what we were doing with notations like "ad0s1g:221" back in Section 3.2.4. Look again at Figure 3.3, where I've put the device IDs in hex below each device name.

  The standard doesn't specify how to break down a device ID, but essentially all implementations treat it as combination of major and minor device numbers, where the major number identifies the driver, and the minor number identifies the actual device, as the same driver can interface with all devices of the same type. Typically the minor number is the rightmost byte.

5. There's no `flstat` because there's no way to get a file descriptor open to a symbolic link. Any attempt to use a symbolic link in an `open` will open the linked-to file, not the symbolic link. And, if there were such a way, `fstat` would suffice, as the file descriptor would already specify the right object.

- The field `st_dev` is the device that contains the i-node; the field `st_rdev`, used only for special files, is the device that the special file represents. As an example, the special file /dev/ad0s1m is on the root file system, so its `st_dev` is that of the root file system. But `st_rdev` is the device that it refers to, which is a different disk entirely.
- The field `st_size` has different interpretations, depending on the type of i-node and the implementation. For an i-node that represents data on disk—regular files, directories, and symbolic links—it's the size of that data (the path, for symbolic links). For shared memory objects, it's the memory size. For pipes, it's the amount of data in the pipe, but that's nonstandard.
- The access time (`st_atime`) is updated whenever a file of any type is read, but not when a directory that appears in a path is only searched.
- The data modification time (`st_mtime`) is updated when a file is written, including when a hard link is added to or removed from a directory.
- The i-node modification time (`st_ctime`), also called the status-change time, is updated when the file's data is written or when the i-node is explicitly modified (e.g., by changing the owner or link count), but not when the access time is changed only as a side-effect of reading the file.
- The field `st_blksize` is in the `stat` structure so that an implementation can vary it by file, if it chooses to do so. In most cases it's probably the same as what's in the superblock (Section 3.2.3).
- If a file has holes, formed by seeking past its end and writing something, the value of `st_blocks * 512` could be less than `st_size`.
- `fstat` is especially useful when you have a file descriptor that did not come from opening a path, such as one for an un-named pipe or a socket. For these, `st_mode`, `st_ino`, `st_dev`, `st_uid`, `st_gid`, `st_atime`, `st_ctime`, and `st_mtime` are required to have valid values, but whether the other fields do is implementation dependent. Usually, though, for named and un-named pipes the `st_size` field contains the number of unread bytes in the pipe.

The field `st_mode` consists of bits that indicate the type of file (regular, directory, etc.), the permissions, and a few other characteristics. Rather than assuming specific bits, a portable application is supposed to use macros. First come the macros for the type of file.

st_mode—bit mask and values for type of file

```
S_IFMT                  /* all type-of-file bits */
S_IFBLK                 /* block special file */
S_IFCHR                 /* character special file */
S_IFDIR                 /* directory */
S_IFIFO                 /* named or un-named pipe */[6]
S_IFLNK                 /* symbolic link */
S_IFREG                 /* regular file */
S_IFSOCK                /* socket */
```

The macro `S_IFMT` defines the bits for the type of file; the others are values of those bits, *not* bit masks. Thus, the test for, say, a socket must not be coded as

```
if ((buf.st_mode & S_IFSOCK) == S_IFSOCK) /* wrong */
```

but as

```
if ((buf.st_mode & S_IFMT) == S_IFSOCK)
```

Or, you can use one of these testing macros each of which returns zero for false and nonzero for true, so they work as C Boolean expressions:

st_mode—type-of-file testing macros

```
S_ISBLK(mode)           /* is a block special file */
S_ISCHR(mode)           /* is a character special file */
S_ISDIR(mode)           /* is a directory */
S_ISFIFO(mode)          /* is a named or un-named pipe */
S_ISLNK(mode)           /* is a symbolic link */
S_ISREG(mode)           /* is a regular file */
S_ISSOCK(mode)          /* is a socket */
```

The test for a socket could be written as:

```
if (S_ISSOCK(buf.st_mode))
```

There are nine bits someplace in the mode for the permissions, and I already introduced the macros for them in Section 2.3, as the same macros are used with the `open` system call and a few others. The macros are of the form `S_I`**pwww** where **p** is the permission (`R`, `W`, or `X`) and **www** is for whom (`USR`, `GRP`, or `OTH`).

This time you do use them as bit masks, like this, which tests for group read *and* write permission:

```
if ((buf.st_mode & (S_IRGRP | S_IWGRP)) == (S_IRGRP | S_IWGRP))
```

In an attempt to clarify things (or perhaps make them worse—sorry!), here's a test for group read *or* write permission:

6. Actually, `S_IFIFO`, while correct, is misspelled; it should have been called `S_IFFIFO`.

```
if ((buf.st_mode & S_IRGRP) == S_IRGRP ||
  (buf.st_mode & S_IWGRP) == S_IWGRP)
```

We could have coded the read *and* write case like this if we wanted to:

```
if ((buf.st_mode & S_IRGRP) == S_IRGRP &&
  (buf.st_mode & S_IWGRP) == S_IWGRP)
```

There are a few other bits in the `st_mode` field, and I'll repeat the permission bits in the box to make them easier to find if you flip back to this page later:

st_mode—permission and miscellaneous bit masks

```
S_Ixwww     /* x = R|W|X, www = USR|GRP|OTH */
            /* examples: S_IRUSR, S_IWOTH */
S_ISUID     /* set-user-ID on execution */
S_ISGID     /* set-group-ID on execution */
S_ISVTX     /* directory restricted-deletion */
```

I explained set-user-ID and set-group-ID in Section 1.1.5. The `S_ISVTX` flag, if set, means that a file may be unlinked from a directory only by the superuser, the owner of that directory, or the owner of the file. If the flag is not set (the normal case), having write permission in the directory is good enough.[7]

A good way to show how to use the mode macros is to display the modes as the `ls` command does, with a sequence of 10 letters (e.g., `drwxr-xr-x`). Briefly, here are the rules that `ls` uses:

- The first letter indicates the type of file.
- The next nine letters are in three groups of three, for owner, group, and others, and are normally `r`, `w`, or `x` if the permission is set, and a dash if not.
- The owner and group execute letters become an `s` (lower case) if the set-user-ID or set-group-ID bit is on, in addition to the execute bit, and an `S` (upper case) if the set bit is on but the execute position is not. The second combination is possible, but meaningless, except for the combination of set-group-ID on and group execute off, which on some systems causes mandatory file locking, as explained in Section 7.11.5.
- The execute letter for others becomes a `t` if the restricted-deletion bit (`I_SVTX`) is set along with the execute (search) bit, and a `T` if the restricted-deletion bit is set but the execute bit is not.

7. That's the SUS definition. Historically, this bit was called the sticky bit, and it was used on executable-program files to keep the instruction segment of an often-used program (e.g., the shell) on the swap device. It's not usually used with modern, paging UNIX systems. (The letters "SVTX" come from "SaVe TeXt," as the instructions are also called the "text.")

The following function should help make the algorithm clear:

```
#define TYPE(b) ((statp->st_mode & (S_IFMT)) == (b))
#define MODE(b) ((statp->st_mode & (b)) == (b))

static void print_mode(const struct stat *statp)
{
    if (TYPE(S_IFBLK))
        putchar('b');
    else if (TYPE(S_IFCHR))
        putchar('c');
    else if (TYPE(S_IFDIR))
        putchar('d');
    else if (TYPE(S_IFIFO)) /* sic */
        putchar('p');
    else if (TYPE(S_IFREG))
        putchar('-');
    else if (TYPE(S_IFLNK))
        putchar('l');
    else if (TYPE(S_IFSOCK))
        putchar('s');
    else
        putchar('?');
    putchar(MODE(S_IRUSR) ? 'r' : '-');
    putchar(MODE(S_IWUSR) ? 'w' : '-');
    if (MODE(S_ISUID)) {
        if (MODE(S_IXUSR))
            putchar('s');
        else
            putchar('S');
    }
    else if (MODE(S_IXUSR))
        putchar('x');
    else
        putchar('-');
    putchar(MODE(S_IRGRP) ? 'r' : '-');
    putchar(MODE(S_IWGRP) ? 'w' : '-');
    if (MODE(S_ISGID)) {
        if (MODE(S_IXGRP))
            putchar('s');
        else
            putchar('S');
    }
    else if (MODE(S_IXGRP))
        putchar('x');
    else
        putchar('-');
    putchar(MODE(S_IROTH) ? 'r' : '-');
    putchar(MODE(S_IWOTH) ? 'w' : '-');
    if (MODE(S_IFDIR) && MODE(S_ISVTX)) {
```

```
        if (MODE(S_IXOTH))
            putchar('t');
        else
            putchar('T');
    }
    else if (MODE(S_IXOTH))
        putchar('x');
    else
        putchar('-');
}
```

This function doesn't terminate its output with a newline, for a reason that will be very clear soon, but I don't want to spoil the surprise. Here's some test code:

```
struct stat statbuf;

ec_neg1( lstat("somefile", &statbuf) )
print_mode(&statbuf);
putchar('\n');
ec_neg1( system("ls -l somefile") )
```

and the output, in case you didn't believe anything I was saying:

```
prw--w--w-
prw--w--w-   1 marc     sysadmin       0 Oct 17 13:57 somefile
```

Note that we called `lstat`, not `stat`, because we want information about the symbolic link itself if the argument is a symbolic link. We don't want to follow the link.

Next, a function to print the number of links, which is way easier than printing the mode. (Do you see where we're headed?)

```
static void print_numlinks(const struct stat *statp)
{
    printf("%5ld", (long)statp->st_nlink);
}
```

Why the cast to `long`? Well, we really don't know what the type `nlink_t` is, other than that it's an integer type. We need to cast it something concrete to ensure that it matches the format in the `printf`. The same goes for some other types in the `stat` structure, as we'll see later in this chapter.

Next we want to print the owner and group names, but for that we need two more library functions.

3.5.2 `getpwuid`, `getgrgid`, and `getlogin` System Calls

The owner and group numbers are easy to get at, as they're in the `stat` structure's `st_uid` and `st_gid` fields, but we want their names. For that there are two functions, `getpwuid` and `getgrgid`, that aren't really system calls, since the information they need is in the password and group files, which any process can read for itself if it wants to take the trouble. A problem with that, though, is that the file layout isn't standardized.

Here's what the password-file entry for "marc" looks like on Solaris, which is fairly typical:

```
$ grep marc /etc/passwd
marc:x:100:14::/home/marc:/bin/sh
```

(The one thing that is *not* in the password file is the password! It's stored encrypted in another file that only the superuser can read.)

On this system "marc" is a member of a few groups, but group 14 is his login group:

```
$ grep 14 /etc/group
sysadmin::14:
```

`getpwuid`—get password-file entry

```
#include <pwd.h>

struct passwd *getpwuid(
    uid_t uid          /* user ID */
);
/* Returns pointer to structure or NULL on error (sets errno) */
```

`struct passwd`—structure for `getpwuid`

```
struct passwd {
    char *pw_name;     /* login name */
    uid_t pw_uid;      /* user ID */
    gid_t pw_gid;      /* group ID */
    char *pw_dir;      /* login directory */
    char *pw_shell;    /* login shell */
};
```

`getgrgid`—get group-file entry

```
#include <grp.h>

struct group *getgrgid(
    gid_t gid          /* group ID */
);
/* Returns pointer to structure or NULL on error (sets errno) */
```

struct group—structure for `getgrgid`

```
struct group {
    char *gr_name;     /* group name */
    gid_t gr_gid;      /* group ID */
    char **gr_mem;     /* member-name array (NULL terminated */
};
```

As I've said before, the two structures here show just the standardized fields. Your implementation may have more, so use the structures as defined in the headers.

Now we can use the two look-up functions to print the login and group names, or just the numbers if the names aren't known. Not finding the name is a common occurrence when a file system is mounted across a network, as a user on one system may not have a login on another.

```
static void print_owner(const struct stat *statp)
{
    struct passwd *pwd = getpwuid(statp->st_uid);

    if (pwd == NULL)
        printf(" %-8ld", (long)statp->st_uid);
    else
        printf(" %-8s", pwd->pw_name);
}

static void print_group(const struct stat *statp)
{
    struct group *grp = getgrgid(statp->st_gid);

    if (grp == NULL)
        printf(" %-8ld", (long)statp->st_gid);
    else
        printf(" %-8s", grp->gr_name);
}
```

While we're at it, here's a function to get the name under which the user logged in:

getlogin—get login name

```
#include <unistd.h>

char *getlogin(void);
/* Returns name or NULL on error (sets errno) */
```

3.5.3 More on Displaying File Metadata

Continuing with the `stat` structure, here's a function to print the file size. In the case of special files, we print the major and minor device numbers instead, as the size isn't

meaningful. As I said in Section 3.5, how they're encoded in the device ID isn't standardized, but taking the rightmost 8 bits as the minor number usually works:

```
static void print_size(const struct stat *statp)
{
    switch (statp->st_mode & S_IFMT) {
    case S_IFCHR:
    case S_IFBLK:
        printf("%4u,%4u", (unsigned)(statp->st_rdev >> 8),
          (unsigned)(statp->st_rdev & 0xFF));
        break;
    default:
        printf("%9lu", (unsigned long)statp->st_size);
    }
}
```

Next comes the file data-modification date and time, using the Standard C function `strftime` to do the hard work. The functions `time` and `difftime` are in Standard C, too. (See Section 1.7.1 for all three.) We don't normally print the year, unless the time is 6 months away, in which case we skip the time to make room:[8]

```
static void print_date(const struct stat *statp)
{
    time_t now;
    double diff;
    char buf[100], *fmt;

    if (time(&now) == -1) {
        printf(" ????????????");
        return;
    }
    diff = difftime(now, statp->st_mtime);
    if (diff < 0 || diff > 60 * 60 * 24 * 182.5) /* roughly 6 months */
        fmt = "%b %e  %Y";
    else
        fmt = "%b %e %H:%M";
    strftime(buf, sizeof(buf), fmt, localtime(&statp->st_mtime));
    printf(" %s", buf);
}
```

The last thing we want to print is the file name, which isn't in the `stat` structure, of course, because it isn't in the i-node. It's a little tricky, however, because if the name is a symbolic link, we want to print both it and its contents:

8. Generally, our policy is to check for all errors except when formatting (for printing) and printing. `difftime` isn't an exception—it's one of those functions that has no error return.

```
static void print_name(const struct stat *statp, const char *name)
{
    if (S_ISLNK(statp->st_mode)) {
        char *contents = malloc(statp->st_size + 1);
        ssize_t n;

        if (contents != NULL && (n = readlink(name, contents,
          statp->st_size)) != -1) {
            contents[n] = '\0'; /* can't assume NUL-terminated */
            printf(" %s -> %s", name, contents);
        }
        else
            printf(" %s -> [can't read link]", name);
        free(contents);
    }
    else
        printf(" %s", name);
}
```

Recall that back in Section 3.3.3 when I introduced `readlink` I pointed out that we needed to know the length of the longest pathname in order to size the buffer. This function shows a more straightforward way, since the exact size of the path (not including the NUL byte) is in the `st_size` field for symbolic links.[9] Note that I allocate one byte more than `st_size`, but pass `st_size` to `readlink` so as to guarantee room for a NUL byte, in case `readlink` doesn't supply one, which it isn't required to do.

Now, for what you've been waiting for, a program that puts all the printing functions together:

```
int main(int argc, char *argv[])
{
    int i;
    struct stat statbuf;

    for (i = 1; i < argc; i++) {
        ec_neg1( lstat(argv[i], &statbuf) )
        ls_long(&statbuf, argv[i]);
    }
    exit(EXIT_SUCCESS);

EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
EC_CLEANUP_END
}
```

9. Most modern file systems store short symbolic links right in the i-node, rather than taking up data blocks, but still the `st_size` field gives the size.

```
static void ls_long(const struct stat *statp, const char *name)
{
    print_mode(statp);
    print_numlinks(statp);
    print_owner(statp);
    print_group(statp);
    print_size(statp);
    print_date(statp);
    print_name(statp, name);
    putchar('\n');
}
```

Here's our simplified `ls` command in action:

```
$ aupls /dev/tty
crw-rw-rw-    1 root     root       5,   0 Mar 23  2002 /dev/tty
$ aupls a.tmp a.out util
lrwxrwxrwx    1 marc     sysadmin        5 Jul 29 13:30 a.tmp -> b.tmp
-rwxr-xr-x    1 marc     sysadmin     8392 Aug  1  2001 a.out
drwxr-xr-x    3 marc     sysadmin      512 Aug 28 12:26 util
```

Notice from the last line that it doesn't know how to list a directory—it only lists the names given as arguments. To add that feature we need to find out how to read a directory to get at its links. That's coming right up.

3.6 Directories

This section covers reading and removing directories, changing the current directory, and walking up and down the directory tree.

3.6.1 Reading Directories

Underneath, UNIX systems nearly always implement directories as regular files, except that they have a special bit set in their i-node and the kernel does not permit writing on them. On some systems you're allowed to read a directory with `read`, but the POSIX and SUS standards don't require this, and they also don't specify the internal format of the directory information. But it's interesting to snoop, so I wrote a little program to read the first 96 bytes of the current directory and dump out the bytes as characters (if they're printable) and in hex:

```
static void dir_read_test(void)
{
```

```
    int fd;
    unsigned char buf[96];
    ssize_t nread;

    ec_neg1( fd = open(".", O_RDONLY) )
    ec_neg1( nread = read(fd, buf, sizeof(buf)) )
    dump(buf, nread);
    return;

EC_CLEANUP_BGN
    EC_FLUSH("dir_read_test");
EC_CLEANUP_END
}

static void dump(const unsigned char *buf, ssize_t n)
{
    int i, j;

    for (i = 0; i < n; i += 16) {
        printf("%4d  ", i);
        for (j = i; j < n && j < i + 16; j++)
            printf("  %c", isprint((int)buf[j]) ? buf[j] : ' ');
        printf("\n      ");
        for (j = i; j < n && j < i + 16; j++)
            printf(" %.2x", buf[j]);
        printf("\n\n");
    }
    printf("\n");
}
```

This was the output (minus the "ec" tracing stuff) on Linux:

```
*** EISDIR (21: "Is a directory") ***
```

So, no luck there. But, on FreeBSD (and Solaris), pay dirt:

```
   0    `                       .           V
       60 d8 05 00 0c 00 04 01 2e 00 00 00 56 d8 05 00

  16                .  .        b
       0c 00 04 02 2e 2e 00 00 62 d8 05 00 0c 00 08 02

  32    m  2        y                       k     C
       6d 32 00 c0 79 d8 05 00 0c 00 08 01 6b 00 43 c6

  48                            p  c  s  y  n  c  _  s
       b3 da 05 00 18 00 08 0c 70 63 73 79 6e 63 5f 73

  64    i  g  .  o
       69 67 2e 6f 00 00 00 00 e3 e3 05 00 10 00 08 06

  80    t  i  m  e  .  o        r
       74 69 6d 65 2e 6f 00 c6 72 d8 05 00 0c 00 08 02
```

This appears at first to be a mishmash, but looking closer it starts to makes some sense. Six names are visible: `.`, `..`, `m2`, `k`, `pcsync_sig.o`, and `time.o`. We know that their i-numbers have to be in there, too, and to find them in hex we use the `-i` option of the `ls` command and the `dc` ("desk calculator") to translate from decimal to hex (comments in italics were added later):[10]

```
$ ls -ldif . .. m2 k
383072 drwxr-xr-x  2 marc  marc  2560 Oct 18 11:02 .
383062 drwxr-xr-x  9 marc  marc   512 Oct 14 18:05 ..
383074 -rwxrwxrwx  1 marc  marc    55 Jul 25 11:14 m2
383097 -rwxr--r--  1 marc  marc   138 Sep 19 13:28 k
$$$ dc
16o           make 16 the output radix
383072p       push the i-number onto the stack and print it
5D860         dc printed i-number in hex (radix 16)
383062p       … ditto …
5D856
q             quit
```

And, sure enough, the numbers 5D860 and 5D856 show up on the second line (first hex line) of the dump. There are some other numbers for each entry there as well, for such things as string sizes, but we really don't want to dig any deeper. There's a better, standardized way to read a directory, using system calls designed just for that purpose:

opendir—open directory

```
#include <dirent.h>

DIR *opendir(
    const char *path       /* directory pathname */
);
/* Returns DIR pointer or NULL on error (sets errno) */
```

closedir—close directory

```
#include <dirent.h>

int closedir(
    DIR *dirp              /* DIR pointer from opendir */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

10. The `-ldif` options to `ls` mean long listing, don't traverse directories, show i-numbers, and don't sort.

readdir—read directory

```
#include <dirent.h>

struct dirent *readdir(
     DIR *dirp              /* DIR pointer from opendir */
);
/* Returns structure or NULL on EOF or error (sets errno) */
```

struct dirent—structure for readdir

```
struct dirent {
     ino_t d_ino;           /* i-number */
     char d_name[];         /* name */
};
```

rewinddir—rewind directory

```
#include <dirent.h>

void rewinddir(
     DIR *dirp              /* DIR pointer from opendir */
);
```

seekdir—seek directory

```
#include <dirent.h>

void seekdir(
     DIR *dirp,             /* DIR pointer from opendir */
     long loc               /* location */
);
```

telldir—get directory location

```
#include <dirent.h>

long telldir(
     DIR *dirp              /* DIR pointer from opendir */
);
/* Returns location (no error return) */
```

These functions work as you would expect: You start with `opendir`, which gives you a pointer to a `DIR` (analogous to calling the Standard C function `fopen` to get a pointer to a `FILE`). That pointer is then used as an argument to the other five functions. You call `readdir` in a loop until it returns `NULL`, which means that you got to the end if `errno` wasn't changed by `readdir`, and an error if it was. (So things don't get confused, you should set `errno` to zero before you call `readdir`.) `readdir` returns a pointer to a `dirent` structure which contains the i-number and name for one entry. When you're done with the `DIR`, you close it with `closedir`.

The i-number returned in the `d_ino` field by `readdir` isn't particularly useful because, if that entry is a mount point, it will be the premount i-number, not the one that reflects the current tree. For example, in Figure 3.3 (back in Section 3.3), `readdir` would give us 221 for directory entry /y/a, but, as it is a mount point, the effective i-node is ad0s1m:2 (plain 2 isn't specific enough, as device ad0s1g also has a 2). I-node 221 isn't even accessible. Therefore, when reading directories to navigate the directory tree, as I will do when I show later how to get the path of the current directory in Section 3.6.4, you have to get the correct i-number for a directory entry with a call to one of the `stat` functions, not from the `d_ino` field.

`rewinddir` is occasionally useful to go back to the beginning, so you can read a directory again without having to close and reopen it. `seekdir` and `telldir` are more rarely used. The `loc` argument to `seekdir` must be something you got from `telldir`; you can't assume it's an entry number, and you can't assume that directory entries take up fixed-width positions, as we've already seen.

`readdir` is one of those functions that uses a single statically allocated structure that the returned pointer points to. It's very convenient but not so good for multithreaded programs, so there's a stateless variant of `readdir` that uses memory you pass in instead:

readdir_r—read directory

```
#include <dirent.h>

int readdir_r(
     DIR *restrict dirp,        /* DIR pointer from opendir */
     struct dirent *entry,      /* structure to hold entry */
     struct dirent **result     /* result (pointer or NULL) */
);
/* Returns 0 on success or error number on error (errno not set) */
```

You need to pass a pointer to a `dirent` structure into `readdir_r` that's large enough to hold a name of at least `NAME_MAX + 1` elements. You get the value for `NAME_MAX` with a call to `pathconf` or `fpathconf`, with the directory as an argument, similar to what we had to do in Section 3.4 to get the value for the maximum path length. `readdir_r` returns its result through the `result` argument; its interpretation is identical to the return value from `readir`, but this time you don't check `errno`—the error number (or zero) is the return value from the function. You can use the `ec_rv` macro (Section 1.4.2) to check it, as I do in this example, which lists the names and i-numbers in the current directory.

```
static void readdir_r_test(void)
{
    bool ok = false;
    long name_max;
    DIR *dir = NULL;
    struct dirent *entry = NULL, *result;

    errno = 0;
    /* failure with errno == 0 means value not found */
    ec_neg1( name_max = pathconf(".", _PC_NAME_MAX) )
    ec_null( entry = malloc(offsetof(struct dirent, d_name) +
      name_max + 1) )
    ec_null( dir = opendir(".") )
    while (true) {
        ec_rv( readdir_r(dir, entry, &result) )
        if (result == NULL)
            break;
        printf("name: %s; i-number: %ld\n", result->d_name,
          (long)result->d_ino);
    }
    ok = true;
    EC_CLEANUP

EC_CLEANUP_BGN
    if (dir != NULL)
        (void)closedir(dir);
    free(entry);
    if (!ok)
        EC_FLUSH("readdir_r_test");
EC_CLEANUP_END
}
```

Some comments on this code:

- A –1 return from `pathconf` with `errno` zero means that there is no limit for the names in a directory, which can't be. But we want our code to handle this case, so we take the shortcut of setting `errno` to zero and then treating all –1 returns as errors. The comment is for anyone who actually gets an error message with an `errno` value of zero. The idea is that we want to ensure that our code handles the impossible cases without dealing with them in an overly complicated way, which we wouldn't be able to test anyway, as we can't create test cases for impossible occurrences.
- The allocation with `malloc` is tricky. All we know for sure about the `dirent` structure is that the field `d_ino` is in it someplace (there may be other, nonstandard fields), and that `d_name` is last. So, the offset of `d_name` plus the size we need is the only safe way to calculate the total size, taking

into account both holes in the structure (allowed by Standard C) and hidden fields.

- `EC_CLEANUP` jumps to the cleanup code, as explained in Section 1.4.2. The Boolean `ok` tells us whether there was an error.

What a pain! It's much easier to use `readdir`, which is fine if you're not multithreading or you can ensure that only one thread at a time is reading a directory:

```
static void readdir_test(void)
{
    DIR *dir = NULL;
    struct dirent *entry;

    ec_null( dir = opendir(".") )
    while (errno = 0, (entry = readdir(dir)) != NULL)
        printf("name: %s; i-number: %ld\n", entry->d_name,
          (long)entry->d_ino);
    ec_nzero( errno )
    EC_CLEANUP

EC_CLEANUP_BGN
    if (dir != NULL)
        (void)closedir(dir);
    EC_FLUSH("readdir_test");
EC_CLEANUP_END
}
```

The one tricky thing here is stuffing the call to `readdir` into the `while` test: The first part of the comma expression zeroes `errno`, and the second part determines the value of the expression as a whole, which is what the `while` tests. You may want a less dense version, which is fine, but don't do this:

```
errno = 0;
while ((entry = readdir(dir)) != NULL) { /* wrong */
    /* process entry */
}
```

because `errno` might be reset during the processing of the entry. You should set `errno` to zero each time you call `readdir`.[11]

Anyway, here are the first few lines of output (from either example):

11. I know what you're thinking: That half the difficulty of programming UNIX is dealing with `errno`. You're right! But don't shoot me, I'm only the messenger! See Appendix B for an easier way.

```
name: .; i-number: 383072
name: ..; i-number: 383062
name: m2; i-number: 383074
name: k; i-number: 383097
```

Now that we can read a directory, let's put a `readdir` loop together with the `ls_long` function that appeared at the end of Section 3.5.3 to get an `ls` command that can list a directory:

```
int main(int argc, char *argv[]) /* has a bug */
{
    bool ok = false;
    int i;
    DIR *dir = NULL;
    struct dirent *entry;
    struct stat statbuf;

    for (i = 1; i < argc; i++) {
        ec_neg1( lstat(argv[i], &statbuf) )
        if (!S_ISDIR(statbuf.st_mode)) {
            ls_long(&statbuf, argv[i]);
            ok = true;
            EC_CLEANUP
        }
        ec_null( dir = opendir(argv[i]) )
        while (errno = 0, ((entry = readdir(dir)) != NULL)) {
            ec_neg1( lstat(entry->d_name, &statbuf) )
            ls_long(&statbuf, entry->d_name);
        }
        ec_nzero( errno )
    }
    ok = true;
    EC_CLEANUP

EC_CLEANUP_BGN
    if (dir != NULL)
        (void)closedir(dir);
    exit(ok ? EXIT_SUCCESS : EXIT_FAILURE);
EC_CLEANUP_END
}
```

This program ran just fine when I listed the current directory, as this output shows (only the first few lines are shown):

```
$ aupls .
drwxr-xr-x    2 marc     marc         2560 Oct 18 12:20 .
drwxr-xr-x    9 marc     marc          512 Oct 14 18:05 ..
-rwxrwxrwx    1 marc     marc           55 Jul 25 11:14 m2
-rwxr--r--    1 marc     marc          138 Sep 19 13:28 k
```

But when I tried it on the /tmp directory, I got this:

```
$ aupls /tmp
drwxr-xr-x    2 marc     marc         2560 Oct 18 12:20 .
drwxr-xr-x    9 marc     marc          512 Oct 14 18:05 ..
ERROR:  0: main [/aup/c3/aupls.c:422] lstat(entry->d_name, &statbuf)
               *** ENOENT (2: "No such file or directory") ***
```

The symptom is that the call to `lstat` in the `readdir` loop fails to find the name, even though we just read it from the directory. The cause is that we're trying to call `lstat` with the path "auplog.tmp," which would be fine if /tmp were the current directory, but it isn't.[12] Do a `cd` first and it works (only first few lines shown):

```
$ cd /tmp
$ /usr/home/marc/aup/aupls .
drwxrwxrwt    3 root     wheel        1536 Oct 18 12:20 .
drwxr-xr-x   18 root     wheel         512 Jul 25 08:01 ..
-rw-r--r--    1 marc     wheel      189741 Aug 30 11:22 auplog.tmp
-rw-------    1 marc     wheel           0 Aug  5 13:23 ed.7GHAhk
```

The fix is to put a call to the `chdir` system before the `readdir` loop, which is a great excuse to go on to the next section.

3.6.2 `chdir` and `fchdir` System Calls

Everybody knows what a shell's `cd` command does, and here's the system call that's behind it:

chdir—change current directory by path

```
#include <unistd.h>

int chdir(
    const char *path          /* pathname */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

fchdir—change current directory by file descriptor

```
#include <unistd.h>

int fchdir(
    int fd                    /* file descriptor */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

12. The first two entries worked only because `.` and `..` are in every directory. The ones that printed weren't the right ones, however.

As you would expect, the argument to `chdir` can be a relative or absolute path, and whatever i-node it leads to, if a directory, becomes the new current directory.

`fchdir` takes a file descriptor open to a directory as its argument. But wait! Didn't I say in Section 3.6.1 that opening a directory was nonstandard? No, I did not; I said that *reading* one was nonstandard. There is exactly one standard, useful reason to open a directory, and that's to get a file descriptor for use with `fchdir`. You have to open it with `O_RDONLY`, though; you're not allowed to open it for writing.

Since you can't read a directory portably, why have `fchdir`—why not always use `chdir`, with a pathname? Because an `open` paired with an `fchdir` is an excellent way to bookmark your place and return to it. Compare these two techniques:

1. get pathname of current directory
2. something that changes current directory
3. chdir using pathname from step 1

1. open current directory
2. something that changes current directory
3. fchdir using file descriptor from step 1

The technique on the left is inferior because:

- It's a lot of trouble to get a path to the current directory, as we saw in Section 3.4.2.
- It's very time consuming, as we'll see when we do it ourselves, in Section 3.6.4.

In some cases, if you go down just one level, you can get back to where you were with:

```
ec_neg1( chdir("..") )
```

which doesn't require you to have a pathname, but it's still better to use the `open`/`fchdir` approach because it doesn't depend on knowledge of how your program traverses directories. You can't use it, however, if you don't have read permission on the directory.

So now let's fix the version of `aupls` from the previous section so it changes to a directory before reading it. We do need to return to where we were because there may be multiple arguments to process, and each assumes that the current directory is unchanged. Here's just the repaired middle part:

```
ec_null( dir = opendir(argv[i]) )
ec_neg1( fd = open(".", O_RDONLY) )
ec_neg1( chdir(argv[i]) )
while (errno = 0, ((entry = readdir(dir)) != NULL)) {
    ec_neg1( lstat(entry->d_name, &statbuf) )
    ls_long(&statbuf, entry->d_name);
}
ec_nzero( errno )
ec_neg1( fchdir(fd) )
```

There's still a problem. If you want to try to find it before I tell you what it is, stop reading *here*.

The problem is that if there's an error between the `chdir` and `fchdir` calls that jumps to the cleanup code, the call to `fchdir` won't be executed, and the current directory won't be restored. It's OK in this program because all such errors terminate the process, and the current directory is unique to each process. (The shell's current directory won't be affected.) But, still, if the error processing should be changed at some point, or if this code is copied and pasted into another program, things could go awry.

A good fix (not shown) is to put a second call to `fchdir` in the cleanup code, so it gets executed no matter what. You can initialize `fd` to –1 and test for that so you don't use it unless it has a valid value.

3.6.3 `mkdir` and `rmdir` System Calls

There are two system calls for making and removing a directory:

mkdir—make directory

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

int mkdir(
    const char *path,          /* pathname */
    mode_t perms               /* permissions */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

rmdir—remove directory

```
#include <unistd.h>

int rmdir(
    const char *path           /* pathname */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

For `mkdir`, the permissions, how they interact with the file-creation mask, and the ownership of the new directory are the same as for `open` (Section 2.4). It automatically creates the special `.` and `..` links.

`rmdir` acts pretty much like `unlink`, which isn't allowed on directories.[13] One big restriction is that the directory to be removed has to be empty (except for `.` and `..`). If it's not empty, you have to remove its links first, which could involve multiple calls to `unlink` and multiple calls to `rmdir`, starting at the bottom of the subtree and working up. If that's what you really want to do, it's much easier just to write:

```
ec_neg1( system("rm -rf somedir") )
```

since the `rm` command knows how to walk a directory tree.

Here's an illustration of what we just said:

```
void rmdir_test(void)
{
    ec_neg1( mkdir("somedir", PERM_DIRECTORY) )
    ec_neg1( rmdir("somedir") )
    ec_neg1( mkdir("somedir", PERM_DIRECTORY) )
    ec_neg1( close(open("somedir/x", O_WRONLY | O_CREAT, PERM_FILE)) )
    ec_neg1( system("ls -ld somedir; ls -l somedir") )
    if (rmdir("somedir") == -1)
        perror("Expected error");
    ec_neg1( system("rm -rf somedir") )
    ec_neg1( system("ls -ld somedir") )
    return;

EC_CLEANUP_BGN
    EC_FLUSH("rmdir_test");
EC_CLEANUP_END
}
```

`PERM_DIRECTORY` and `PERM_FILE` were explained in Section 1.1.5. The line that creates the file is a little weird, but handy when you just want to create a file. The only bad thing about it is that if `open` fails, the `errno` value reported will be the one for `close` (which will get an argument of –1), and we'll have to guess why the file couldn't be created.

The output from executing the function:

13. The superuser can use it on directories on some systems, but that's nonstandard.

```
drwx------    2 marc     users          72 Oct 18 15:32 somedir
total 0
-rw-r--r--    1 marc     users           0 Oct 18 15:32 x
Expected error: Directory not empty
ls: somedir: No such file or directory
```

3.6.4 Implementing `getcwd` (Walking Up the Tree)

We really had no fun at all calling `getcwd` back in Section 3.4.2, what with all the trouble it took to size the buffer. Let's have some fun now by *implementing* it!

The basic idea is to start with the current directory, get its i-number, and then go to its parent directory to find the entry with that i-node, thereby getting the name of the child. Then we repeat, until we can go no higher.

What does "no higher" mean? It means that either

```
chdir("..")
```

returns –1 with an `ENOENT` error, or that it succeeds but leaves us in the same directory—either behavior is possible, and both mean that we're at the root.

As we walk up the tree, we'll accumulate the components of the path as we discover them in a linked list with the newest (highest-level) entry at the top. So if the name of the current directory is `grandchild`, and then we move to its parent, `child`, and then to its parent, `parent`, and then find that we're at the root, the linked list we'll end up with is shown in Figure 3.4.

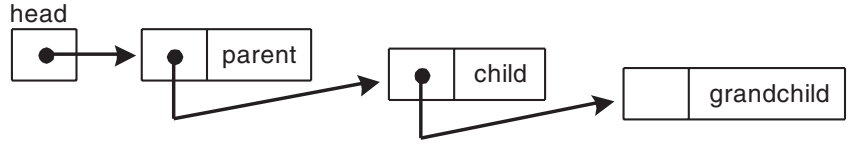


Figure 3.4 Liked list of pathname components.

Here's the structure for each linked-list node and the function that creates a new node and places it at the head of the list:

```
struct pathlist_node {
    struct pathlist_node *c_next;
    char c_name[1]; /* flexible array */
};

static bool push_pathlist(struct pathlist_node **head, const char *name)
{
    struct pathlist_node *p;

    ec_null( p = malloc(sizeof(struct pathlist_node) + strlen(name)) )
    strcpy(p->c_name, name);
    p->c_next = *head;
    *head = p;
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

The size of each `pathlist_node` is variable; the structure has enough space for the NUL byte that terminates the string, and, when allocating a node, we have to add to this space for the name, which you can see in the argument to `malloc`. Note that we put each node at the head of the list to keep the list in reverse order of when we encountered each node. That's exactly the order in which we want to assemble the components of the path, as shown by the function `get_pathlist`:

```
static char *get_pathlist(struct pathlist_node *head)
{
    struct pathlist_node *p;
    char *path;
    size_t total = 0;

    for (p = head; p != NULL; p = p->c_next)
        total += strlen(p->c_name) + 1;
    ec_null( path = malloc(total + 1) )
    path[0] = '\0';
    for (p = head; p != NULL; p = p->c_next) {
        strcat(path, "/");
        strcat(path, p->c_name);
    }
    return path;

EC_CLEANUP_BGN
    return NULL;
EC_CLEANUP_END
}
```

This function makes two passes over the list. The first just totals the space we need for the path (the +1 in the loop is for the slash), and the second builds the path string.

The final path-manipulation function frees the linked list, presumably to be called after `get_pathlist`:

```
static void free_pathlist(struct pathlist_node **head)
{
    struct pathlist_node *p, *p_next;

    for (p = *head; p != NULL; p = p_next) {
        p_next = p->c_next;
        free(p);
    }
    *head = NULL;
}
```

We used the `p_next` variable to hold the pointer to the next node because `p` itself becomes invalid as soon as we call `free`.

Here's some test code that builds the list shown in Figure 3.4:

```
struct pathlist_node *head = NULL;
char *path;

ec_false( push_pathlist(&head, "grandchild") )
ec_false( push_pathlist(&head, "child") )
ec_false( push_pathlist(&head, "parent") )
ec_null( path = get_pathlist(head) );
free_pathlist(&head);
printf("%s\n", path);
free(path);
```

And this is what got printed:

```
/parent/child/grandchild
```

A really convenient feature of handling the path this way is that we didn't have to preallocate space for the path, as we did when we called the standard `getcwd` function, in Section 3.4.2.

Now we're ready for `getcwdx`, our version of `getcwd`, which walks up the tree, calling `push_pathlist` at each level once it identifies an entry name as that of a child, and stops at the root. First comes a macro that tests whether two `stat`

structures represent the same i-node, by testing both the device IDs and i-numbers:

```
#define SAME_INODE(s1, s2)  ((s1).st_dev == (s2).st_dev &&\
                            (s1).st_ino == (s2).st_ino)
```

We'll use this macro twice in the `getcwdx` function:

```
char *getcwdx(void)
{
    struct stat stat_child, stat_parent, stat_entry;
    DIR *sp = NULL;
    struct dirent *dp;
    struct pathlist_node *head = NULL;
    int dirfd = -1, rtn;
    char *path = NULL;

    ec_neg1( dirfd = open(".", O_RDONLY) )
    ec_neg1( lstat(".", &stat_child) )
    while (true) {
        ec_neg1( lstat("..", &stat_parent) )

        /* change to parent and detect root */
        if (((rtn = chdir("..")) == -1 && errno == ENOENT) ||
          SAME_INODE(stat_child, stat_parent)) {
            if (head == NULL)
                ec_false( push_pathlist(&head, "") )
            ec_null( path = get_pathlist(head) )
            EC_CLEANUP
        }
        ec_neg1( rtn )

        /* read directory looking for child */
        ec_null( sp = opendir(".") )
        while (errno = 0, (dp = readdir(sp)) != NULL) {
            ec_neg1( lstat(dp->d_name, &stat_entry) )
            if (SAME_INODE(stat_child, stat_entry)) {
                ec_false( push_pathlist(&head, dp->d_name) )
                break;
            }
        }
        if (dp == NULL) {
            if (errno == 0)
                errno = ENOENT;
            EC_FAIL
        }
        stat_child = stat_parent;
    }
```

```
EC_CLEANUP_BGN
    if (sp != NULL)
        (void)closedir(sp);
    if (dirfd != -1) {
        (void)fchdir(dirfd);
        (void)close(dirfd);
    }
    free_pathlist(&head);
    return path;
EC_CLEANUP_END
}
```

Some comments on this function:

- We use the `open`/`fchdir` technique to get and reset the current directory, as the function changes it as it walks the tree.
- In the loop, `stat_child` is for the child whose name we're trying to get the entry for, `stat_entry` is for each entry we get from a call to `readdir`, and `stat_parent` is for the parent, which becomes the child when we move up.
- The test for arriving at the root was explained at the beginning of this section: Either `chdir` fails or it doesn't take us anywhere.
- If we get to the root with the list still empty, we push an empty name onto the stack so the pathname will come out as `/`.
- In the `readdir` loop, we could have just skipped nondirectory entries, but it really wasn't necessary, as the `SAME_INODE` test is just as fast and accurate.

Finally, here's a little program that makes this into a command that behaves like the standard `pwd` command:

```
int main(void)
{
    char *path;
    ec_null( path = getcwdx() )
    printf("%s\n", path);
    free(path);
    exit(EXIT_SUCCESS);

EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
EC_CLEANUP_END
}
```

3.6.5 Implementing `ftw` (Walking Down the Tree)

There's a standard library function, `ftw` ("file tree walk"), which I won't describe specifically here, that provides a generalized way to recursively process the

entries in a directory tree. You give it pointer to a function which it calls for each object it encounters, but it's more interesting for us to implement our own tree-walking function, using what we've learned in this chapter.

The first thing to clear up is whether the directory structure is really a tree, because our recursive algorithm depends on that. If there are any loops our program might get stuck, and if the same directory is linked to twice (by entries other than `.` and `..`), we might visit the same directory (and its subtree) more than once. There are two kinds of problems to think about:

1. It's pretty common for symbolic links to link to a directory, and that creates at least two links, since every directory is also hard linked to. There's no protection against symbolic links creating a loop, either.
2. On some systems the superuser can create a second hard link to a directory, with the `link` system call. This is almost never done, but it's a possibility our program might encounter.

Problem 1 can easily be dealt with by simply not following any symbolic links. We'll ignore Problem 2 for now, although Exercise 3.5 at the end of this chapter deals with it. So, for our purposes, the tree formed by hard links is indeed a tree.

We want our program, `aupls` (an extension of the one presented in Section 3.5.3) to act somewhat like `ls`, in that a `-R` argument makes it recurse, and a `-d` argument tells it to just list the information for a directory, rather than for that directory's contents. Without either of these arguments it behaves like the earlier `aupls`.

With the `-R` argument, `aupls` (like `ls`) first lists the path to a directory and all of the entries at that level, including directories, and then for each directory does the same thing, recursively, as in this example:

```
$ aupls -R /aup/common

/aup/common:
-rwxr-xr-x    1 root    root      1145 Oct  2 10:21 makefile
-rwxr-xr-x    1 root    root      171 Aug 23 10:41 logf.h
-rwxr-xr-x    1 root    root      1076 Aug 26 15:24 logf.c
drwxr-xr-x    1 root    root      4096 Oct  2 12:20 cf
-rwxr-xr-x    1 root    root      245 Aug 26 15:29 notes.txt

/aup/common/cf:
-rwxr-xr-x    1 root    root      1348 Oct  3 13:52 cf-ec.c-ec_push.htm
-rwxr-xr-x    1 root    root      576 Oct  3 13:52 cf-ec.c-ec_print.htm
```

```
-rwxr-xr-x    1 root    root      450 Oct  3 13:52 cf-ec.c-ec_reinit.htm
-rwxr-xr-x    1 root    root      120 Oct  3 13:52 cf-ec.c-ec_warn.htm
```

As you'll see when we get to the code, for each level (a `readdir` loop), there are two passes. In Pass 1 we print the "stat" information, and then in Pass 2, for each directory, we recurse, with a Pass 1 in that directory, and so on.

It's convenient to represent the traversal information with a structure that's passed to each function in the program, rather than using global variables or a long argument list:

```
typedef enum {SHOW_PATH, SHOW_INFO} SHOW_OP;

struct traverse_info {
    bool ti_recursive;                                  /* -R option? */
    char *ti_name;                                      /* current entry */
    struct stat ti_stat;                                /* stat for ti_name */
    bool (*ti_fcn)(struct traverse_info *, SHOW_OP); /* callback fcn */
};
```

Here's the callback function we're going to use:

```
static bool show_stat(struct traverse_info *p, SHOW_OP op)
{
    switch (op) {
    case SHOW_PATH:
        ec_false( print_cwd(false) )
        break;
    case SHOW_INFO:
        ls_long(&p->ti_stat, p->ti_name);
    }
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

`show_stat` is called with `op` set to `SHOW_PATH` to print a pathname heading, and with `SHOW_INFO` for the detail lines. `ls_long` is from Section 3.5.3, and `print_cwd` is almost identical to the sample code we showed in Section 3.4.2:

```
static bool print_cwd(bool cleanup)
{
    char *cwd;
```

```
    if (cleanup)
        (void)get_cwd(true);
    else {
        ec_null( cwd = get_cwd(false) )
        printf("\n%s:\n", cwd);
    }
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

Later we'll see a `print_cwd(true)` call in the overall cleanup code.

Now let's jump up to the top level for the `main` function to see how everything is initialized and how the listing gets started:

```
#define USAGE "Usage: aupls [-Rd] [dir]\n"

static long total_entries = 0, total_dirs = 0;

int main(int argc, char *argv[])
{
    struct traverse_info ti = {0};
    int c, status = EXIT_FAILURE;
    bool stat_only = false;

    ti.ti_fcn = show_stat;
    while ((c = getopt(argc, argv, "dR")) != -1)
        switch(c) {
        case 'd':
            stat_only = true;
            break;
        case 'R':
            ti.ti_recursive = true;
            break;
        default:
            fprintf(stderr, USAGE);
            EC_CLEANUP
        }
    switch (argc - optind) {
    case 0:
        ti.ti_name = ".";
        break;
    case 1:
        ti.ti_name = argv[optind];
        break;
```

```
    default:
        fprintf(stderr, USAGE);
        EC_CLEANUP
    }
    ec_false( do_entry(&ti, stat_only) )
    printf("\nTotal entries: %ld; directories = %ld\n", total_entries,
      total_dirs);
    status = EXIT_SUCCESS;
    EC_CLEANUP

EC_CLEANUP_BGN
    print_cwd(true);
    exit(status);
EC_CLEANUP_END
}
```

The two globals `total_entries` and `total_dirs` are used to keep overall counts, which we found interesting to display at the end; we'll see where they're incremented shortly.

`getopt` is a standard library function (part of POSIX/SUS, not Standard C). Its third argument is a list of allowable option letters. When it's finished with options, it sets the global `optind` to the index of the next argument to be processed, which in our example is an optional pathname. If none is given, we assume the current directory.

The real work is done by `do_entry`, which looks like this:

```
static bool do_entry(struct traverse_info *p, bool stat_only)
{
    bool is_dir;

    ec_neg1( lstat(p->ti_name, &p->ti_stat) )
    is_dir = S_ISDIR(p->ti_stat.st_mode);
    if (stat_only || !is_dir) {
        total_entries++;
        if (is_dir)
            total_dirs++;
        ec_false( (p->ti_fcn)(p, SHOW_INFO) )
    }
    else if (is_dir)
        ec_false( do_dir(p) )
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

`do_entry` is used in one of two ways: If the `stat_only` argument is `true` or if the current entry isn't a directory, it increments the global counters and then calls the callback function in `SHOW_INFO` mode. This is done when the `-d` argument is specified, when a nondirectory is specified on the `aupls` command line, or during Pass 1 of a directory listing. Otherwise, for a directory, it calls the function `do_dir` that contains the `readdir` loops to process a directory's entries:

```
static bool do_dir(struct traverse_info *p)
{
    DIR *sp = NULL;
    struct dirent *dp;
    int dirfd = -1;
    bool result = false;

    ec_neg1(  dirfd = open(".", O_RDONLY) )
    if ((sp = opendir(p->ti_name)) == NULL || chdir(p->ti_name) == -1) {
        if (errno == EACCES) {
            fprintf(stderr, "%s: Permission denied.\n", p->ti_name);
            result = true;
            EC_CLEANUP
        }
        EC_FAIL
    }
    if (p->ti_recursive)
        ec_false( (p->ti_fcn)(p, SHOW_PATH) )
    while (errno = 0, ((dp = readdir(sp)) != NULL)) {
        if (strcmp(dp->d_name, ".") == 0 ||
          strcmp(dp->d_name, "..") == 0)
            continue;
        p->ti_name = dp->d_name;
        ec_false( do_entry(p, true) )
    }
    if (errno != 0)
        syserr_print("Reading directory (Pass 1)");
    if (p->ti_recursive) {
        rewinddir(sp);
        while (errno = 0, ((dp = readdir(sp)) != NULL)) {
            if (strcmp(dp->d_name, ".") == 0 ||
              strcmp(dp->d_name, "..") == 0)
                continue;
            p->ti_name = dp->d_name;
            ec_false( do_entry(p, false) )
        }
        if (errno != 0)
            syserr_print("Reading directory (Pass 2)");
    }
    result = true;
    EC_CLEANUP
```

```
EC_CLEANUP_BGN
    if (dirfd != -1) {
        (void)fchdir(dirfd);
        (void)close(dirfd);
    }
    if (sp != NULL)
        (void)closedir(sp);
    return result;
EC_CLEANUP_END
}
```

`do_dir` is where the recursion takes place. It opens `dirfd` to the current directory so it can be restored on cleanup. Then it opens the directory with `opendir` and changes to it so the entries can be processed relative to their parent directory. It's very common to get an `EACCES` error because a directory isn't readable (`opendir` fails) or not searchable (`chdir` fails), so we just want to print a message in those cases—not terminate processing.

Next, if the `-R` argument was specified, we tell the callback function to print the current directory path. This is so all normal printing can be localized to one callback function.

Then we have the Pass 1 `readdir` loop (in which `do_entry` is called with `stat_only` set to true), a `rewinddir` system call, and, if `-R` was specified, the Pass 2 `readdir` loop (calling `do_entry` with `stat_only` set to `false`). The recursion occurs because a Pass 2 call to `do_entry` might call `do_dir` recursively. Note that both `readdir` loops skip the `.` and `..` entries, which `ls` also omits by default.

We decided to treat errors from `readdir` as nonfatal, to keep things going, rather than bailing out with the `ec_nzero` macro, which explains the two calls to `syserr_print`, which is:

```
void syserr_print(const char *msg)
{
    char buf[200];

    fprintf(stderr, "ERROR: %s\n", syserrmsg(buf, sizeof(buf), msg,
      errno, EC_ERRNO));
}
```

`syserrmsg` was in Section 1.4.1.

That's it—our own fully recursive version of `ls`!

3.7 Changing an I-Node

The `stat` family of system calls (Section 3.5.1) retrieves information from an i-node, and I explained there how various data fields in the i-node get changed as a file is manipulated. Some of the fields can also be changed directly by system calls that are discussed in this section. Table 3.3 indicates what system calls do what. The notation "fixed" means the field isn't changeable at all—you have to make a new i-node if you want it to be something else—and the notation "side-effect" means it's changeable only as a side-effect of doing something else, such as making a new link with `link`, but isn't changeable directly.

Table 3.3 Changing I-Node Fields

I-Node Field	Description	Changed by…
`st_dev`	device ID of file system	fixed
`st_ino`	i-number	fixed
`st_mode`	mode	`chmod, fchmod`
`st_nlink`	number of hard links	side-effect
`st_uid`	user ID	`chown, fchown, lchown`
`st_gid`	group ID	`chown, fchown, lchown`
`st_rdev`	device ID (if special file)	fixed
`st_size`	size in bytes	side-effect
`st_atime`	last access	`utime`
`st_mtime`	last data modification	`utime`
`st_ctime`	last i-node modification	side-effect
`st_blksize`	optimal I/O size	fixed
`st_blocks`	allocated 512-byte blocks	side-effect

3.7.1 chmod and fchmod System Calls

chmod—change mode of file by path

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

int chmod(
     const char *path,      /* pathname */
     mode_t mode            /* new mode */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

fchmod—change mode of file by file descriptor

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

int fchmod(
     int fd,                /* file descriptor */
     mode_t mode            /* new mode */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

The `chmod` system call changes the mode of an existing file of any type. It can't be used to change the type itself, however, so only the `S_ISUID`, `S_ISGID`, and `S_ISVTX` flags or the permission bits can be changed. Use the macros as with the `stat` structure (Section 3.5.1). `fchmod` is similar but takes an open file descriptor instead of a path.

The caller's effective user-ID must match the user-ID of the file, or the caller must be the superuser. It's not enough to have write permission or for the caller's effective group-ID to match that of the file. In addition, if `S_ISGID` is being set, the effective group-ID must match (except for the superuser).

Unless you're going to set the entire mode, you'll first have to get the existing mode with a call to one of the `stat` functions, set or clear the bits you want to change, and then execute `chmod` with the revised mode.

It's uncommon for `chmod` to be called from within an application, as the mode is usually set when a file is created. Users, however, frequently use the `chmod` command.

3.7.2 chown, fchown, and lchown System Calls

`chown` changes the user-ID and group-ID of a file. Only the owner (process's effective user-ID equal to the file's user-ID) or the superuser may execute it. If either `uid` or `gid` is –1, the corresponding ID is left alone.

chown—change owner and group of file by path

```
#include <unistd.h>

int chown(
     const char *path,      /* pathname */
     uid_t uid,             /* new user ID */
     gid_t gid              /* new group ID */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

fchown—change owner and group of file by file descriptor

```
#include <unistd.h>

int fchown(
     int fd,                /* file descriptor */
     uid_t uid,             /* new user ID */
     gid_t gid              /* new group ID */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

lchown—change owner and group of symbolic link by path

```
#include <unistd.h>

int lchown(
     const char *path,      /* pathname */
     uid_t uid,             /* user ID */
     gid_t gid              /* group ID */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

`fchown` is similar but takes an open file descriptor instead of a path. `lchown` is also similar but acts on the path argument directly, rather than what it leads to if it's a symbolic link.

Unless the caller is the superuser, these system calls clear the set-user-ID and set-group-ID bits. This is to prevent a rather obvious form of break-in:

```
$ cp /bin/sh mysh       [get a personal copy of the shell]
$ chmod 4700 mysh       [turn on the set-user-ID bit]
$ chown root mysh       [make the superuser the owner]
$ my sh                 [become superuser]
```

If you want your own superuser shell, you must reverse the order of `chmod` and `chown`; however, unless you are already the superuser, you won't be allowed to execute the `chmod`. So the loophole is plugged.

Some UNIX systems are configured to operate with a slightly different rule if the macro `_POSIX_CHOWN_RESTRICTED` is set (tested with `pathconf` or `fpathconf`).

Only the superuser can change the owner (user-ID), but the owner can change the group-ID to the process's effective group-ID or to one of its supplemental group-IDs. In other words, the owner can't give the file away—at most the group-ID can be changed to one of the group-IDs associated with the process.

As ownership is usually changed by the `chown` command (which calls the `chown` system call), these rules don't usually affect applications directly. They do affect system administration and how users are able to use the system.

3.7.3 `utime` System Call

utime—set file access and modification times

```
#include <utime.h>

int utime(
    const char *path,              /* pathname */
    const struct utimbuf *timbuf   /* new times */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

struct utimbuf—structure for utime

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

`utime` changes the access and modification times of a file of any type.[14] The type `time_t` in the structure is the number of seconds since the epoch, as defined in Section 1.7.1. Only the owner or superuser can change the times with a `timbuf` argument.

If the `timbuf` argument is `NULL`, however, the access and modification times are set to the current time. This is done to force a file to appear up-to-date without rewriting it and is primarily for the benefit of the `touch` command. Anyone with write permission can do this, since writing, which they're allowed, would also change the times.

Aside from `touch`, `utime` is most often used when restoring files from a dump tape or when receiving files across a network. The times are reset to the values they had originally, from data that comes along with the file somehow. The status-

14. On some systems there's a similar `utimes` system call, but it's obsolete.

change time can't be reset, but that's appropriate since the i-node didn't move anywhere—a new one got created.

3.8 More File-Manipulation Calls

This section covers some additional file-manipulation system calls that didn't fit in the previous sections.

3.8.1 `access` System Call

Unlike any other system call that deals with permissions, `access` checks the *real* user-ID or group-ID, not the *effective* ones.

access—determine accessibility of file

```
#include <unistd.h>

int access(
     const char *path,      /* pathname */
     int what               /* permission to be tested */
);
/* Returns 0 if allowed or -1 if not or on error (sets errno) */
```

The argument `what` uses the following flags, the first three of which can be ORed together:

```
R_OK                        /* read permission */
W_OK                        /* write permission */
X_OK                        /* execute (search) permission */
F_OK                        /* test for existence */
```

If the process's real user-ID matches that of the path, the owner permissions are checked; if the real group-ID matches, the group permissions are checked; and if neither, the other permissions are checked.

There are two principal uses of `access`:

- To check whether the real user or group has permission to do something with a file, in case the set-user-ID or set-group-ID bits are set. For example, there might be a command that sets the user-ID to superuser on execution, but it wants to check whether the real user has permission to unlink a file from a directory before doing so. It can't just test to see if `unlink` will fail

with an EACCES error because, as the effective user-ID is the superuser, it cannot fail for that reason.
- To check whether a file exists. You could use stat instead, but access is simpler. (Actually, the whole access system call could be written as a library function calling stat, and, in some implementations, it even might be implemented that way.)

If path is a symbolic link, that link is followed until a nonsymbolic link is found.[15]

When you call access, you probably don't want to put it in an ec_neg1 macro because, if it returns –1, you'll want to distinguish an EACCES error (for R_OK, W_OK, and/or X_OK) or an ENOENT error (for F_OK) from other errors. Like this:

```
if (access("tmp", F_OK) == 0)
    printf("Exists\n");
else if (errno == ENOENT)
    printf("Does not exist\n");
else
    EC_FAIL
```

3.8.2 mknod System Call

mknod—make file

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

int mknod(
    const char *path,          /* pathname */
    mode_t perms,              /* mode */
    dev_t dev                  /* device-ID */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

mknod makes a regular file, a directory, a special file, or a named pipe (FIFO). The only portable use (and the only one for which you don't need to be superuser) is to make a named pipe, but for that you have mkfifo (Section 7.2.1). You also don't need it for regular files (use open) or for directories (use mkdir) either. You can't use it for symbolic links (use symlink) or for sockets (use bind).

15. This is true of almost all system calls that take a pathname, unless they start with the letter l. Two exceptions are unlink and rename.

Therefore, the main use of `mknod` is to make the special files, usually in the /dev directory, that are used to access devices. As this is generally done only when a new device driver is installed, the `mknod` command is used, which executes the system call.

The `perms` argument uses the same bits and macros as were defined for the `stat` structure in Section 3.5.1, and `dev` is the device-ID as defined there. If you've installed a new device driver, you'll know what the device-ID is.

3.8.3 `fcntl` System Call

In Section 2.2.3, which you may want to reread before continuing, I explained that several open file *descriptors* can share the same open file *description,* which holds the file offset, the status flags (e.g., `O_APPEND`), and access modes (e.g., `O_RDONLY`). Normally, you set the status flags and access modes when a file is opened, but you can also get and set the status flags at any time with the `fcntl` system call. You can also use it to get, but not set, the access modes.

fcntl—control open file

```
#include <unistd.h>
#include <fcntl.h>

int fcntl(
     int fd,                /* file descriptor */
     int op,                /* operation */
     ...                    /* optional argument depending on op */
);
/* Returns result depending on op or -1 on error (sets errno) */
```

There are ten operations in all, but I'm only going to discuss four of them right here. The others will be discussed elsewhere, as indicated in Table 3.4.

Table 3.4 `fcntl` Operations

Operation	Purpose	Where
`F_DUPFD`	Duplicate file descriptor	Section 6.3
`F_GETFD`	Get file-descriptor flags	Here
`F_SETFD`	Set file-descriptor flags (uses third `int` argument)	Here
`F_GETFL`	Get file-description status flags and access modes	Here

Table 3.4 `fcntl` Operations (cont.)

Operation	Purpose	Where
`F_SETFL`	Set file-description status flags (uses third `int` argument)	Here
`F_GETOWN`	Used with sockets*	Section 8.7
`F_SETOWN`	Used with sockets*	Section 8.7
`F_GETLK`	Get a lock	Section 7.11.4
`F_SETLK`	Set or clear a lock	Section 7. 11.4
`F_SETLKW`	Set or clear a lock	Section 7. 11.4
* Too complicated to even summarize here.		

For all of the "get" and "set" operations, you should get the current value, set or clear the flags you want to modify, and then do the set operation, even if only one flag is defined. That way your code will still work if more flags are added later or if there are implementation-dependent, nonstandard flags that you don't know about. For example, the wrong way to set the `O_APPEND` flag is:

```
ec_neg1( fcntl(fd, F_SETFL, O_APPEND) ) /* wrong */
```

and the correct way is:

```
ec_neg1( flags = fcntl(fd, F_GETFL) )
ec_neg1( fcntl(fd, F_SETFL, flags | O_APPEND) )
```

If we wanted to clear the `O_APPEND` flag, the second line would have been:

```
    ec_neg1( fcntl(fd, F_SETFL, flags & ~O_APPEND) )
```

(The rule is "OR to set, AND complement to clear.")

The only standard file-descriptor flag that's defined is the close-on-exec flag, `FD_CLOEXEC`, which indicates whether the file descriptor will be closed when an `exec` system call is issued. There's more about close-on-exec in Section 5.3. You use the `F_GETFD` and `F_SETFD` operations for it. This is an important use for `fcntl`, as this flag can't be set when a file is opened.

You use the `F_GETFL` and `F_SETFL` operations for the file-description status flags and access modes. The access modes, which you can get but not set, are one of

O_RDONLY, O_WRONLY, or O_RDWR. As these are values, not bit-masks, the mask O_ACCMODE must be used to pick them out of the returned value, like this:

```
ec_neg1( flags = fcntl(fd, F_GETFL) )
if ((flags & O_ACCMODE) == O_RDONLY)
    /* file is opened read-only */
```

The following two `if` statements are both wrong:

```
if (flags & O_RDONLY)                /* wrong */
if ((flags & O_RDONLY) == O_RDONLY)  /* still wrong */
```

The status flags, all of which are listed in Table 2.1 in Section 2.4.4, are more interesting. You can get and set O_APPEND, O_DSYNC, O_NOCTTY, O_NONBLOCK, O_RSYNC, and O_SYNC. You can get O_CREAT, O_TRUNC, and O_EXCL, but it makes no sense to set them because they play a role only with the `open` system call (after that it's too late). I showed an example using O_APPEND previously.

There's more on the O_NONBLOCK flag in Sections 4.2.2 and 7.2.

3.9 Asynchronous I/O

This section explains how to initiate an I/O operation so that your program doesn't have to wait around for it to be completed. You can go off and do something else, and then check back later for the operation's status.

3.9.1 Synchronized vs. Synchronous, Once Again

Before reading this section, make sure you've read Section 2.16.1, where synchronized (vs. nonsynchronized) I/O is explained. Here we're concerned with asynchronous I/O, which means that the I/O operation is initiated by a system call (e.g., `aio_read`) that returns before it's completed, and its completion is dealt with separately. That is:

- *Synchronized* means that the I/O is completed when physical I/O is completed. *Nonsynchronized* means that I/O between the process and the buffer cache is good enough for completion.
- *Synchronous* means that the system call returns only when the I/O has completed, as defined by the previous paragraph. *Asynchronous* means that the system call returns as soon as the I/O has been initiated, and completion is tested for by other system calls.

To say the same thing in one sentence: Synchronous/asynchronous refers to whether the call waits for completion, and synchronized/nonsynchronized specifies what "completion" means.

Some of this section also assumes you know how to work with signals, so you may want to read Chapter 9 before reading this section, or at least refer to parts of Chapter 9 as you read, especially Section 9.5.6.

As I've noted (Section 2.9), most `write`s in UNIX are already somewhat asynchronous because the system call returns as soon as the data has been transferred to the buffer cache, and the actual output takes place later. If the `O_SYNC` or `O_DSYNC` flags (Section 2.16.3) are set, however, a `write` doesn't return until the actual output is completed. By contrast, normal `read`s usually involve waiting for the actual input because, unless there's been read-ahead or the data just happens to be in a buffer, a `read` waits until the data can be physically read.

With asynchronous I/O (AIO), a process doesn't have to wait for a `read`, or for a synchronized (`O_SYNC` or `O_DSYNC`) `write`. The I/O operation is initiated, the AIO call returns right away, and later the process can call `aio_error` to find out whether the operation has completed or even be notified when it completes via a signal or the creation of a new thread (Section 5.17).

The term "completed" as used here means just what it does with `read` and `write`. It doesn't mean that the I/O is synchronized unless the file descriptor has been set for synchronization, as explained in Section 2.16.3. Thus, there are four cases, as shown in Table 3.5.

Table 3.5 Synchronized vs. Synchronous read and write

	Synchronous	Asynchronous
Nonsynchronized	`read`/`write`; `O_SYNC` and `O_DSYNC` clear	`aio_read`/`aio_write`; `O_SYNC` and `O_DSYNC` clear
Synchronized	`read`/`write`; `O_SYNC` or `O_DSYNC` set	`aio_read`/`aio_write`; `O_SYNC` or `O_DSYNC` set

AIO can increase performance on `read`s, synchronized or not, if you can organize your application so that it can initiate `read`s before it needs the data and then go do something else useful. If it doesn't have anything else it can do in the meantime, however, it may as well just block in the `read` and let another process or

thread run. AIO also helps with synchronized `write`s but doesn't do much for nonsynchronized `write`s, as the buffer cache already does much the same thing.

Don't confuse asynchronous I/O with nonblocking I/O that you specify by setting the `O_NONBLOCK` flag (Sections 2.4.4 and 4.2.2). Nonblocking means that the call returns if it would block, but it doesn't do the work.[16] Asynchronous means that it initiates the work and then returns.

The AIO functions are part of the Asynchronous Input and Output option, as represented by the `_POSIX_ASYNCHRONOUS_IO` macro. You can test it at run-time with `pathconf`, and I showed example code that does that in Section 1.5.4.

One system call in this section, `lio_listio`, can be used for either synchronous or asynchronous I/O.

3.9.2 AIO Control Block

All of the AIO system calls use a control block to keep track of the state of an operation, and different operations have to use distinct control blocks. When an operation is complete, its control block can be reused, of course.

struct aiocb—AIO control block

```
struct aiocb {
     int aio_fildes;               /* file descriptor */
     off_t aio_offset;             /* file offset */
     volatile void *aio_buf;       /* buffer */
     size_t aio_nbytes;            /* size of transfer */
     int aio_reqprio;              /* request priority offset */
     struct sigevent aio_sigevent; /* signal information */
     int aio_lio_opcode;           /* operation to be performed */
};
```

The first four members are like the ones for `pread` and `pwrite` (Section 2.14). That is, the three arguments that `read` or `write` would take plus a file offset for an implicit `lseek`.

The `sigevent` structure is explained fully in Section 9.5.6. With it, you can arrange for a signal to be generated or a thread to be started when the operation is complete. You can pass an arbitrary integer or pointer value to the signal handler or the thread, and usually this will be a pointer to the control block. If you don't

16. `connect` is an exception; see Section 8.1.2.

want a signal or a thread, you set member `aio_sigevent.sigev_notify` to `SIGEV_NONE`.

The `aio_reqprio` member is used to affect the priority of the operation and is only available when two other POSIX options, `_POSIX_PRIORITIZED_IO` and `_POSIX_PRIORITY_SCHEDULING`, are supported. For more on this feature, see [SUS2002].

The last member, `aio_lio_opcode`, is used with the `lio_listio` system call, as explained in Section 3.9.9.

3.9.3 `aio_read` and `aio_write`

The basic AIO functions are `aio_read` and `aio_write`. Just as with `pread` and `pwrite`, the `aio_offset` member determines where in the file the I/O is to take place. It's ineffective if the file descriptor is open to a device that doesn't allow seeking (e.g., a socket) or, for `aio_write`, if the `O_APPEND` flag is set (Section 2.8). The buffer and size are in the control block instead of being passed as arguments.

aio_read—asynchronous read from file

```
#include <aio.h>

int aio_read(
    struct aiocb *aiocbp      /* control block */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

aio_write—asynchronous write to file

```
#include <aio.h>

int aio_write(
    struct aiocb *aiocbp      /* control block */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

A successful return from these functions just means that the operation was initiated; it certainly doesn't mean that the I/O was successful, or even that the values set in the control block were OK. You always have to check the result with `aio_error` (next section). An implementation might report an error such as invalid offset right away, with a –1 return from `aio_read` or `aio_write`, or it might return 0 and report the bad news later.

3.9.4 `aio_error` and `aio_return`

aio_error—retrieve error status for asynchronous I/O operation

```
#include <aio.h>

int aio_error(
    const struct aiocb *aiocbp    /* control block */
);
/* Returns 0, errno value, or EINPROGRESS (does not set errno) */
```

If the operation is complete, `aio_error` returns 0 if it was successful or the `errno` value that an equivalent `read`, `write`, `fsync`, or `fdatasync` would have returned (Sections 2.9, 2.10, and 2.16.2). If the operation isn't complete, it returns `EINPROGRESS`. If you know that the operation completed (e.g., you got a signal saying so), you can treat `EINPROGRESS` just like any other error and use the `ec_rv` macro, as we did in Section 3.6.1:

```
ec_rv( aio_error(aiocbp) )
```

I should mention, though, that the asynchronous I/O calls are usually used in fairly advanced applications for which our simple "ec" error checking may not be suitable, as I noted in Section 1.4.2. Still, `ec_rv` is fine during development because of its ability to provide a function-call trace.

If `aio_error` says the operation was successful, you still may want the return value from the equivalent `read` or `write` to get the actual number of bytes transmitted. You do that with `aio_return`:

aio_return—retrieve return status of asynchronous I/O operation

```
#include <aio.h>

ssize_t aio_return(
    struct aiocb *aiocbp          /* control block */
);
/* Returns operation return value or -1 on error (sets errno) */
```

You're supposed to call `aio_return` only if `aio_error` reported success. A –1 return from `aio_return` means that you called it wrong, not that the operation returned an error. Also, you can call `aio_return` only once per operation, as the value may be discarded once it's been retrieved.

3.9.5 aio_cancel

You can cancel an outstanding asynchronous operation with aio_cancel:

aio_cancel—cancel asynchronous I/O request

```
#include <aio.h>

int aio_cancel(
    int fd,                     /* file descriptor */
    struct aiocb *aiocbp        /* control block */
);
/* Returns result code or -1 on error (sets errno) */
```

If the aiocbp argument is NULL, this call attempts to cancel all asynchronous operations on file descriptor fd. Operations that weren't cancelled report their completion in the normal way, but those that got cancelled report (via aio_error), an error code of ECANCELED.

If the aiocbp argument isn't NULL, aio_cancel tries to cancel just the operation that was started with that control block. In this case the fd argument must be the same as the aio_fildes member of the control block.

If aio_cancel succeeds it returns one of these result codes:

AIO_CANCELED	All the requested operations were cancelled.
AIO_NOTCANCELED	One or more requested operations (maybe all) were not cancelled because they were already in progress. You have to call aio_error on each operation to find out which were cancelled.
AIO_ALLDONE	None of the requested operations were cancelled because all were completed.

3.9.6 aio_fsync

Aside from read and write equivalents, there's a third kind of I/O that can be done asynchronously: flushing of the buffer-cache, as is done with fsync or fdatasync (see Section 2.16.2 for the distinction). Both kinds of flushing are handled with the same call:

aio_fsync—initiate buffer-cache flushing for one file

```
#include <aio.h>

int aio_fsync(
     int op,                          /* O_SYNC or O_DSYNC */
     struct aiocb *aiocbp             /* control block */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

The control block is used a little differently than it is with the other calls: All buffers associated with the file descriptor given by the `aio_fildes` member are flushed, not just those associated with the operation that was started with the control block, if one even was. The request to synchronize (i.e., flush the buffers) is asynchronous: It doesn't happen right away, but is only initiated, and you can check for completion the usual way (e.g., with `aio_error` or with a signal). So, as with `aio_read` and `aio_write`, a return of zero just means that the initiation went OK.

3.9.7 `aio_suspend`

Instead of getting notified asynchronously via a signal or a thread when I/O is completed, you can decide to become synchronous by simply waiting for it to complete:

aio_suspend—wait for asynchronous I/O request

```
#include <aio.h>

int aio_suspend(
     const struct aiocb *const list[], /* array of control blocks */
     int cbcnt,                        /* number of elements in array */
     const struct timespec *timeout    /* max time to wait */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

If `timeout` is `NULL`, `aio_suspend` takes an array of `cbcnt` control blocks, and blocks until *one* of them completes. Each must have been used to initiate an asynchronous I/O operation. If at least one is already complete at the time of the call, `aio_suspend` returns immediately.

To help in reusing the same array, it's OK for an element of `list` to be `NULL`, but such an element is included in `cbcnt`.

If `timeout` isn't `NULL`, `aio_suspend` blocks at most for that amount of time, and then returns –1 with `errno` set to `EAGAIN`. The `timespec` structure is in Section 1.7.2.

Like most other system calls that block, `aio_suspend` can be interrupted by a signal, as explained further in Section 9.1.4. This leads to one very tricky situation: The completion that `aio_suspend` is waiting for could, depending on how the control block is set up, generate a signal, and that will cause `aio_suspend` to be interrupted, which causes it to return –1 with `errno` set to `EINTR`. In some cases why it returned doesn't matter, but the problem with the `EINTR` return is that any signal could have caused it, not just one from the completed I/O request. You can avoid having an asynchronous-completion signal interrupt `aio_suspend` in one of two ways:

- Use a signal to indicate completion or call `aio_suspend`, but not both.
- Set the `SA_RESTART` flag (Section 9.1.6) for the signal.

Or, let a signal interrupt `aio_suspend` and just call `aio_error` for each element of the list you pass to `aio_suspend` when it returns to see what, if anything, happened, and reissue the `aio_suspend` (perhaps in a loop) if nothing did.

3.9.8 Example Comparing Synchronous and Asynchronous I/O

This section shows an example of how to use the AIO calls and also demonstrates some of their advantages over synchronous I/O.

First, here's a program called `sio` that uses conventional, synchronous I/O to read a file. Every 8000 reads, it also reads the standard input.

```
#define PATH "/aup/c3/datafile.txt"
#define FREQ 8000

static void synchronous(void)
{
    int fd, count = 0;
    ssize_t nread;
    char buf1[512], buf2[512];

    ec_neg1( fd = open(PATH, O_RDONLY) )
    timestart();
    while (true) {
        ec_neg1( nread = read(fd, buf1, sizeof(buf1)) )
        if (nread == 0)
            break;
```

```
        if (count % FREQ == 0)
            ec_neg1( read(STDIN_FILENO, buf2, sizeof(buf2)) )
        count++;
    }
    timestop("synchronous");
    printf("read %d blocks\n", count);
    return;

EC_CLEANUP_BGN
    EC_FLUSH("synchronous")
EC_CLEANUP_END
}
```

(`timestart` and `timestop` were in Section 1.7.2.)

The standard input is connected to a pipe that's being filled from a program called `feed`:

```
$ feed | sio
```

`feed` is a very reluctant writer—it writes only once every 20 seconds:

```
int main(void)
{
    char buf[512];

    memset(buf, 'x', sizeof(buf));
    while (true) {
        sleep(20);
        write(STDOUT_FILENO, buf, sizeof(buf));
    }
}
```

With the pipeline as shown, `sio` took 0.12 sec. of user CPU time, 0.73 sec. of system CPU time, and 200.05 sec. of elapsed time to run to completion. Obviously, a lot of the elapsed time was spent waiting for input on the pipe.

This is a contrived example, but, nonetheless, it provides an opportunity to improve the elapsed time with asynchronous I/O. If reading of the pipe can be done asynchronously, the file can be read in the meantime, as in this recoding:

```
static void asynchronous(void)
{
    int fd, count = 0;
    ssize_t nread;
    char buf1[512], buf2[512];
    struct aiocb cb;
    const struct aiocb *list[1] = { &cb };
```

```
    memset(&cb, 0, sizeof(cb));
    cb.aio_fildes = STDIN_FILENO;
    cb.aio_buf = buf2;
    cb.aio_nbytes = sizeof(buf2);
    cb.aio_sigevent.sigev_notify = SIGEV_NONE;
    ec_neg1( fd = open(PATH, O_RDONLY) )
    timestart();
    while (true) {
        ec_neg1( nread = read(fd, buf1, sizeof(buf1)) )
        if (nread == 0)
            break;
        if (count % FREQ == 0) {
            if (count > 1) {
                ec_neg1( aio_suspend(list, 1, NULL) )
                ec_rv( aio_error(&cb) )
            }
            ec_neg1( aio_read(&cb) )
        }
        count++;
    }
    timestop("asynchronous");
    printf("read %d blocks\n", count);
    return;

EC_CLEANUP_BGN
    EC_FLUSH("asynchronous")
EC_CLEANUP_END
}
```

Notice how the control block is set up; it's zeroed first to make sure that any additional implementation-dependent members are zeroed. Every 8000 reads (`FREQ` was defined earlier), the program issues an `aio_read` call, and thereafter it waits in `aio_suspend` for the control block to be ready before it issues the next call. While the `aio_read` is working, it continues reading the file, not suspending again for 8000 more reads. This time the user CPU time was worse—0.18 sec.—because there was more work to do. The system time was about the same (0.70 sec.), but the elapsed time was shorter: 180.58 sec.

You won't always see a benefit this great in using AIO, and sometimes you'll see an improvement even better than this example showed. The keys are:

- The program has to have some useful work to do while the I/O is processing asynchronously.
- The benefits have to outweigh the increased bookkeeping costs and the increased number of system calls. You might also consider the increased programming complexity and the fact that AIO is not supported on all systems.

3.9.9 `lio_listio`

There's one other use for a list of control blocks: Batching I/O requests together and initiating them with a single call:

lio_listio—list-directed I/O

```
#include <aio.h>

int lio_listio(
     int mode,                        /* LIO_WAIT or LIO_NOWAIT */
     struct aiocb *const list[],      /* array of control blocks */
     int cbcnt,                       /* number of elements in array */
     struct sigevent *sig             /* NULL or signal to generate */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

Like `aio_suspend`, `lio_listio` takes a list of `cbcnt` control blocks, ignoring `NULL` elements in `list`. It initiates the requests in no particular order. Each operation is specified by the `aio_lio_opcode` member of the control block:

`LIO_READ` Same as if `aio_read` or `pread` had been called.

`LIO_WRITE` Same as if `aio_write` or `pwrite` had been called.

`LIO_NOP` A no-op; ignore the control block.

The `mode` argument determines whether the I/O initiated by `lio_listio` is synchronous or asynchronous, as shown by Table 3.6, which resembles Table 3.5 (see Section 3.9.1).

Table 3.6 Synchronized vs. Synchronous `lio_listio`

	Synchronous	Asynchronous
Nonsynchronized	`LIO_WAIT`; `O_SYNC` and `O_DSYNC` clear	`LIO_NOWAIT`; `O_SYNC` and `O_DSYNC` clear
Synchronized	`LIO_WAIT`; `O_SYNC` or `O_DSYNC` set	`LIO_NOWAIT`; `O_SYNC` or `O_DSYNC` set

So, `lio_listio` isn't only for asynchronous I/O like the "aio" system calls—it can be used any time you want to batch I/O calls.

For asynchronous `lio_listio`s, with a mode of `LIO_NOWAIT`, you can request notification via a signal or thread-startup with the `sig` argument, just as with the

aio_sigevent member of a control block. But here, the notification means that *all* the requests in the list have completed. You can still be notified of completions via the aio_sigevent members of the individual control blocks.

For LIO_WAIT, the sig argument isn't used.

As for some of the other AIO calls, a successful return from lio_listio only means that that call went well. It says nothing about the I/O operations, which you test for with aio_error.

Exercises

3.1. Modify the program in Section 3.2.2 as suggested by the last paragraph of that section.

3.2. Write the standard df command.

3.3. Modify the aupls command as suggested by the last paragraph of Section 3.6.2.

3.4. Modify the getcwdx function in Section 3.6.4 so it never changes the current directory.

3.5. Fix Problem 2 that's described at the start of Section 3.6.5.

3.6. Modify the aupls command from Section 3. 3.6.5 so it sorts the list by name.

3.7. Modify the aupls command from Section 3. 3.6.5 so it takes a -t option to sort by modified time and name.

3.8. Modify the aupls command from Section 3. 3.6.5 so it takes additional standard options (your choice).

3.9. Write a program to copy an entire tree of directories and files. It should have two arguments: the root of the tree (e.g., /usr/marc/book), and the root of the copy (e.g., /usr/marc/backup/book). Don't bother dealing with symbolic or hard links (i.e., it's OK to make multiple copies), and don't bother with preserving ownership, permissions, or times.

3.10. Same as Exercise 3.9, but, to the extent possible, preserve ownership, permissions, and times. Make "the extent possible" dependent on whether the command is running as superuser.

3.11. Same as Exercise 3.10, but preserve the symbolic and hard link structure.

3.12. Why is there no `lchmod` system call?

3.13. Write `access` as a function. You may use any system call except `access`.

3.14. Implement `readv` and `writev` (Section 2.15 using `lio_listio`. Are there any advantages to implementing them this way? Disadvantages?

3.15. Can you do 3.14 for `read`, `write`, `pread`, `pwrite`, `fsync`, `fdatasync`, `aio_read`, `aio_write`, and `aio_fsync`? If so, do it. Any advantages? Disadvantages?

3.16. List the pros and cons of checking for AIO completion by `aio_suspend`, a signal, a thread-start, or polling with `aio_error`. (Requires information that's in Chapters 5 and 9.)

This page intentionally left blank

4

Terminal I/O

4.1 Introduction

Terminal I/O is so complex that it needs most of a chapter all to itself. The problem isn't with normal terminal I/O, which I'll start with—that's even simpler than file I/O. What complicates matters are the numerous variations in terminal attributes. In this chapter I'll also explain how terminals are related to sessions and process groups. Then I'll show how to set up pseudo terminals, which allow one process to control another process that's written for a terminal.

Terminal I/O on UNIX treats a terminal like an old-fashioned hard copy Teletype, the model with the massive type box that makes a terrible racket as it bounces around, now seen only in museums or old movies. There is no specific support in the kernel for screens (character or graphical), function keys, or mice and other pointing devices. Some of these more modern devices can be handled by libraries that call on the standard device driver. The best-known such package for character displays is Curses, which is now part of [SUS2002]. So-called graphical user-interface (GUI) applications for UNIX are generally built to run on the X Window System, perhaps with one of its toolkits such as Motif, Qt, KDE, or Gnome.

Because this chapter's subject is mostly about device drivers rather than an inherent part of the kernel like the file system, specific features vary even more than usual from one UNIX version to another. Sometimes even individual UNIX sites make modifications to the terminal device driver. Remember, also, that not all attributes of terminal I/O are caused by UNIX itself. With the increasing use of smart terminals, local-area networks, and front-end processors, the character stream is subject to much processing before the UNIX kernel sees it and after it has been sent it on its way. Details of this pre- and post-processing vary much too widely to be given here. As usual, I'll concentrate on the standardized properties of terminal I/O. With this as background you should be able to figure out your own system from its manual.

4.2 Reading from a Terminal

This section explains how to read from a terminal, including how not to get blocked if the terminal has nothing ready to be read.

4.2.1 Normal Terminal I/O

I'll start by explaining how normal terminal input and output work; that is, how they work when you first log in and before you've used the `stty` command to customize them to your taste. Then, in subsequent sections, I'll show how the `fcntl` and `tcsetattr` system calls can be used to vary the normal behavior.

There are three ways to access a terminal for input or output:

1. You can open the character special file /dev/tty for reading, writing, or, most commonly, both. This special file is a synonym for the process's controlling terminal (see Section 4.3.1).
2. If you know the actual name of the special file (e.g., /dev/tty04), you can open it instead. If you just want the controlling terminal, this has no advantages over using the generic name /dev/tty. In application programs, it's mainly used to access terminals other than the controlling terminal.
3. Conventionally, each process inherits three file descriptors already open: the standard input, the standard output, and the standard error output. These may or may not be open to the controlling terminal, but normally, they should be used anyhow, since if they are open to a regular file or a pipe instead, it's because the user or parent process decided to redirect input or output.

The basic I/O system calls for terminals are `open`, `read`, `write`, and `close`. It doesn't make much sense to call `creat`, since the special file must already exist. `lseek` has no effect on terminals, as there is no file offset, and that means that `pread` and `pwrite` can't be used.

Unless you've specified the `O_NONBLOCK` flag, an attempt to open a terminal device waits until the device is connected to a terminal. This property is chiefly for the benefit of a system process that blocks in an `open` waiting for a user to connect. It then gets a return from `open` and invokes the login process so the user can log in. After login, the user's shell as specified in the password file is invoked and the user is in business.

By default, the `read` system call also acts on terminals differently from the way it acts on files: It never returns more than one line of input, and no characters are returned until the entire line is ready, even if the `read` requests only a single character. This is because until the user has ended the line, we can't assume it's in final form—the erase and kill characters can be used to revise it or to cancel it entirely. The count returned by `read` is used to determine how many characters were actually read. (What we just described is called *canonical* terminal input, but you can disable it; see Section 4.5.9.)

The user can end a line in one of two ways. Most often, a newline character is the terminator. The return key is pressed, but the device driver translates the return (octal 15) to a newline (octal 12). Alternatively, the user can generate an end-of-file (EOF) character by pressing Ctrl-d. In this case the line is made available to `read` as is, without a newline terminator. One special case is important: If the user generates an EOF at the *start* of the line, `read` will return with a zero count, since the line that the EOF terminated is empty. This looks like an end-of-file, and that's why Ctrl-d can be thought of as an end-of-file "character."

The following function can be used to read a terminal using the standard-input file descriptor `STDIN_FILENO` (defined as 0). It removes a terminating newline character if one is present, and adds a terminating null character to make the line into a C string.

```
bool getln(char *s, ssize_t max, bool *iseof)
{
    ssize_t nread;

    switch (nread = read(STDIN_FILENO, s, max - 1)) {
    case -1:
        EC_FAIL
    case 0:
        *iseof = true;
        return true;
    default:
        if (s[nread - 1] == '\n')
            nread--;
        s[nread] = '\0';
        *iseof = false;
        return true;
    }

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

The return value is used to indicate an error, so an EOF is indicated with the third argument, as in this calling example:

```
ec_false( getln(s, sizeof(s), &iseof) )
if (iseof)
    printf("EOF\n");
else
    printf("Read: %s\n", s);
```

`getln` is efficient for terminals because it reads the entire line with a single system call. Furthermore, it doesn't have to search for the end—it just goes by the count returned by `read`. But it doesn't work on files or pipes at all because, since the one-line limit doesn't apply, `read` will in general read too much. Instead of `getln` reading a line, it will read the next `max - 1` characters (assuming that many characters are present).

A more universal version of `getln` would ignore that unique property of terminals—reading one line at most. It would simply examine each character, looking for a newline:

```
bool getln2(char *s, ssize_t max, bool *iseof)
{
    ssize_t n;
    char c;

    n = 0;
    while (true)
        switch (read(STDIN_FILENO, &c, 1)) {
        case -1:
            EC_FAIL
        case 0:
            s[n] = '\0';
            *iseof = true;
            return true;
        default:
            if (c == '\n') {
                s[n] = '\0';
                *iseof = false;
                return true;
            }
            if (n >= max - 1) {
                errno = E2BIG;
                EC_FAIL
            }
            s[n++] = c;
        }
```

```
EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

This version treats a Ctrl-d typed anywhere as indicating an EOF, which is different from what `getln` did (treating only a Ctrl-d at the beginning of a line as an EOF). With anything other than a terminal, remember, there is no Ctrl-d; an end-of-file is simply the end of the file or a pipe with no open writing file descriptor.

Although `getln2` reads terminals, files, and pipes properly, it reads those sources more slowly than it might because it doesn't buffer the input as described in Section 2.12. This is easily fixed by changing `getln2` to call `Bgetc`, which is part of the `BUFIO` package introduced in that section. `Bopen` is already suitable for opening terminal special files (e.g., /dev/tty). However, to allow us to use `Bgetc` on the standard input, we need to add a function called `Bfdopen` (Exercise 4.2) that initializes a `BUFIO` pointer from an already-open file descriptor instead of from a path. Then we could read a character from the standard input, whether it's a terminal, file, or pipe, like this:

```
ec_null( stin = Bfdopen(STDIN_FILENO, "r") )
while ((c = Bgetc(stin)) != -1)
    /* process character */
```

We're now reading as fast as we can in each case: a block at a time on files and pipes, and a line at a time on terminals.

Our implementation of the `BUFIO` package doesn't allow the same `BUFIO` pointer to be used for both input and output, so if output is to be sent to the terminal, a second `BUFIO` must be opened using file descriptor `STDOUT_FILENO` (defined as 1).

The UNIX standard I/O Library provides three predefined, already-opened `FILE` pointers to access the terminal: `stdin`, `stdout`, and `stderr`, so its function `fdopen`, which is like our `Bfdopen`, usually need not be called for a terminal.

Output to a terminal is more straightforward than input, since nothing like erase and kill processing is done. As many characters as we output with `write` are immediately queued up for sending to the terminal, whether a newline is present or not.

`close` on a file descriptor open to a terminal doesn't do any more than it does for a file. It just makes the file descriptor available for reuse; however, since the file

descriptor is most often 0, 1, or 2, no obvious reuse comes readily to mind. So no one bothers to close these file descriptors at the end of a program.[1]

4.2.2 Nonblocking Input

As I said, if a line of data isn't available when `read` is issued on a terminal, `read` waits for the data before returning. Since the process can do nothing in the meantime, this is called *blocking*. No analogous situation occurs with files: either the data is available or the end-of-file has been reached. The file may be added to later by another process, but what matters is where the end is at the time the `read` is executed.

The `O_NONBLOCK` flag, set with `open` or `fcntl`, makes `read` nonblocking. If a line of data isn't available, `read` returns immediately with a –1 return and `errno` set to `EAGAIN`.[2]

Frequently we want to turn blocking on and off at will, so we'll code a function `setblock` to call `fcntl` appropriately. (The technique I'll use is identical to what I showed in Section 3.8.3 for setting the `O_APPEND` flag.)

```
bool setblock(int fd, bool block)
{
    int flags;

    ec_neg1( flags = fcntl(fd, F_GETFL) )
    if (block)
        flags &= ~O_NONBLOCK;
    else
        flags |= O_NONBLOCK;
    ec_neg1( fcntl(fd, F_SETFL, flags) )
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

Here's a test program for `setblock`. It turns off blocking and then reads lines in a loop. If there's nothing there, it sleeps for 5 seconds before continuing. I include

1. We will be closing them in Chapter 6 when we connect two processes with a pipe.
2. In some versions of UNIX, there is a similar flag called `O_NDELAY`. If set and no data is available, `read` returns with a 0, which is indistinguishable from an end-of-file return. Better to use `O_NONBLOCK`.

in the prompt the time since the loop started, using the `time` system call that I introduced in Section 1.7.1.

```
static void test_setblock(void)
{
    char s[100];
    ssize_t n;
    time_t tstart, tnow;

    ec_neg1( tstart = time(NULL) )
    ec_false( setblock(STDIN_FILENO, false) )
    while (true) {
        ec_neg1( tnow = time(NULL) )
        printf("Waiting for input (%.0f sec.) ...\n",
          difftime(tnow, tstart));
        switch(n = read(STDIN_FILENO, s, sizeof(s) - 1)) {
        case 0:
            printf("EOF\n");
            break;
        case -1:
            if (errno == EAGAIN) {
                sleep(5);
                continue;
            }
            EC_FAIL
        default:
            if (s[n - 1] == '\n')
                n--;
            s[n] = '\0';
            printf("Read \"%s\"\n", s);
            continue;
        }
        break;
    }
    return;

EC_CLEANUP_BGN
    EC_FLUSH("test_setblock")
EC_CLEANUP_END
}
```

Here's the output from one run. I waited a while before typing "hello," and then waited awhile longer before typing Ctrl-d:

```
Waiting for input (0 sec.) ...
Waiting for input (5 sec.) ...
Waiting for input (10 sec.) ...
hello
Waiting for input (15 sec.) ...
```

```
Read "hello"
Waiting for input (15 sec.) ...
Waiting for input (20 sec.) ...
Waiting for input (25 sec.) ...
^DEOF
```

The approach of sleeping for 5 seconds is a compromise between issuing `read`s so frequently that it wastes CPU time and waiting so long that we don't process the user's input right away. As it is, you can see that several seconds elapsed between when I typed "hello" and when the program finally read the input and echoed it back. Thus, generally speaking, turning off blocking and getting input in a `read`/`sleep` loop is a lousy idea. We can do much better than that, as I'll show in the next section.

As I mentioned, you can also set the `O_NONBLOCK` flag when you open a terminal special file with `open`. In this case `O_NONBLOCK` affects `open` as well as `read`: If there is no connection, `open` returns without waiting for one.

One application for nonblocking input is to monitor several terminals. The terminals might be laboratory instruments that are attached to a UNIX computer through terminal ports. Characters are sent in sporadically, and we want to accumulate them as they arrive, in whatever order they show up. Since there's no way to predict when a given terminal might transmit a character, we can't use blocking I/O, for we might wait for one terminal that has nothing to say while other talkative terminals are being ignored. With nonblocking I/O, however, we can poll each terminal in turn; if a terminal is not ready, `read` will return –1 (`errno` set to `EAGAIN`) and we can just go on. If we make a complete loop without finding any data ready, we sleep for a second before looping again so as not to hog the CPU.

This algorithm is illustrated by the function `readany`. Its first two arguments are an array `fds` of file descriptors and a count `nfds` of the file descriptors in the array. It doesn't return until a `read` of one of those file descriptors returns a character. Then it returns via the third argument (`whichp`) the subscript in `fds` of the file descriptor from which the character was read. The character itself is the value of the function; 0 means end-of-file; –1 means error. The caller of `readany` is presumably accumulating the incoming data in some useful way for later processing.[3]

3. Assume the laboratory instruments are inputting newline-terminated lines. In Section 4.5.9 we'll see how to read data without waiting for a complete line to be ready.

```
int readany(int fds[], int nfds, int *whichp)
{
    int i;
    unsigned char c;

    for (i = 0; i < nfds; i++)
        setblock(fds[i], false); /* inefficient to do this every time */
    i = 0;
    while (true) {
        if (i >= nfds) {
            sleep(1);
            i = 0;
        }
        c = 0; /* return value for EOF */
        if (read(fds[i], &c, 1) == -1) {
            if (errno == EAGAIN) {
                i++;
                continue;
            }
            EC_FAIL
        }
        *whichp = i;
        return c;
    }

EC_CLEANUP_BGN
    return -1;
EC_CLEANUP_END
}
```

The comment about the call to `setblock` being inefficient means that we really don't have to do it every time `readany` is called. It would be more efficient, if less modular, to make it the caller's responsibility.

Here's a test function for `readany`. It opens two terminals: /dev/tty is the controlling terminal (a `telnet` application running elsewhere on the network), as we explained in Section 4.2.1, and /dev/pts/3 is an `xterm` window on the screen attached directly to the computer, which is running SuSE Linux. (I discovered the name /dev/pts/3 with the `tty` command.)

```
static void readany_test(void)
{
    int fds[2] = {-1, -1}, which;
    int c;
    bool ok = false;

    ec_neg1( fds[0] = open("/dev/tty", O_RDWR) )
    ec_neg1( fds[1] = open("/dev/pts/3", O_RDWR) )
```

```
    while ((c = readany(fds, 2, &which)) > 0)
        printf("Got %c from terminal %d\n", isprint(c) ? c : '?', which);
    ec_neg1( c )
    ok = true;
    EC_CLEANUP

EC_CLEANUP_BGN
    if (fds[0] != -1)
        (void)close(fds[0]);
    if (fds[1] != -1)
        (void)close(fds[1]);
    if (!ok)
        EC_FLUSH("readany_test1")
EC_CLEANUP_END
}
```

Here's the output I got at the controlling terminal, which is where the `printf` function wrote:

```
$ readany_test
dog
Got d from terminal 0
Got o from terminal 0
Got g from terminal 0
Got ? from terminal 0
Got c from terminal 1
Got o from terminal 1
Got w from terminal 1
Got ? from terminal 1
```

Here's what I did to get this output: First, I typed "dog" followed by a return on the controlling terminal. That resulted in the four lines of output following the echo of "dog." (The question mark was for the newline.) Then, I went across the room to the Linux computer and typed "cow" followed by a return, which got me this output:

```
cow: command not found
```

Nothing showed up at the controlling terminal. The problem was that I was running a shell in the `xterm` window, and it was blocked in a `read` waiting for some input. That's normal for an abandoned terminal. I had two processes reading the same terminal, and the shell got the letters first, which, of course, it interpreted as a command. So then I tried again, this time like this:

```
$ sleep 10000
cow
```

The `sleep` command, run in the foreground, made the shell wait for a long time, and this time my input came to the `readany_test` command, which printed the last four lines of input. This illustrates that the terminal driver isn't owned by any particular process; just because one process with it opened isn't reading doesn't mean that another process can't.

One other comment on the test program: An EOF from any of the input terminals terminates it, which is how I decided to program it. You might want to do things differently in your own applications.

4.2.3 `select` System Call

Calling `sleep` to pass the time has two disadvantages: First, there could be a delay of as long as a second before an incoming character is processed, as I mentioned earlier, which might be a problem in a time-critical application. Second, we might wake up and make another futile polling loop, perhaps many times, before a character is ready. Best would be a system call that said, "Put me to sleep until an input character is ready on any file descriptor." If we had that, we wouldn't even need to make the `read`s nonblocking, since a `read` won't block if data is ready.

The system call we're looking for is named `select` (we'll get to the similar `pselect` shortly):

select—wait for I/O to be ready

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

int select(
     int nfds,                  /* highest fd + 1 */
     fd_set *readset,           /* read set or NULL */
     fd_set *writeset,          /* write set or NULL */
     fd_set *errorset,          /* error set or NULL */
     struct timeval *timeout    /* time-out (microseconds) or NULL */
);
/* Returns number of bits set or -1 on error (sets errno) */
```

pselect—wait for I/O to be ready

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

int pselect(
     int nfds,                  /* highest fd + 1 */
     fd_set *readset,           /* read set or NULL */
     fd_set *writeset,          /* write set or NULL */
     fd_set *errorset,          /* error set or NULL */
     const struct timespec *timeout, /* time-out (nanoseconds) or NULL */
     const sigset_t *sigmask    /* signal mask */
);
/* Returns number of bits set or -1 on error (sets errno) */
```

Unlike with our `readany`, where we passed in an array of file descriptors, with `select` you set a bit[4] in one of the `fd_set` arguments for each file descriptor you want to test. There are four macros for manipulating a set:

FD_ZERO—clear entire fd_set

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

void FD_ZERO(
     fd_set *fdset          /* fd_set to clear */
);
```

FD_SET—set fd_set file descriptor

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

void FD_SET(
     int fd,                /* file descriptor to set */
     fd_set *fdset          /* fd_set */
);
```

FD_CLR—clear fd_set file descriptor

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

void FD_CLR(
     int fd,                /* file descriptor to clear */
     fd_set *fdset          /* fd_set */
);
```

FD_ISSET—test fd_set file descriptor

```
#include <sys/select.h>
int FD_ISSET(
     int fd,                /* file descriptor to test */
     fd_set *fdset          /* fd_set */
);
/* Returns 1 if set or 0 if clear (no error return) */
```

You can use `select` with 0, 1, 2, or 3 sets, depending on what you're interested in waiting for (reading, writing, or an error). With all `fd_set` arguments `NULL`, `select` just blocks, until it times out or is interrupted by a signal; this isn't particularly useful.

You initialize a set with `FD_ZERO`, and then use `FD_SET` to set a bit for each file descriptor you're interested in, like this:

4. They're not required to be bits, although they usually are. Consider my use of the term in this context as just a model, not an implementation.

```
fd_set set;

FD_ZERO(&set);
FD_SET(fd1, &set);
FD_SET(fd2, &set);
```

Then you call `select`, which blocks (even if `O_NONBLOCK` is set) until one or more file descriptors are ready for reading or writing, or has an error. It modifies the sets you passed it, this time setting a bit for each file descriptor that's ready, and you have to test each file descriptor to see if it's set, like this:

```
if (FD_ISSET(fd1, &set)) {
    /* do something with fd1, depending on which fd_set */
}
```

There's no way to get the list of file descriptors; you have to ask about each one separately. You know what a file descriptor is ready for by what set you test. You can't, therefore, use the same set for more than one argument, even if the input sets are identical. You need distinct output sets so you can get the results.

A bit is set on output only if it was also set on input, even if the corresponding file descriptor was ready. That is, you only get answers for questions you asked.

On success, `select` returns the total number of bits set in all sets, a nearly useless number except as a double-check on the number of file descriptors you come up with via the calls (probably in a loop) to `FD_ISSET`.

The first argument, `nfds`, is *not* the number of bits set on input. It's the length of the sets that `select` should consider. That is, for each input set, all of the bits that are set must be in the range of file descriptors numbered 0 through `nfds - 1`, inclusive. Generally, you calculate the maximum file descriptor number and add 1. If you don't want to do that, you can use the constant `FD_SETSIZE`, which is the maximum number of file descriptors that a set can have. But, as this number is pretty large (1024, say), it's usually much more efficient to give `select` the actual number.

The last argument to `select` is a time-out interval, in terms of a `timeval` structure, which we explained in Section 1.7.1. If nothing happens during that time interval, `select` returns with a 0 value—no error and no bits set. Note that `select` might modify the structure, so reset it before calling `select` again.

If the `timeout` argument is `NULL`, there's no time-out; it's the same as an infinite time interval. If it's not `NULL` but the time is zero, `select` runs through the tests

and then returns immediately; you can use this for polling, instead of waiting. This makes sense if your application has other work to do: You can poll from time to time while the other work is in progress, since your application is using the CPU productively, and then switch to blocking (a nonzero time, or a `NULL` `timeout` argument) when the work is done and there's nothing left to do but wait.

There are three common ways that programmers use `select` incorrectly:

- Setting the first argument `nfds` to the highest-numbered file descriptor, instead of that number plus one.
- Calling `select` again with the same sets that it modified, not realizing that on return only bits representing ready file descriptors are set. You have to re-initialize the input sets for each call.
- Not realizing what "ready" means. It doesn't mean that data is ready, only that a `read` or `write` with `O_NONBLOCK` clear would not block. Returning with a zero count (EOF), an error, or with data are all possibilities. It doesn't matter whether a file descriptor has `O_NONBLOCK` set, which means that it will never block; `select` considers it ready only in the hypothetical case that it would not block with `O_NONBLOCK` clear.

Now here's a *much* better implementation of `readany`:

```
int readany2(int fds[], int nfds, int *whichp)
{
    fd_set set_read;
    int i, maxfd = 0;
    unsigned char c;

    FD_ZERO(&set_read);
    for (i = 0; i < nfds; i++) {
        FD_SET(fds[i], &set_read);
        if (fds[i] > maxfd)
            maxfd = fds[i];
    }
    ec_neg1( select(maxfd + 1, &set_read, NULL, NULL, NULL) )
    for (i = 0; i < nfds; i++) {
        if (FD_ISSET(fds[i], &set_read)) {
            c = 0; /* return value for EOF */
            ec_neg1( read(fds[i], &c, 1) )
            *whichp = i;
            return c;
        }
    }
```

```
    /* "impossible" to get here */
    errno = 0;
    EC_FAIL

EC_CLEANUP_BGN
    return -1;
EC_CLEANUP_END
}
```

This version is very efficient. The `read`s are left blocking, there's no polling loop, and no sleeping. All the wait is in `select`, and when it returns we can read the character and immediately return with it.

If there's a chance that another thread could be reading the same file descriptor as the blocking `read`, it's possible for the data that `select` said was ready to be gone before the code gets to the `read`, which might mean that it would block forever. Perhaps the easiest way to fix this is to make the `read` nonblocking and then to loop back to the `select` if it returns –1 with `errno` set to `EAGAIN`.

It's unusual to worry about a `write` to a terminal blocking. Even if it does, because the driver's buffer is full, it will probably clear itself quickly. Usually blocked writes are more critical with other outputs, such as pipes or sockets. I'll talk about them in later chapters. In particular, there's an example in Section 8.1.3 that uses `select` in the typical way it's used with a socket that accepts connections from multiple clients.

If a bit is set in a read set, you read; if one is set in a write set, you write. What do you do if one is set in the error set? It depends on what the file descriptor is open to. For sockets, the "error" might better be termed "exceptional condition," which translates to out-of-band data being ready (Section 8.7). For terminals, which is what this chapter is about, there aren't any standard exceptional or error conditions, although it's possible for an implementation to have something nonstandard, for which you'll have to consult the documentation for that system.

There's a fancier variant of select called `pselect`; the differences between it and `select` are:

- For `pselect`, the time-out is a `timespec` structure, measured in nanoseconds, and it's not modified.
- There's a sixth argument, a signal mask (Section 9.1.5) to be set during the call to `pselect`, with the old mask restored when it returns. For reasons that are explained in Chapter 9, you should use `pselect` if you're expecting it to be interrupted by a signal, rather than `select`. (If the `sigmask` argument is

NULL, select and pselect behave identically as far as signals are concerned.)

pselect is new with SUS Version 3, so it's just now beginning to appear, and your system probably doesn't have it.

4.2.4 poll System Call

poll—wait for I/O to be ready

```
#include <poll.h>

int poll(
    struct pollfd fdinfo[], /* info on file descriptors to be tested */
    nfds_t nfds,            /* number of elements in fdinfo array */
    int timeout             /* time-out (milliseconds) */
);
/* Returns number of ready file descriptors or -1 on error (sets errno) */
```

struct pollfd—structure for poll

```
struct pollfd {
    int fd;                 /* file descriptor */
    short events;           /* event flags (see table, below) */
    short revents;          /* returned event flags (see table, below) */
};
```

poll was originally designed to be used with the STREAMS I/O facility in AT&T System V (Section 4.9), but, like select, it can be used with file descriptors open to any type of file. Despite its name, poll, like select, can be used either for waiting or for polling. Most systems have poll, but Darwin 6.6 is an exception.

Whereas with select you set up bit masks, with poll you set up pollfd structures, one for each file descriptor you want to know about. For each file descriptor, you set flags in the events field for each event (e.g., reading, writing) to be monitored. When poll returns, each structure's revents field has bits set to indicate which events occurred. So, unlike with select, the inputs aren't disturbed by the call, and you can reuse them.

If you're trying to test a file descriptor with a huge value, poll may be more efficient than select. To see why, suppose you want to test file descriptor 1000. select would have to test all bits 0 through 1000, whereas you could build an array for poll with only one element. Another problem with select is that the sets are sized according to the number of potential file descriptors, and some ker-

nels are configured to allow really huge numbers. Bit masks with 10,000 bits are very inefficient.

The `timeout` argument is in milliseconds, rather than being a `timeval` or `timespec` structure, as with `select` and `pselect`. If the `timeout` is –1, `poll` blocks until at least one of the requested events occurs on some listed file descriptor. If it's 0, as with `select` and `pselect`, `poll` just tests the file descriptors and returns, possibly with a 0 return value, which indicates that no event has occurred. If `timeout` is positive, `poll` blocks for at most that amount of time. (So, as with `select` and `pselect`, the 0 and positive cases are really the same.)

Table 4.1 lists the event flags for `poll`. To make sense of them, you need to know that `poll` distinguishes between "normal" data, "priority" data, and "high-priority" data. The meaning of those terms varies with what the file descriptor is open to.

Table 4.1 Event Flags for `poll` System Call

Flag	Meaning
`POLLRDNORM`	Normal data ready to be read
`POLLRDBAND`	Priority data ready to be read
`POLLIN*`	Same as `POLLRDNORM` | `POLLRDBAND`
`POLLPRI*`	High-priority data ready to be read
`POLLWRNORM`	Normal data read to be written
`POLLOUT*`	Same as `POLLWRNORM`
`POLLWRBAND`	Priority data may be written
`POLLERR*`	I/O error has occurred; set only in `revents` (i.e., not on input)
`POLLHUP*`	Device disconnected (no longer writable, but may be readable); set only in `revents`
`POLLNVAL*`	Invalid file descriptor; set only in `revents`

* Most common flags for non-STREAMS; `POLLPRI` is generally only used with sockets.

Except in unusual situations, you can consider `POLLIN | POLLPRI` to be equivalent to `select`'s read and `POLLOUT | POLLWRBAND` to be equivalent to `select`'s write. Note that with `poll` you don't have to ask specifically to be informed about exceptional conditions—the last three flags in the table will be set if they apply no matter what the input flags were.

If you have a `pollfd` array already set up and you just want to disable checking for a file descriptor, you can set its `fd` field to –1.

Here's a version of `readany` (from Section 4.2.2) using `poll`. (Section 4.2.3 had a version using `select`.)

```
#define MAXFDS 100

int readany3(int fds[], int nfds, int *whichp)
{
    struct pollfd fdinfo[MAXFDS] = { { 0 } };
    int i;
    unsigned char c;

    if (nfds > MAXFDS) {
        errno = E2BIG;
        EC_FAIL
    }
    for (i = 0; i < nfds; i++) {
        fdinfo[i].fd = fds[i];
        fdinfo[i].events = POLLIN | POLLPRI;
    }
    ec_neg1( poll(fdinfo, nfds, -1) )
    for (i = 0; i < nfds; i++) {
        if (fdinfo[i].revents & (POLLIN | POLLPRI)) {
            c = 0; /* return value for EOF */
            ec_neg1( read(fdinfo[i].fd, &c, 1) )
            *whichp = i;
            return c;
        }
    }
    /* "impossible" to get here */
    errno = 0;
    EC_FAIL

EC_CLEANUP_BGN
    return -1;
EC_CLEANUP_END
}
```

Comments on `readany3`:

- It's a good idea to initialize structures to all zeros, which is what's going on in the declaration for the `fdinfo` array, in case the implementation has defined additional fields that we don't know about. Also, should we ever have to use a debugger on the structure, zeros instead of garbage for the fields we're not initializing explicitly (e.g., `revents`) will make the display easier to read.
- Our program is limited to only 100 file descriptors. Changing it to allocate the array dynamically would be an easy improvement.
- Strictly speaking, the test on `revents` assumes that the flags use distinct bits, which doesn't seem to be stated in the SUS, but it's a safe assumption for `poll`.
- We didn't check the `POLLERR`, `POLLHUP`, or `POLLNVAL` flags, but we probably should have.
- In your own application, you probably don't want to set up the array of `pollfd` structures for every call, as the file descriptors of interest don't usually change that often.
- The blocking `read` could be a problem if another thread is reading the same file descriptor. The solution in the `select` example (Section 4.2.3) could be used here as well.

4.2.5 Testing and Reading a Single Input

In Section 4.2.2 we motivated the use of `select` and `poll` with an example where there were multiple inputs. Sometimes, though, there's just one, but you want to know if a character is ready, without reading it; you want to separate the test from the reading. You can use `select` or `poll` for that by setting the timeout to zero. But, if there's only one input, it's just as easy to do it with `read` alone, with `O_NONBLOCK` set. We'll do it with two functions: `cready` tells whether a character is ready, and `cget` reads it.

A glitch with `cready` is that if a character is ready, `read` will read it, but we wanted `cget` to read it. The simple solution is to keep the prematurely read character in a buffer. Then `cget` can look in the buffer first. If the character is there, we return it; if not, we turn on blocking and call `read`. That scheme is used in the following implementation of `cready` and `cget`:

```
#define EMPTY '\0'
static unsigned char cbuf = EMPTY;
typedef enum {CR_READY, CR_NOTREADY, CR_EOF} CR_STATUS;
```

```
bool cready(CR_STATUS *statusp)
{
    if (cbuf != EMPTY) {
        *statusp = CR_READY;
        return true;
    }
    setblock(STDIN_FILENO, false);
    switch (read(STDIN_FILENO, &cbuf, 1)) {
    case -1:
        if (errno == EAGAIN) {
            *statusp = CR_NOTREADY;
            return true;
        }
        EC_FAIL
    case 0:
        *statusp = CR_EOF;
        return true;
    case 1:
        return true;
    default: /* "impossible" case */
        errno = 0;
        EC_FAIL
    }

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}

bool cget(CR_STATUS *statusp, int *cp)
{
    if (cbuf != EMPTY) {
        *cp = cbuf;
        cbuf = EMPTY;
        *statusp = CR_READY;
        return true;
    }
    setblock(0, true);
    switch (read(STDIN_FILENO, cp, 1)) {
    case -1:
        EC_FAIL
    case 0:
        *cp = 0;
        *statusp = CR_EOF;
        return true;
    case 1:
        *statusp = CR_READY;
        return true;
```

```
    default: /* "impossible" case */
        errno = 0;
        EC_FAIL
    }

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

Both functions return `true` on success and `false` on error. The `statusp` argument gives the result for `cready`: `CR_READY`, `CR_NOTREADY`, or `CR_EOF`. For `cget`, the only possibilities are `CR_READY` and `CR_EOF`.

We've preempted the NUL byte as our empty-buffer indicator (`EMPTY`), which means that NUL bytes read by `cready` will be ignored. If this isn't acceptable, a separate `bool` variable can be used as an empty-buffer flag.

Notice the way `cready` and `cget` shift back and forth between blocking and nonblocking input. Since we're reading only one input stream, there's no need to stay in the nonblocking state while we test for an available character, as we had to do in the first, inefficient, version of `readany` in Section 4.2.2. We revert to blocking and issue a `read`. After all, waiting for a character is what blocking means.

`cready` and `cget` are most useful when a program has some discretionary work to do, work that can be postponed if a character is ready. A good example is a program that uses Curses (Section 4.8). The way Curses works, all the screen output is held in a buffer until `refresh` is called to send it to the physical screen. This is time-consuming because `refresh` compares what is currently on the screen with the new image so as to minimize the number of characters transmitted. A fast typist can easily get ahead of the updates, especially if the screen must be updated on each keystroke, as it must with a screen editor. A neat solution is to call `refresh` from within the input routine, but only if input is not waiting. If input is waiting, then the user has obviously typed ahead without waiting for the screen to catch up, so the screen update is skipped. The screen will be updated the next time the input routine is executed, unless a character is waiting then too. When the user stops typing, the screen will become current. On the other hand, if the program can process characters faster than the typist types them, the screen will get updated with each keystroke.

This may sound complex, but by using functions we've already developed, it takes only a few lines of code:

```
ec_false( cready(&status) )
if (status == CR_NOTREADY)
    refresh();
ec_false( cget(&status, &c) )
```

4.3 Sessions and Process Groups (Jobs)

This section explains sessions and process groups (also called jobs), which are mainly for use by shells.

4.3.1 Terminology

When a user logs in, a new *session* is created, which consists of a new *process group,* comprising a single process that is running the login shell. That process is the *process-group leader,* and its process ID (73056, say) is the *process-group ID*. That process is also the *session leader,* and the process ID is also the *session ID*.[5] The terminal from which the user logged in becomes the *controlling terminal* of the session. The session leader is also the *controlling process*. This arrangement is shown in the right part of Figure 4.1, with the process group marked FG, for foreground.

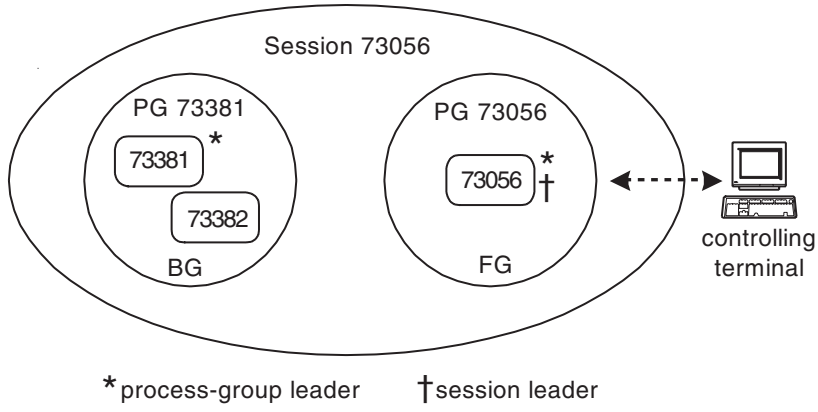


Figure 4.1 Session, process groups, processes, and controlling terminal.

5. The various standards use the phrase "process-group ID of the session leader," but I prefer to just call it the session ID.

If *job control* is enabled by the shell and a command or pipeline of commands is then run in the background, like this:

```
$ du -a | grep tmp >out.tmp&
```

a new process group is formed, this time with two processes in it, as shown at the left of Figure 4.1, with the process group marked BG, for background. One of the two new processes is the leader of this new process group.

Both process groups (73056 and 73381) are in the same session and have the same controlling terminal. With job control, a new process group is created for each command line, and each such process group is also called a *job*.

Just because a process group runs in the background, it doesn't mean its standard file descriptors aren't still directed to and from the terminal. In the previous example pipeline, `du`'s standard output is connected via a pipe to the standard input of `grep`, and `grep`'s standard output is redirected to a file, but that's because the user specifically set them up that way, not because they're running in the background.

What job control does mean is that if certain signals are generated from the controlling terminal, such as interrupt (`SIGINT`), quit (`SIGQUIT`), or suspend (`SIGTSTP`), they are only sent to the foreground process group, leaving any background process groups alone. The "control" part means that the user can execute shell commands to move process groups (jobs) back and forth between foreground and background. Usually, if a process group is running the foreground, a Ctrl-z sends it a `SIGTSTP` signal, which causes the shell to move it to the background, bringing one of the background process groups to the foreground. Or the `fg` shell command can bring a specific background process group to the foreground, which also then sends the old foreground process group to the background.

If job control is not enabled, the two new processes in the example pipeline (running `du` and `grep`) don't run in their own process group, but run in process group 73056, along with the first process, the shell, as shown in Figure 4.2. They still run in the background, but all that means is that the shell doesn't wait for them to complete. As there is only one process group, there is no way to move process groups between foreground and background, and any signals generated from the controlling terminal go to all processes in the only process group in the session. Tying Ctrl-c at the terminal will send a `SIGINT` to all three processes, including the two background processes, and, unless those processes have set themselves up to catch or ignore the signal, it will terminate them, which is the default action for this particular signal. (Much more about signals in Chapter 9.)

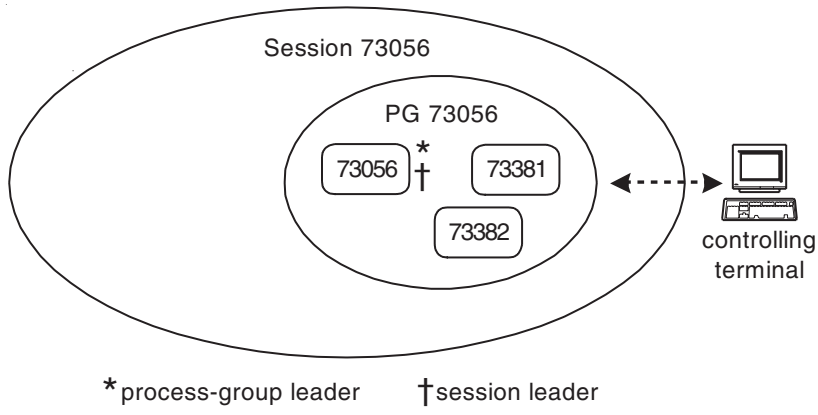


Figure 4.2 No job control; only one process group per session.

With or without job control, when the controlling terminal hangs up or is disconnected, a `SIGHUP` signal is sent to the controlling process, and the default for that signal is to terminate it. For a typical session, this terminates the login shell and logs out the user. The controlling terminal is no longer the controlling terminal for the session, which then has no controlling terminal. Whether other processes in the session can continue to access the terminal is not specified by any of the standards, and most implementations will probably prevent such access by, for example, returning an error from a `read` or `write`.

However, once the controlling process terminates for *any* reason (not just receiving a `SIGHUP`), every process in the foreground (or only) process group gets a `SIGHUP`, which by default causes them to terminate. Unless they've arranged to catch or ignore this signal, they never get far enough to attempt to access the former controlling terminal.

What happens if a process in a background process group attempts to access the controlling terminal? Regardless of whether the controlling process is running, the basic rule is that any attempt by a background process to read the controlling terminal causes it to be sent a `SIGTTIN` signal, and any attempt to write it generates a `SIGTTOU` signal. Here "write" also means using the terminal-control system calls `tcsetattr`, `tcdrain`, `tcflow`, `tcflush`, and `tcsendbreak` (Sections 4.5 and 4.6).

The default behavior for these signals is to suspend the process, which makes a lot of sense: Background processes that need access to the controlling terminal stop, and you move them to the foreground when you're ready to interact with them.

The exceptions to the basic rule are:

- A background process attempting to read with `SIGTTIN` signals ignored or blocked instead gets an `EIO` error from `read`.
- An orphaned background process (process-group leader has terminated) also gets an `EIO` from `read`.
- If the `TOSTOP` terminal attribute (see Section 4.5.6) is clear, the process is allowed to write.
- If `TOSTOP` is set but `SIGTTOU` signals are ignored or blocked, the process is allowed to write.
- If `TOSTOP` is set, an orphaned background process gets an `EIO` error from `write`.

So, to summarize the rules: A background process can't read the controlling terminal no matter what; it gets an error or is stopped. An orphaned background process gets an error if it tries to write the terminal. Nonorphaned background processes can write it if `TOSTOP` is clear or if `SIGTTOU`s are ignored or blocked; otherwise they are stopped.

The asymmetry between reading and writing is because it never makes sense for two processes to both read characters—it's much too dangerous. (Imagine typing `rm *.o` and having the shell only get `rm *`.) Two processes writing is only confusing, and if that's what the user really wants, then so be it.

4.3.2 System Calls for Sessions

setsid—create session and process group

```
#include <unistd.h>

pid_t setsid(void);
/* Returns process-group ID or -1 on error (sets errno) */
```

getsid—get session ID

```
#include <unistd.h>

pid_t getsid(
    pid_t pid          /* process ID or 0 for calling process */
);
/* Returns session ID or -1 on error (sets errno) */
```

I said that each login started in a new session, but where do sessions come from? Actually, any process that isn't already a session leader can call `setsid` to become the leader of a new session, and the process-group leader of the only process group in that session. Importantly, the new session has no controlling terminal. This is great for daemons, which don't need one, and also for sessions that want to establish a different controlling terminal from the one established at login time. The first terminal device opened by the new session becomes its controlling terminal.

4.3.3 System Calls for Process Groups

setpgid—set or create process-group

```
#include <unistd.h>

int setpgid(
    pid_t pid,              /* process ID or 0 for calling process */
    pid_t pgid              /* process-group ID */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

getpgid—get process-group ID

```
#include <unistd.h>

pid_t getpgid(
    pid_t pid               /* process ID or 0 for calling process */
);
/* Returns process-group ID or -1 on error (sets errno) */
```

With a call to `setpgid`, a process that isn't already a process-group leader can be made the leader of a new process group. This occurs if the two arguments are equal and no process-group with that ID exists. Or, if the second argument specifies an existing process-group ID, the process-group indicated by the `pid` argument is changed. However, there are a bunch of restrictions:

- `pid` must be the calling process or a child of the calling process that has not yet done an "exec" system call (explained in Section 5.3).
- `pid` must be in the same session as the calling process.
- If `pgid` exists, it must be in the same session as the calling process.

Practically speaking, these restrictions don't amount to much. Typically, a shell creates child processes for each command in a pipeline, chooses one as the process-group leader, creates the new process-group with a call to `setpgid`, and then puts the other commands in the pipeline into that group with addi-

tional calls to `setpgid`. Once this is all set up, process-group assignments don't change, although it's theoretically possible.

Two older calls set and get process-group IDs, `setpgrp` and `getpgrp`, but they're less functional than `setpgid` and `getpgid` and are obsolete.

4.3.4 System Calls for Controlling Terminals

tcsetpgrp—set foreground process-group ID

```
#include <unistd.h>

int tcsetpgrp(
     int fd,             /* file descriptor */
     pid_t pgid          /* process-group ID */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

tcgetpgrp—get foreground process-group ID

```
#include <unistd.h>

pid_t tcgetpgrp(
     int fd              /* file descriptor */
);
/* Returns process-group ID or -1 on error (sets errno) */
```

tcgetsid—get session ID

```
#include <termios.h>

pid_t tcgetsid(
     int fd              /* file descriptor */
);
/* Returns session ID or -1 on error (sets errno) */
```

The `tcsetpgrp` system brings a process-group to the foreground, which means that it receives signals generated from the controlling terminal. Whatever used to be in the foreground is moved to the background. The process group must be in the same session as the calling process.

The shell `fg` command uses `tcsetpgrp` to bring the requested process to the foreground. Typing Ctrl-z doesn't directly move the foreground process to the background; what actually happens is that it sends a `SIGTSTP` signal to the process, which stops it, and then its parent, the shell, gets a return from the `waitpid` system call that tells it what happened. The shell then executes `tcsetpgrp` to move itself to the foreground.

`tcgetsid` is only on SUS systems, so it isn't available on FreeBSD. You can get the session ID from a file descriptor open to the controlling terminal, however, first by calling `tcgetpgrp` to get the foreground process group ID, and then, since that must be the process ID of a process in the session, you can call `getsid` to get the session ID.

4.3.5 Using the Session-Related System Calls

Here's a function that prints lots of session- and process-group-related information using the previous system calls:

```
#include <termios.h>

static void showpginfo(const char *msg)
{
    int fd;

    printf("%s\n", msg);
    printf("\tprocess ID = %ld; parent = %ld\n",
      (long)getpid(), (long)getppid());
    printf("\tsession ID = %ld; process-group ID = %ld\n",
      (long)getsid(0), (long)getpgid(0));
    ec_neg1( fd = open("/dev/tty", O_RDWR) )
    printf("\tcontrolling terminal's foreground process-group ID = %ld\n",
      (long)tcgetpgrp(fd));
#if _XOPEN_VERSION >= 4
    printf("\tcontrolling-terminal's session ID = %ld\n",
      (long)tcgetsid(fd));
#else
    printf("\tcontrolling-terminal's session ID = %ld\n",
      (long)getsid(tcgetpgrp(fd)));
#endif
    ec_neg1( close(fd) )
    return;

EC_CLEANUP_BGN
    EC_FLUSH("showpginfo")
EC_CLEANUP_END
}
```

We want to call `showpginfo` when the process starts up and when it gets a `SIGCONT` signal so we can see what's going on when it's running in the background. I'm not going to tell much about signal catching until Chapter 9, and all you need to know for now is that the initializing of the structure and the call of `sigaction` arranges for the function `catchsig` to be executed when a

`SIGCONT` signal arrives.[6] After that, the main program sleeps. If it returns from `sleep` because a signal arrived, it goes back to sleeping again.

```
int main(void)
{
    struct sigaction act;

    memset(&act, 0, sizeof(act));
    act.sa_handler = catchsig;
    ec_neg1( sigaction(SIGCONT, &act, NULL) )
    showpginfo("initial call");
    while (true)
        sleep(10000);

EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
EC_CLEANUP_END
}

static void catchsig(int signo)
{
    if (signo == SIGCONT)
        showpginfo("got SIGCONT");
}
```

Interacting with `pginfo` is illuminating:

```
$ echo $$
5140
$ pginfo
initial call
        process ID = 6262; parent = 5140
        session ID = 5140; process-group ID = 6262
        controlling terminal's foreground process-group ID = 6262
        controlling-terminal's session ID = 5140

^Z[1]+  Stopped                 pginfo
$ bg %1
[1]+ pginfo &
got SIGCONT
        process ID = 6262; parent = 5140
        session ID = 5140; process-group ID = 6262
        controlling terminal's foreground process-group ID = 5140
        controlling-terminal's session ID = 5140
```

6. In Section 9.1.7 I'm going to say that there are restrictions on what system calls can be used in a signal handler, and, therefore, some of what `showpginfo` does is technically illegal. We're only in Chapter 4, though, and what we don't know won't hurt us too badly to get through this example.

First, we see that the shell's process ID is 5140. Initially, `pginfo` is running in the foreground. It's in its own process group, whose ID is 6262, which is the same as `pginfo`'s process ID. This means it's the process-group leader. The session ID is the same as the shell's process ID, which means that `pginfo` is in the same session as the shell. Then when the user typed Ctrl-z the shell got the status of its child, `pginfo`, reported as stopped, and brought itself to the foreground, allowing us to type another command to the shell, `bg`, which restarted `pginfo` in the background. That sent a `SIGCONT` signal to `pginfo` which caused it to print the information again. It's much the same, except this time the foreground process group is 5140, that of the shell.

Most of the system calls related to sessions and process groups are used by shells and not by other application programs. However, one of them, `setsid`, is very important to applications, as we'll see in Section 4.10.1, when we use it to switch a process's controlling terminal to a pseudo terminal.

4.4 `ioctl` System Call

Recall that there are two kinds of UNIX devices: block and character. Block devices were dealt with in Chapter 3, and one important kind of character device, that for terminals, is what we've been talking about in this chapter. There's a general-purpose system call for controlling character devices of all types called `ioctl`:

ioctl—control character device

```
#include <...>

int ioctl(
     int fd,                /* file descriptor */
     int req,               /* request */
     ...                    /* arguments that depend on request */
);
/* Returns -1 on error (sets errno); some other value on success */
```

With two exceptions, because the POSIX and SUS standards don't specify devices, the include file, the various requests, and the associated third arguments used with `ioctl` are implementation dependent, and you generally find the details in the documentation for the driver you're trying to control.

The two exceptions are:

- Almost everything you can do to a terminal with `ioctl` also has its own standardized function, primarily so that the types of the arguments can be checked at compile time. These functions are `tcgetattr`, `tcsetattr`, `tcdrain`, `tcflow`, `tcflush`, and `tcsendbreak`, and they're described in the next section.
- The SUS does specify how `ioctl` is to be used with STREAMS, which I touch on briefly in Section 4.9.

Outside of Section 4.9, I won't discuss `ioctl` any further in this book.

4.5 Setting Terminal Attributes

The two important system calls for controlling terminals are `tcgetattr`, which gets the current attributes, and `tcsetattr`, which sets new attributes. Four other functions, `tcdrain`, `tcflow`, `tcflush`, and `tcsendbreak`, are explained in Section 4.6.

4.5.1 Basic `tcgetattr` and `tcsetattr` Usage

tcgetattr—get terminal attributes

```
#include <termios.h>

int tcgetattr(
    int fd,                 /* file descriptor */
    struct termios *tp      /* attributes */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

tcsetattr—set terminal attributes

```
#include <termios.h>

int tcsetattr(
    int fd,                 /* file descriptor */
    int actions,            /* actions on setting */
    const struct termios *tp  /* attributes */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

struct termios—structure for terminal-control functions

```
struct termios {
    tcflag_t c_iflag;       /* input flags */
    tcflag_t c_oflag;       /* output flags */
    tcflag_t c_cflag;       /* control flags */
    tcflag_t c_lflag;       /* local flags */
    cc_t c_cc[NCCS];        /* control characters */
};
```

The terminal-information structure (`termios`) contains about 50 flag bits that tell the driver how to process characters coming in and going out, how to set communication-line parameters, such as the baud rate, and so on. The structure also defines several control characters, such as erase (normally Del or backspace), kill (normally Ctrl-u), and EOF (normally Ctrl-d).

Before you can change attributes, you need to call `tcgetattr` to initialize the structure. It's complicated and possibly loaded with implementation-defined flags, so it's impractical to just initialize a structure from scratch. After you get the structure, you adjust the flags and other fields as you like and then call `tcsetattr` to effect the changes. Its second argument, `action`, controls how and when the changes occur; one of these symbols is used:

`TCSANOW` Set the terminal immediately, according to the information in the structure.

`TCSADRAIN` Similar to `TCSANOW`, but wait for all pending output characters to be sent first. This should be used when output-affecting changes are made, to ensure characters previously output are processed under the rules in effect when they were written.

`TCSAFLUSH` Similar to `TCSADRAIN` but in addition to waiting for pending output to be drained, the input queue is flushed (characters are discarded). When beginning a new interactive mode, as when starting a screen editor, this is the safest command to use because it prevents characters that may have been typed ahead from causing an unintended action.

The following subsections describe most of the commonly used flags and how they're generally combined for typical uses (e.g., "raw" mode). For a complete list of standardized flags, see [SUS2002]. Or, you can execute `man termios` or `man termio` to see your implementation's flags.

4.5.2 Character Size and Parity

Flags in `c_cflag` represent the character size (`CS7` for 7 bits; `CS8` for 8), the number of stop bits (`CSTOPB` for 2, clear for 1), whether parity should be checked (`PARENB` if so), and the parity (`PARODD` for odd, clear for even). There's

no creativity here; one is usually quite satisfied to find a combination that works.

You can also get more information about characters that fail the parity check. If the `PARMRK` flag of `c_iflag` is on, bad characters are preceded with the two characters 0377 and 0. Otherwise, characters with incorrect parity are input as NUL bytes. Alternatively, flag `IGNPAR` of `c_iflag` can be set to ignore characters with bad parity entirely.

4.5.3 Speed

Nowadays most terminals run pretty fast, but there still is a list of symbols (not integers) defined for various standard speeds, including some very low ones. Their names are of the form `B`n, where n is `50`, `75`, `110`, `134`, `150`, `200`, `300`, `600`, `1200`, `1800`, `2400`, `4800`, `9600`, `19200`, or `38400`. The units are bits-per-second, which is called "baud" in UNIX documentation and standards.

Some implementations encode the speed in the `c_cflag` field, but for portability you use four standard functions to get or set the input and output speeds, rather than manipulating the `termios` structure directly. These functions only manipulate the structure—you have to first populate the structure with a call to `tcgetattr`, and then you have to call `tcsetattr` to make a speed change effective. Also, keep in mind that the only standardized values for a `speed_t` are the `B`n symbols. You can't portably pass in a plain integer, although some implementations might allow that.

cfgetispeed—get input speed from termios structure

```
#include <termios.h>

speed_t cfgetispeed(
    const struct termios *tp  /* attributes */
);
/* Returns speed (no error return) */
```

cfgetospeed—get output speed from termios structure

```
#include <termios.h>

speed_t cfgetospeed(
    const struct termios *tp  /* attributes */
);
/* Returns speed (no error return) */
```

cfsetispeed—set input speed in termios structure

```
#include <termios.h>

int cfsetispeed(
     struct termios *tp,        /* attributes */
     speed_t speed              /* speed */
);
/* Returns 0 on success or -1 on error (may set errno) */
```

cfsetospeed—set output speed in termios structure

```
#include <termios.h>

int cfsetospeed(
     struct termios *tp,        /* attributes */
     speed_t speed              /* speed */
);
/* Returns 0 on success or -1 on error (may set errno) */
```

The speed `B0` is special: If it's set as the output speed, the call to `tcsetattr` will disconnect the terminal. If it's set as the input speed, it means that the output and input speeds are the same.

4.5.4 Character Mapping

The mapping of newlines to returns and returns to newlines on input is controlled by flags `INLCR` and `ICRNL` in `c_oflag`; normally the first is clear and the second is set. For terminals that input both a return and a newline when the return key is pressed, the return can be ignored (`IGNCR`).

On output we usually want to map a newline to a return-newline pair; flag `ONLCR` in `c_oflag` does this job. Other flags cause a return to be changed to a newline (`OCRNL`), and a return at column 0 to be suppressed (`ONOCR`). If a newline also causes a return, there's a flag (`ONLRET`) to tell the terminal driver this so it can keep track of the column (for tabs and backspaces).

The driver can also handle uppercase-only terminals, which are much less common than they once were.[7] If `XCASE` in `c_lflag` is set, `IUCLC` in `c_iflag` is set, and `OLCUC` in `c_oflag` is set, then uppercase letters are mapped to lowercase on input, and lowercase letters are mapped to uppercase on output. Since UNIX uses lowercase more than upper, this is desirable. To input or output uppercase, the letter is preceded with a backslash (\).

7. The words "much less common" appeared in the 1985 edition of this book. Now they probably don't even exist outside of museums.

If the `ISTRIP` flag in `c_iflag` is set, input characters are stripped to seven bits (after parity is checked). Otherwise, all eight bits are input. Some terminals use ASCII, which is a seven-bit code, so stripping is normally desirable on those. Modern devices and programs such as `xterm`, `telnet`, or `ssh` may, however, transmit a full eight bits. On output, all bits written by the process are sent. If eight data bits are to be sent to the terminal, then parity generation must be turned off by clearing the `PARENB` flag in `c_cflag`.

4.5.5 Delays and Tabs

Terminals used to be mechanical and lacked big buffers so they required time to perform various motions, such as to return the carriage. Flags in `c_oflag` can be set to adjust delays for newlines, returns, backspaces, horizontal and vertical tabs, and form-feeds, but they're unlikely to be needed any more.

Another flag, `TAB3`, causes output tabs to be replaced by an appropriate number of spaces. This is useful when the terminal has no tabs of its own, when it's too much trouble to set them, or when the terminal is really another computer that is downloading the output and only spaces are wanted.

4.5.6 Flow Control

Output to the terminal can be stopped momentarily either by the user pressing Ctrl-s or by a process calling `tcflow` (see Section 4.6). Flow is restarted by the user typing Ctrl-q or by `tcflow`.

If the `IXANY` flag in `c_iflag` is set, the user can type any character to restart the flow, not just Ctrl-q. If the `IXON` flag is clear, no output flow control is available to the user at all; Ctrl-s and Ctrl-q have no special meaning.

The terminal driver also supports input flow control. If the `IXOFF` flag is set, then, when the input queue gets full, the driver will send a Ctrl-s to the terminal to suspend input. When the queue length drops because a process read some queued-up characters, a Ctrl-q will be sent to tell the terminal to resume input. Of course, this feature can be used only with those terminals that support it.

If the `TOSTOP` flag in `c_lflag` is set, a `SIGTTOU` signal is sent if a process in a background process group attempts to write to the controlling terminal, as I explained in Section 4.3.1.

4.5.7 Control Characters

Several control characters can be changed from their defaults by setting elements of the `c_cc` array in the `termios` structure. Table 4.2 gives, for each settable control character, the subscript in `c_cc` and the default value. In the array a character is represented by its internal value.

The ASCII control characters are as follows: Ctrl-a through Ctrl-z have the values 1 through 26; Ctrl-[, Ctrl-\, Ctrl-], Ctrl-^, and Ctrl-_ have values 27 through 31; Ctrl-? (Del) has value 127; Ctrl-@ has value 0. Values 32 through 126 are the 95 printable ASCII characters and aren't useful for control characters. Values above 127 cannot be generated from a standard US English keyboard, but may be available on other keyboards. There's no standard UNIX way to use function keys that generate multiple-character sequences.

Table 4.2 `c_cc` Subscripts

Subscript	Meaning	Typical Default
`VEOF`	end-of-file	Ctrl-d
`VEOL`	alternative end-of-line (rarely used)	undefined
`VERASE`	erase character	Ctrl-?[†]
`VINTR`	interrupt; generates `SIGINT`	Ctrl-c
`VKILL`	kill line	Ctrl-u
`VQUIT`	quit; generates `SIGQUIT`	Ctrl-\
`VSUSP`	stop process; generates `SIGTSTP`[*]	Ctrl-z
`VSTART`	resume input or output	Ctrl-q
`VSTOP`	suspend input or output	Ctrl-s

[*] Switching the names `VSUSP` and `VSTOP` might make sense, but the table is correct.
[†] Because some terminals generate Ctrl-? (ASCII DEL) when the backspace key is pressed, not Ctrl-h (ASCII BS).

Most implementations define additional characters for such things as word-erase, reprint, and discard output; the only standardized ones are those in the table, however. The total number of elements in the array is `NCCS`.

The characters specified by the c_cc subscripts VINTR, VQUIT, and VSUSP generate signals, as I said in Section 4.3. By default, SIGINT terminates a process, SIGQUIT terminates it with a core dump, and SIGTSTP causes it to stop until a SIGCONT signal is received. There's lots more on signals in Chapter 9.

To suppress a control character, you can set it to _POSIX_VDISABLE. Alternatively, you can suppress interrupt, quit, and stop (c_cc[VSUSP]) by clearing the ISIG flag in c_lflag, which disables signal generation. If _POSIX_VDISABLE is not in the header unistd.h, you get its value from pathconf or fpathconf (Section 1.5.6). It's usually defined as zero.

4.5.8 Echo

Terminals normally run in full duplex, which means that data can flow across the communication line in both directions simultaneously. Consequently, the computer, not the terminal, echoes typed characters to provide verification that they were correctly received. The usual output character mapping applies, so, for example, a return is mapped to a newline and then, when echoed, the newline is mapped to both a return and a newline.

To turn echo off, the ECHO flag in c_lflag is cleared. This is done either to preserve secrecy, as when typing a password, or because the process itself must decide if, what, and where to echo, as when running a screen editor.

Two special kinds of echo are available for the erase and kill characters. The flag ECHOE can be set so erase character echoes as backspace-space-backspace. This has the pleasing effect of clearing the erased character from a CRT screen (it has no effect at all on hard-copy terminals, except to wiggle the type element). The flag ECHOK can be set to echo a newline (possibly mapped to return and newline) after a kill character; this gives the user a fresh line to work on.

4.5.9 Punctual vs. Canonical Input

Normally, input characters are queued until a line is complete, as indicated by a newline or an EOF. Only then are any characters made available to a read, which might only ask for and get one character. In many applications, such as screen editors and form-entry systems, the reading process wants the characters as they are typed, without waiting for a line to be assembled. Indeed, the notion of "lines" may have no meaning.

If the flag ICANON ("canonical") in c_lflag is clear, input characters are not assembled into lines before they are read; therefore, erase and kill editing are unavailable. The erase and kill characters lose their special meaning. Two parameters, MIN and TIME, determine when a read is satisfied. When the queue becomes MIN characters long, or when TIME tenths of a second have elapsed after a byte has been received, the characters in the queue become available. TIME uses an inter-byte timer that's reset when a byte is received, and it doesn't start until the first byte is received so you won't get a timeout with no bytes received. (An exception when MIN is zero follows.)

The subscripts VMIN and VTIME of the c_cc array hold MIN and TIME. As those positions may be the same ones used by VEOF and VEOL, make sure you set MIN and TIME explicitly or you might get whatever the current EOF and EOL characters work out to, which will cause very strange results. If you ever find a process getting input on every fourth character typed, this is probably what's happened. (Ctrl-d, the usual EOF character, is 4.)

The idea behind MIN and TIME is to allow a process to get characters as, or soon after, they are typed without losing the benefits of reading several characters with a single read system call. Unless you code your input routine to buffer input, however, you might as well set MIN to 1 (TIME is then irrelevant, as long as it's greater than zero), since your read system calls will be reading only a single character anyhow.

What I just described applies to MIN and TIME both greater than zero. If either or both are zero, it's one of these special cases:

- If TIME is zero, the timer is turned off, and MIN applies, assuming it's greater than zero.
- If MIN is zero, TIME is no longer treated as an inter-byte timer, and timing starts as soon as the read is issued. It returns when one byte is read or TIME expires, whichever comes first. So, in this case, read could return with a count of zero.
- In both are zero, read returns when the minimum of the number of bytes requested (by read's third argument) and the number available in the input queue. If no bytes are available, read returns with a count of zero, so this acts like a nonblocking read.

The standards don't specify exactly what happens if O_NONBLOCK is set and MIN and/or TIME are nonzero. O_NONBLOCK may have precedence (causing an

immediate return if no characters are available), or MIN/TIME may have precedence. To avoid this uncertainty, you should not set `O_NONBLOCK` when `ICANON` is clear.

Here's a buffering, but punctual, input routine called `tc_keystroke`. Since it will change the terminal flags, we've also provided the companion routine `tc_restore` to restore the terminal to the way it was before the first call to `tc_keystroke`. The calling program must call `tc_restore` before terminating.

```
static struct termios tbufsave;
static bool have_attr = false;

int tc_keystroke(void)
{
    static unsigned char buf[10];
    static ssize_t total = 0, next = 0;
    static bool first = true;
    struct termios tbuf;

    if (first) {
        first = false;
        ec_neg1( tcgetattr(STDIN_FILENO, &tbuf) )
        have_attr = true;
        tbufsave = tbuf;
        tbuf.c_lflag &= ~ICANON;
        tbuf.c_cc[VMIN] = sizeof(buf);
        tbuf.c_cc[VTIME] = 2;
        ec_neg1( tcsetattr(STDIN_FILENO, TCSAFLUSH, &tbuf) )
    }
    if (next >= total)
        switch (total = read(0, buf, sizeof(buf))) {
        case -1:
            syserr("read");
        case 0:
            fprintf(stderr, "Mysterious EOF\n");
            exit(EXIT_FAILURE);
        default:
            next = 0;
        }
    return buf[next++];

EC_CLEANUP_BGN
    return -1;
EC_CLEANUP_END
}
```

```
bool tc_restore(void)
{
    if (have_attr)
        ec_neg1( tcsetattr(STDIN_FILENO, TCSAFLUSH, &tbufsave) )
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

Note that we've cleared only `ICANON` (and set MIN and TIME, of course). Typically, we would want to clear more flags, to turn off echo, output character mapping, and so on. This is addressed in the next section. In a specific application, you will want to experiment with values for MIN and TIME. We've used 10 characters and .2 seconds, which seems to work well.

Here's some test code that calls `tc_keystroke` in a loop:

```
setbuf(stdout, NULL);
while (true) {
    c = tc_keystroke();
    putchar(c);
    if (c == 5) { /* ^E */
        ec_false( tc_restore() )
        printf("\n%s\n", "Exiting...");
        exit(EXIT_SUCCESS);
    }
}
```

The important thing about this test code is that characters are echoed by the terminal driver when they're typed and then again by the `putchar` function when they're returned by `tc_keystroke`. This allows us to see the effect of MIN and TIME, as in the following example output where the letters in "slow" were typed very slowly and the letters in "fast" were typed very fast. The typed letters are underlined.

```
sslloowwfastfast^E
Exiting...
```

Thus, MIN and TIME cause no change in how `tc_keystoke` operates, in that it's still punctual (within .2 seconds, with our setting of TIME) yet cuts down on `read`s by a factor of 10 (or whatever we set MIN to).

4.5.10 Raw Terminal I/O

Punctual, or noncanonical, input, as described in the previous section, is useful, but sometimes we want echo turned off as well, along with most everything else.

That's normally called *raw* mode, which means that no special input or output processing is done and that characters are readable immediately, without waiting for a line to be assembled. There's no single flag to set raw mode; instead you have to set the various attributes one-by-one. (The `stty` *command* does have a `raw` option.)

We want raw terminal I/O to have the following attributes, although there's no hard-and-fast definition, and you may want to vary this a bit for your own purposes:

1. *Punctual input.* Clear `ICANON` and set MIN and TIME.
2. *No character mapping.* Clear `OPOST` to turn off output processing. For input, clear `INLCR` and `ICRNL`. Set the character size to `CS8`. Clear `ISTRIP` to get all eight bits and clear `INPCK` and `PARENB` to turn off parity checking. Clear `IEXTEN` to turn off extended-character processing.
3. *No flow control.* Clear `IXON`.
4. *No control characters.* Clear `BRKINT` and `ISIG` and set all the control characters to the disabled value, even those that are implementation defined.
5. *No echo.* Clear `ECHO`.

These operations are encapsulated in the function `tc_setraw`. Note that it saves the old `termios` structure for use by `tc_restore` (previous section).

```
bool tc_setraw(void)
{
    struct termios tbuf;
    long disable;
    int i;

#ifdef _POSIX_VDISABLE
    disable = _POSIX_VDISABLE;
#else
    /* treat undefined as error with errno = 0 */
    ec_neg1( (errno = 0, disable = fpathconf(STDIN_FILENO, _PC_VDISABLE)) )
#endif
    ec_neg1( tcgetattr(STDIN_FILENO, &tbuf) )
    have_attr = true;
    tbufsave = tbuf;
    tbuf.c_cflag &= ~(CSIZE | PARENB);
    tbuf.c_cflag |= CS8;
    tbuf.c_iflag &= ~(INLCR | ICRNL | ISTRIP | INPCK | IXON | BRKINT);
    tbuf.c_oflag &= ~OPOST;
```

```
    tbuf.c_lflag &= ~(ICANON | ISIG | IEXTEN | ECHO);
    for (i = 0; i < NCCS; i++)
        tbuf.c_cc[i] = (cc_t)disable;
    tbuf.c_cc[VMIN] = 5;
    tbuf.c_cc[VTIME] = 2;
    ec_neg1( tcsetattr(STDIN_FILENO, TCSAFLUSH, &tbuf) )
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

Here's a test program for `tc_setraw` that uses the `stty` command to display the terminal settings:

```
setbuf(stdout, NULL);
printf("Initial attributes:\n");
system("stty | fold -s -w 60");
printf("\r\nRaw attributes:\n");
tc_setraw();
system("stty | fold -s -w 60");
tc_restore();
printf("\r\nRestored attributes:\n");
system("stty | fold -s -w 60");
```

What follows is the output I got (on Solaris). Note that `tc_setraw` turned off all the `c_cc` characters, not just the standard ones that we know about. But, it didn't turn off all the flags that might make the terminal nonraw, such as `imaxbel`.[8] You may find that you have to adjust your version of `tc_setraw` for each system you port your code to, or else use the Curses library (Section 4.8), which has its own way of setting the terminal to raw.

```
Initial attributes:
speed 38400 baud; evenp
rows = 40; columns = 110; ypixels = 0; xpixels = 0;
swtch = <undef>;
brkint -inpck icrnl -ixany imaxbel onlcr
echo echoe echok echoctl echoke iexten

Raw attributes:
speed 38400 baud; -parity
rows = 40; columns = 110; ypixels = 0; xpixels = 0;
min = 5; time = 2;
intr = <undef>; quit = <undef>; erase = <undef>; kill =
<undef>; eof = ^e; eol = ^b; swtch = <undef>; start =
```

8. A nonstandard flag that controls ringing of the bell when an input line is too long.

```
<undef>; stop = <undef>; susp = <undef>; dsusp = <undef>;
rprnt = <undef>; flush = <undef>; werase = <undef>; lnext =
<undef>;
-inpck -istrip -ixon imaxbel -opost
-isig -icanon -echo echoe echok echoctl echoke

Restored attributes:
speed 38400 baud; evenp
rows = 40; columns = 110; ypixels = 0; xpixels = 0;
swtch = <undef>;
brkint -inpck icrnl -ixany imaxbel onlcr
echo echoe echok echoctl echoke iexten
```

Sometimes during (or after!) debugging, a program that's put the terminal in raw mode aborts before it can restore the original settings. Users sometimes think the computer has crashed or that their terminal has "locked up"—they can't even use EOF to log off. But it's possible to recover from raw mode. First, recall that `ICRNL` is clear; this means you'll have to end your input lines with Ctrl-j instead of using the return key.[9] Second, you won't see what you type because `ECHO` is off, too. Start by typing a few line feeds; you should see a series of shell prompts but not at the left margin because output processing (`OPOST`) is turned off. Then type `stty sane`, followed by a line feed, and all should be well.

4.6 Additional Terminal-Control System Calls

As we saw in Section 4.5.1, when you set terminal attributes with `tcsetattr`, you can drain (wait for) the output queue before the attributes are set with the `TCSADRAIN` action, and you can in addition flush (throw away) any characters in the input queue with the `TCSAFLUSH` action. Two system calls allow you to control draining and flushing separately, without also setting attributes:

tcdrain—drain (wait for) terminal output

```
#include <termios.h>

int tcdrain(
    int fd              /* file descriptor */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

9. A nice enhancement to the shell would be for it to accept a return as a line terminator as well as a newline. (Footnote first appeared in 1985—no progress so far.)

tcflush—flush (throw away) terminal output, input, or both

```
#include <termios.h>

int tcflush(
    int fd,             /* file descriptor */
    int queue           /* queue to be affected */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

These functions act on the queue—that is, characters received from the terminal but not yet read by any process, or characters written by a process but not yet transmitted to the terminal.

The second argument (`queue`) for `tcflush` is one of:

`TCIFLUSH` Flush the input queue.

`TCOFLUSH` Flush the output queue.

`TCIOFLUSH` Flush both queues.

The `TCSADRAIN` argument of `tcsetattr` is therefore equivalent to calling `tcdrain` before the attributes have been set, and the `TCSAFLUSH` argument of `tcsetattr` is equivalent to calling both `tcdrain` and `tcflush` with a queue argument of `TCIFLUSH`.

The next system call, `tcflow`, allows an application to suspend or restart terminal input or output:

tcflow—suspend or restart flow of terminal input or output

```
#include <termios.h>

int tcflow(
    int fd,             /* file descriptor */
    int action          /* direction and suspend/restart */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

The `action` argument is one of:

`TCOOFF` Suspend output.

`TCOON` Restart suspended output.

`TCIOFF` Send a STOP character, intended to make the terminal suspend input.

`TCION` Send a START character, intended to make the terminal restart suspended input.

`TCOOF` could be used by an application that's writing multiple pages to the terminal to suspend output after each page so the user can read it before continuing by typing the START character from the keyboard, which is Ctrl-q by default (Section 4.5.7). After calling `tcflow` with `TCOOF`, the application can keep writing pages of output; when the terminal's queue is full, the next `write` will block until it gets drained a bit.

However, this approach is not what users expect, and it's not how page-at-a-time applications typically work. Commands like `more` and `man` output a page and then prompt the user to type an ordinary character (e.g., space or return) to get the next page. Internally, such applications output one page, output the prompt, and block on a `read` (typically with canonical input and echo turned off), waiting for the user to type the go-ahead character.

On the input side, `TCIOFF` and `TCION` had a historical use in preventing a terminal from overflowing the input queue, but nowadays this is handled entirely by the driver and/or the hardware. Today `tcflow` would probably be used only for specialized devices, not actual terminals.

In the same category would be `tcsendbreak`, which sends a break to a terminal:

tcsendbreak—send break to terminal

```
#include <termios.h>

int tcsendbreak(
    int fd,             /* file descriptor */
    int duration        /* duration of break */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

A "break" is a stream of 0 bits sent to a terminal for a period of time, often as a way of sending the terminal into a special "attention" mode.

If the `duration` argument is 0, the period is between a quarter and half second. If it's nonzero, what it means is implementation dependent. Don't worry about portability, though—you'll probably never have occasion to send a break to a terminal.

4.7 Terminal-Identification System Calls

As I said in Section 4.2.1, the generic pathname /dev/tty provides a way to access a process's controlling terminal without knowing its name. That wasn't always a requirement, however, and instead there was a system call for getting such a synonym:

ctermid—get pathname for controlling terminal

```
#include <stdio.h>

char *ctermid(
     char *buf          /* buffer of size L_ctermid or NULL */
);
/* Returns pathname or empty string on error (errno not defined) */
```

You can call `ctermid` with a `NULL` argument, in which case it uses a static buffer of the appropriate size. Or, you can supply a buffer of size `L_ctermid` (which includes room for the NUL byte) to avoid any conflicts in case there are multiple threads calling the function. Note that on error `ctermid` returns the empty string, not `NULL`. `ctermid` isn't required to return anything more useful than the string `/dev/tty`, and on most (if not all) systems that's all it does.

Somewhat more useful would be a system call that returned an actual terminal name, and that's what `ttyname` and `ttyname_r` are for, only you have to give them an open file descriptor as input:

ttyname—find pathname of terminal

```
#include <unistd.h>

char *ttyname(
     int fd             /* file descriptor */
);
/* Returns string or NULL on error (sets errno) */
```

ttyname_r—find pathname of terminal

```
#include <unistd.h>

int ttyname_r(
     int fd,            /* file descriptor */
     char *buf,         /* buffer for pathname */
     size_t bufsize     /* size of buffer */
);
/* Returns 0 on success or error number on error (errno not set) */
```

As we've seen with other functions, the `_r` suffix means that `ttyname_r` is reentrant—it uses no static storage but rather the buffer you pass in. And, as we've seen with other functions, it's a pain in the neck to size that buffer: You can use the macro `TTY_NAME_MAX` if it's defined (in limits.h), and if it isn't you have to call `sysconf` (Section 1.5.5) with the argument `_SC_TTY_NAME_MAX`. Or, you can just call `sysconf` all the time. If you know that only one thread is calling `ttyname`, it's the easier of the two to use by far.

Here's my version of the `tty` command, which prints the name of the terminal connected to the standard input or "not a tty" if it's not a terminal:

```
int main(void)
{
    char *ctty;

    if ((ctty = ttyname(STDIN_FILENO)) == NULL) {
        printf("not a tty\n");
        exit(1);
    }
    printf("%s\n", ctty);
    exit(0);
}
```

Here I used the numbers 1 and 0 instead of the symbols `EXIT_FAILURE` and `EXIT_SUCCESS` because the SUS explicitly says that the return code shall be 1 or 0. (POSIX specifies only that `EXIT_FAILURE` be nonzero, not that it be 1.)

If you really want the name of the controlling terminal, even if the standard input is somehow opened to a different terminal or a nonterminal input, you might think that you could open /dev/tty and pass that file descriptor to `ttyname`. But, alas, it doesn't work that way on the systems where I tried it—`ttyname` just returns /dev/tty instead of the actual name. In fact, there seems to be no portable way to get the name of the controlling terminal, although if the standard input is a terminal, it's probably the controlling terminal.

You can test whether a file descriptor is opened to a terminal with the `isatty` system call:

isatty—test for terminal

```
#include <unistd.h>

int isatty(
    int fd                  /* file descriptor */
);
/* Returns 1 if a terminal and 0 if not (may set errno on 0 return) */
```

You can't depend on `errno` being set if `isatty` returns 0, but in most cases you don't care about the reason if it returns anything other than 1.

4.8 Full-Screen Applications

Traditional UNIX commands are sometimes interactive (e.g., `dc`, `more`), but they still read and write text lines. There are at least three other interesting categories of applications that more fully exploit the capabilities of display screens and their associated input devices (e.g., mice):

- Character-oriented full-screen applications that run on cheap terminals (hardly made anymore) and terminal emulators such as `telnet` and `xterm`. The best-known such application is probably the `vi` text editor.
- Graphical-User-Interface (GUI) applications written for the X Window System, perhaps using a higher-level toolkit such as Motif, Gnome, KDE, or even Tcl/Tk. (There are also some other GUI systems besides X.)
- Web applications, in which the user interface is written with technologies such as HTML, JavaScript, and Java.

Only the first group makes use of the terminal functionality that's the subject of this chapter. X applications talk to the so-called X server over a network, and the GUI processing actually occurs in the server, which may or may not be running on a UNIX-based computer. If it is, it talks directly to the display and input devices via their device drivers, not through the character-terminal driver. Similarly, Web applications communicate with the browser over a network (Section 8.4.3), and the browser handles the user interaction. If the browser is running on UNIX, it's often an X application.

I'm going to present an example of a very simple character-oriented full-screen application that clears the screen and puts up a menu that looks like this:

```
What do you want to do?
1. Check out tape/DVD
2. Reserve tape/DVD
3. Register new member
4. Search for title/actor
5. Quit

(Type item number to continue)
```

Figure 4.3 Menu.

The number that the user types isn't echoed and doesn't have to be followed by a return. In my little example, all you get if you type, say, 3, is the screen in Figure 4.4.

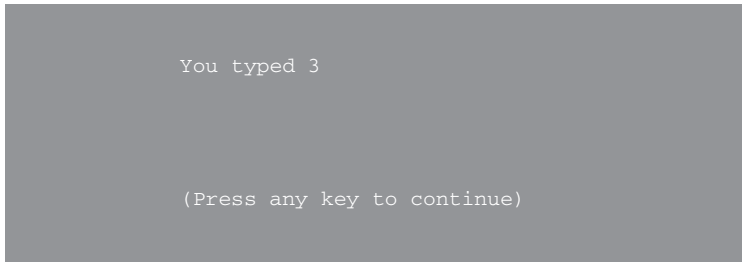


Figure 4.4 Response.

I'll show two implementations of this application. The first is just for ANSI terminals and works fine on anything that can emulate a VT100 terminal. ANSI terminals (and VT100s) have dozens of control sequences, of which we're only going to use two:

- The sequence **ε[r;cH** positions the cursor to row r, column c. (ε is the escape character.)
- The sequence **ε[2J** clears the screen.

Two functions use these control sequences; note that `clear` also moves the cursor to the upper-left of the screen:

```
#define ESC "\033"

bool mvaddstr(int y, int x, const char *str)
{
    return printf(ESC "[%d;%dH%s", y, x, str) >= 0;
}

bool clear(void)
{
    return printf(ESC "[2J") >= 0 &&
      mvaddstr(0, 0, "");
}
```

We want to ring the terminal's bell if the user types a wrong character, and we can use an ASCII code for that:

```
#define BEL "\007"

int beep(void)
{
    return printf(BEL) >= 0;
}
```

Assuming the terminal is in raw mode from a call to `tc_setraw`, we call `getch` to read a character:

```
int getch(void)
{
    char c;

    switch(read(STDIN_FILENO, &c, 1)) {
    default:
        errno = 0;
        /* fall through */
    case -1:
        return -1;
    case 1:
        break;
    }
    return c;
}
```

Those are all the user-interface service routines we need. The whole application is in the `main` function. Obviously, if it did anything it would be much longer.

```
int main(void)
{
    int c;
    char s[100];
    bool ok = false;

    ec_false( tc_setraw() )
    setbuf(stdout, NULL);
    while (true) {
        ec_false( clear() )
        ec_false( mvaddstr(2, 9, "What do you want to do?") )
        ec_false( mvaddstr(3, 9, "1. Check out tape/DVD") )
        ec_false( mvaddstr(4, 9, "2. Reserve tape/DVD") )
        ec_false( mvaddstr(5, 9, "3. Register new member") )
        ec_false( mvaddstr(6, 9, "4. Search for title/actor") )
        ec_false( mvaddstr(7, 9, "5. Quit") )
        ec_false( mvaddstr(9, 9, "(Type item number to continue)") )
        ec_neg1( c = getch() )
```

```
        switch (c) {
        case '1':
        case '2':
        case '3':
        case '4':
            ec_false( clear() )
            snprintf(s, sizeof(s), "You typed %c", c);
            ec_false( mvaddstr(4, 9, s) )
            ec_false( mvaddstr(9, 9, "(Press any key to continue)") )
            ec_neg1( getch() )
            break;
        case '5':
            ok = true;
            EC_CLEANUP
        default:
            ec_false( beep() )
        }
    }

EC_CLEANUP_BGN
    (void)tc_restore();
    (void)clear();
    exit(ok ? EXIT_SUCCESS : EXIT_FAILURE);
EC_CLEANUP_END
}
```

You probably don't want to code for just one kind of terminal, no matter how ubiquitous. Instead, you'll want to use a standardized library called Curses, which was introduced in Section 4.1. It keeps a database of the control-sequences for every terminal ever made so it can switch to the appropriate control sequence at run-time.

I cleverly named the functions `mvaddstr`, `clear`, `beep`, and `getch` after Curses functions of the same names, which do the same thing. So all we have to show for the Curses version is the `main` function:

```
#include <curses.h>

/* "ec" macro for ERR (used by Curses) */
#define ec_ERR(x) ec_cmp(x, ERR)

int main(void)
{
    int c;
    char s[100];
    bool ok = false;
```

```
    (void)initscr();
    ec_ERR( raw() )
    while (true) {
        ec_ERR( clear() )
        ec_ERR( mvaddstr( 2, 9, "What do you want to do?") )
        ec_ERR( mvaddstr( 3, 9, "1. Check out tape/DVD") )
        ec_ERR( mvaddstr( 4, 9, "2. Reserve tape/DVD") )
        ec_ERR( mvaddstr( 5, 9, "3. Register new member") )
        ec_ERR( mvaddstr( 6, 9, "4. Search for title/actor") )
        ec_ERR( mvaddstr( 7, 9, "5. Quit") )
        ec_ERR( mvaddstr( 9, 9, "(Type item number to continue)") )
        ec_ERR( c = getch() )
        switch (c) {
        case '1':
        case '2':
        case '3':
        case '4':
            ec_ERR( clear() )
            snprintf(s, sizeof(s), "You typed %c", c);
            ec_ERR( mvaddstr( 4, 9, s) )
            ec_ERR( mvaddstr( 9, 9, "(Press any key to continue)") )
            ec_ERR( getch() )
            break;
        case '5':
            ok = true;
            EC_CLEANUP
        default:
            ec_ERR( beep() )
        }
    }

EC_CLEANUP_BGN
    (void)clear();
    (void)refresh();
    (void)endwin();
    exit(ok ? EXIT_SUCCESS : EXIT_FAILURE);
EC_CLEANUP_END
}
```

Some comments on this program:

- Most Curses functions return `ERR` on an error, so I defined an `ec_ERR` macro just for them. The documentation for Curses doesn't say that it uses `errno`, but I left it alone, knowing that if I do get a message its value may be meaningless.
- Curses requires that you begin with `initscr` and end with `endwin`.
- The Curses version of `tc_setraw` is `raw`.

I won't cover any more of Curses in this book, as it's not a kernel service.[10] Curses is part of SUS2 and SUS3, so you can read the specification online at the SUS2 site at *www.unix-systems.org/version2/online.html*. On most systems you can also type `man curses` or `man ncurses` (a free version of Curses).

4.9 STREAMS I/O

STREAMS are a mechanism on some implementations of UNIX that allow character device drivers to be implemented in a modular fashion. Applications can, to some extent, chain various STREAMS modules together to get the kind of driver capabilities they want. It's somewhat analogous to the way UNIX filters work at the shell level; for example, the `who` command may not have an option to sort its output, but you can pipe it into `sort` if that's what you want.

I'll show an example of STREAMS-based pseudo terminals in the next section. What I'll do there is open a device file to get a pseudo terminal and then use `ioctl` to "push" two STREAMS modules onto it (`ldterm` and `ptem`) to make it behave like a terminal. Unlike with shell pipelines, where you have dozens of filters to choose from that can be combined hundreds of creative ways, STREAMS modules are usually used in cookbook fashion—you push the modules the documentation tells you to push, and, as with cooking, you won't get anything delicious if you just push things willy-nilly. (STREAMS should not be confused with the standard I/O concept of streams, used by `fopen`, `fwrite`, etc.. The two are totally unrelated.)

The STREAMS feature was required by SUS1 and SUS2, but not by SUS3. Linux, which sets `_XOPEN_VERSION` to 500 (indicating it's SUS2), doesn't provide STREAMS, which makes it nonconforming in that sense.

Aside from my use of STREAMS for pseudo terminals in the next section and a few other mentions in this book, I won't provide a full treatment of STREAMS. The easiest way to find out more about them is from the SUS or from Sun's site at *www.sun.com,* where you can find the *STREAMS Programming Guide*.

10. Besides, if we included Curses, we'd have to include X, and that would kill us. (In this context, "us" means "you and me.")

4.10 Pseudo Terminals

A normal terminal device driver connects a process to an actual terminal, as shown in Figure 4.5a, where the user at the terminal is running `vi`. A *pseudo terminal* device driver acts just like a terminal as far as the interactive (*slave*) process (`vi`) is concerned, but the other end is connected to a *master* process, not to an actual device. This allows the master process to feed input to the slave process as though it were coming from an actual terminal, and to capture the output from the slave process as though it were going to an actual terminal. The slave process can use `tcgetattr`, `tcsetattr`, and other terminal-only system calls just as it normally does, and they will all work exactly as it expects them to.

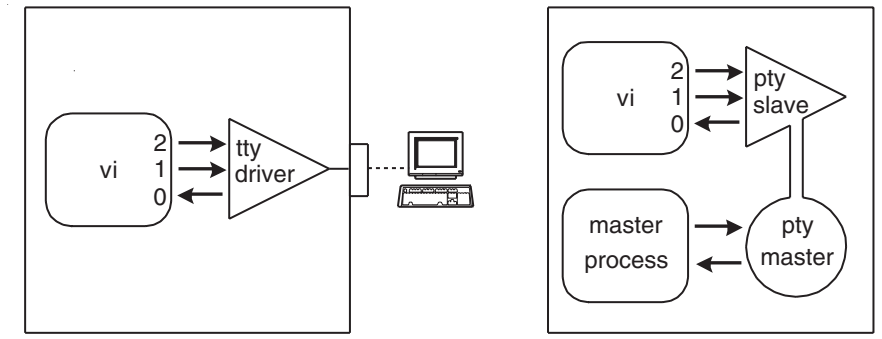


Figure 4.5 Terminal driver vs. pseudo-terminal driver.

You can think of connecting a pseudo terminal to a process as similar in concept to redirecting its input and output with the shell but doing it with a kind of terminal instead of with pipes. Indeed, `vi` won't work with pipes:

```
$ vi >tmp
ex/vi: Vi's standard input and output must be a terminal
```

Perhaps the most common use of pseudo terminals is by the `telnetd` server as shown in Figure 4.6. It communicates with the `telnet` client over a network and with whatever the `telnet` user is doing via a pseudo terminal. In practice, `telnet` sessions usually start with a shell, just as actual-terminal sessions; in the

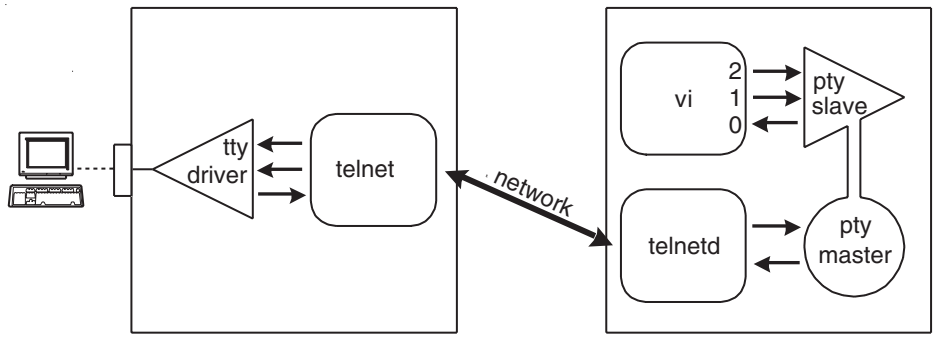


Figure 4.6 Pseudo terminal used by `telnet` server (`telnetd`).

picture `vi` is running because the user typed the `vi` command to the shell. In fact, these days, using `telnet` or `xterm` is far more common than using an actual terminal, as most users connect to a UNIX computer over a network, not by dialing it up directly. (They may dial up their Internet ISP, but that's just to get them onto the Internet.) I won't show how to implement a `telnetd` server (see Exercise 4.6); we're just concerned with the pseudo-terminal part of the picture.

On the pseudo-terminal master side, the master process gets whatever the slave process is writing. In the case of `vi` and other screen-oriented programs, this data includes escape sequences (e.g., to clear the screen), as we saw in Section 4.8. Neither the terminal driver nor the pseudo-terminal driver care about what those sequences are. On the other hand, control operations such as those done with `tcsetattr` are handled entirely by the drivers and don't get to the master process. So, in Figure 4.6, the escape sequences generated by `vi` get all the way to the `telnet` process running on the computer at the left. If it's running on the full screen, it can just send them straight to the actual terminal. It's even more common for the `telnet` process not to be connected to a terminal driver at all but to run under a window system. In this case it has to know how to translate the escape sequences to whatever the window system requires to display characters in a window. Such programs are called *terminal emulators*. So remember that terminal emulators and pseudo terminals are two very different things: the first processes escape sequences, and the second replaces the terminal driver so as to reroute the I/O to a master process.

4.10.1 Pseudo-Terminal Library

As this section uses a few system calls (e.g., `fork`) from Chapter 5, you may want to read this section after you've read that chapter.

Pseudo terminals (called "ptys" from now on), like all other devices, are represented by special files whose names vary from system to system. Unfortunately, connecting a process to a pty isn't nearly as straightforward as just doing something like this:

```
$ vi </dev/pty01 >/dev/pty01
```

It's more complicated: We have to get the name of the pty, execute some system calls to prepare it, and make it the controlling terminal. Some of the system calls involved are standardized by SUS1 (Section 1.5.1) and later standards, but pre-SUS systems like FreeBSD do things a completely different way. Worse, systems that use STREAMS to implement ptys require some extra steps, and there were some changes between SUS2 and SUS3.

I'm going to encapsulate the differences between the systems into a little library of functions, each of which starts with `pt_`. Then I'll use this pt-library to implement a record/playback application as an example.

Here's the overall scheme for using ptys presented as nine steps:

1. Open the master side of the pty for reading and writing.
2. Get access to the pty (explained below).
3. From the name or file descriptor of the master side, get the name of the slave side. Don't open it yet.
4. Execute the `fork` system call (Section 5.5) to create a child process.
5. In the child, call `setsid` (Section 4.3.2) to make it relinquish its controlling terminal.
6. In the child, open the slave side of the pty. It will become the new controlling terminal. On systems that support STREAMS (e.g., Solaris), you have to set up the STREAM. On BSD systems, you have to explicitly make it the controlling terminal with:

   ```
   ec_neg1( ioctl(fd, TIOCSCTTY) )
   ```

7. Redirect the child's standard input, output, and error file descriptors to the pty.

8. Execute the `execvp` system call (Section 5.3) to make the child run the desired program (e.g., `vi`). (Any of the six variants of the "exec" system call will work; `execvp` is the one I use in my example.)
9. Now the parent can read and write the master side of the pty using the file descriptor it got in step 1. The child will read from its standard input and write to its standard output and standard error output as it normally would (it knows nothing about the context in which it was executed), and those file descriptors will act as though they are open to a terminal device.

All of this plumbing is shown in Figure 4.5b.

There are three methods used to open the master side of a pty (step 1):

A. On a SUS3 system you just call `posix_openpt` (see below).
B. On most SUS1 and SUS2 systems (including Solaris and Linux), you open the clone file /dev/ptmx, and that will provide you with a unique pty, although you won't know the actual file name, if there even is one. That's OK, as all you need is the open file descriptor.
C. On BSD-based systems, there are a collection of special files whose names are of the form /dev/ptyXY, where X and Y are a digit or letter. You have to try them all until you find one you can open. (No kidding!)

We can use the `_XOPEN_VERSION` macro to distinguish between methods A and B, and we'll define the macro `MASTER_NAME_SEARCH` for those systems that use method C. In step 6 (of the 9 steps listed previously), we'll define the macro `NEED_STREAM_SETUP` for systems that need the STREAM to be set up, and the macro `NEED_TIOCSCTTY` for those systems that need the `ioctl` call. Here's the code in the pt-library that sets these macros for Solaris and FreeBSD; Linux doesn't need any of them set:

```
#if defined(SOLARIS) /* add to this as necessary */
#define NEED_STREAM_SETUP
#endif

#if defined(FREEBSD) /* add to this as necessary */
#define NEED_TIOCSCTTY
#endif

#ifndef _XOPEN_UNIX
#define MASTER_NAME_SEARCH
#endif
```

You'll want to augment this code for your own system if it's not already taken into account.

Here's the synopsis for `posix_openpt`, available only in SUS3-conforming systems:

posix_openpt—open pty

```
#include <stdlib.h>
#include <fcntl.h>

int posix_openpt(
    int oflag          /* O_RDWR optionally ORed with O_NOCTTY */
);
/* Returns file descriptor on success or -1 on error (sets errno) */
```

You use the flag `O_NOCTTY` to prevent the master side from being the controlling terminal, in case the process has no controlling terminal already. (Recall that the first terminal device opened for a process without a controlling terminal becomes the controlling terminal.) We usually don't want the master side to be the controlling terminal, although we will want exactly that for the slave side (step 5), so we will use the `O_NOCTTY` flag.

Now, getting on with the pt-library code, it needs some includes that aren't in defs.h:

```
#ifdef _XOPEN_UNIX
#include <stropts.h> /* for STREAMS */
#endif
#ifdef NEED_TIOCSCTTY
#include <sys/ttycom.h> /* for TIOCSCTTY */
#endif
```

The pt-library functions all operate on a `PTINFO` structure that holds file descriptors for the master and slave sides and their path names, if known:

```
#define PT_MAX_NAME 20

typedef struct {
    int pt_fd_m;                    /* master file descriptor */
    int pt_fd_s;                    /* slave file descriptor */
    char pt_name_m[PT_MAX_NAME];    /* master file name */
    char pt_name_s[PT_MAX_NAME];    /* slave file name */
} PTINFO;

#define PT_GET_MASTER_FD(p)  ((p)->pt_fd_m)
#define PT_GET_SLAVE_FD(p)   ((p)->pt_fd_s)
```

The two macros are for use by applications that call the pt-library, since they need the file descriptors.

The function `pt_open_master`, which I'll show shortly, allocates a `PTINFO` structure and returns a pointer to it for use with the other library functions, just as the standard I/O library uses a `FILE` structure. Internally, `pt_open_master` calls `find_and_open_master` to implement step 1, using one of the three methods previously outlined:

```
#if defined(MASTER_NAME_SEARCH)
#define PTY_RANGE \
  "0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz"
#define PTY_PROTO    "/dev/ptyXY"
#define PTY_X        8
#define PTY_Y        9
#define PTY_MS       5 /* replace with 't' to get slave name */
#endif /* MASTER_NAME_SEARCH */

static bool find_and_open_master(PTINFO *p)
{
#if defined(_XOPEN_UNIX)
#if _XOPEN_VERSION >= 600
    p->pt_name_m[0] = '\0'; /* don't know or need name */
    ec_neg1( p->pt_fd_m = posix_openpt(O_RDWR | O_NOCTTY) )
#else
    strcpy(p->pt_name_m, "/dev/ptmx"); /* clone device */
    ec_neg1( p->pt_fd_m = open(p->pt_name_m, O_RDWR | O_NOCTTY) )
#endif
#elif defined(MASTER_NAME_SEARCH)
    int i, j;
    char proto[] = PTY_PROTO;

    if (p->pt_fd_m != -1) {
        (void)close(p->pt_fd_m);
        p->pt_fd_m = -1;
    }
    for (i = 0; i < sizeof(PTY_RANGE) - 1; i++) {
        proto[PTY_X] = PTY_RANGE[i];
        proto[PTY_Y] = PTY_RANGE[0];
        if (access(proto, F_OK) == -1) {
            if (errno == ENOENT)
                continue;
            EC_FAIL
        }
```

```
            for (j = 0; j < sizeof(PTY_RANGE) - 1; j++) {
                proto[PTY_Y] = PTY_RANGE[j];
                if ((p->pt_fd_m = open(proto, O_RDWR)) == -1) {
                    if (errno == ENOENT)
                        break;
                }
                else {
                    strcpy(p->pt_name_m, proto);
                    break;
                }
            }
            if (p->pt_fd_m != -1)
                break;
        }
        if (p->pt_fd_m == -1) {
            errno = EAGAIN;
            EC_FAIL
        }
#else
        errno = ENOSYS;
        EC_FAIL
#endif
        return true;

EC_CLEANUP_BGN
        return false;
EC_CLEANUP_END
}
```

Most of the code in `find_and_open_master` is for that silly name search—as many as 3844 names! To speed it up a bit, we use `access` (Section 3.8.1) to test whether each name of the form /dev/ptyX0 exists, and, if not, we don't bother with /dev/ptyX1, /dev/ptyX2, and the other 59 names in that series but just move to the next X.

Step 2 is to get access to the pty. On SUS systems, you have to call `grantpt` to get access to the slave side and `unlockpt` in case the pty was locked:

grantpt—get access to slave side of pty

```
#include <stdlib.h>

int grantpt(
    int fd              /* file descriptor */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

unlockpt—unlock pty

```
#include <stdlib.h>

int unlockpt(
    int fd              /* file descriptor */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

Step 3 is to get the name of the slave side. On SUS1 systems there's a system call for it:

ptsname—get name of slave side of pty

```
#include <stdlib.h>

char *ptsname(
    int fd              /* file descriptor */
);
/* Returns name or NULL on error (errno not defined) */
```

On BSD systems (`MASTER_NAME_SEARCH` defined) you get the name from the name of the master by replacing the "p" in /dev/ptyXY with a "t." That is, if you discovered that /dev/ptyK4 could be opened as the master side, the corresponding slave side's name will be /dev/ttyK4.

Here's the code for the function `pt_open_master` that calls the function `find_and_open_master` shown previously for step 1 and then does steps 2 and 3, ending up with the slave side named but not open:

```
PTINFO *pt_open_master(void)
{
    PTINFO *p = NULL;
    char *s;

    ec_null( p = calloc(1, sizeof(PTINFO)) )
    p->pt_fd_m = -1;
    p->pt_fd_s = -1;
    ec_false( find_and_open_master(p) )
#ifdef _XOPEN_UNIX
    ec_neg1( grantpt(p->pt_fd_m) )
    ec_neg1( unlockpt(p->pt_fd_m) )
    ec_null( s = ptsname(p->pt_fd_m) )
    if (strlen(s) >= PT_MAX_NAME) {
        errno = ENAMETOOLONG;
        EC_FAIL
    }
```

```
    strcpy(p->pt_name_s, s);
#elif defined(MASTER_NAME_SEARCH)
    strcpy(p->pt_name_s, p->pt_name_m);
    p->pt_name_s[PTY_MS] = 't';
#else
    errno = ENOSYS;
    EC_FAIL
#endif
    return p;

EC_CLEANUP_BGN
    if (p != NULL) {
        (void)close(p->pt_fd_m);
        (void)close(p->pt_fd_s);
        free(p);
    }
    return NULL;
EC_CLEANUP_END
}
```

Step 4, `fork`ing to create a child process, is done by the program that uses the library—there's no library function to do it.

Step 5 is to call `setsid` (Section 4.3.2) in the child to make it relinquish its controlling terminal because we want the pty to be the controlling terminal. Step 6, also done in the child, is to open the slave side, whose name we already have. Those two steps are done by `pt_open_slave`:

```
bool pt_open_slave(PTINFO *p)
{
    ec_neg1( setsid() )
    if (p->pt_fd_s != -1)
        ec_neg1( close(p->pt_fd_s) )
    ec_neg1( p->pt_fd_s = open(p->pt_name_s, O_RDWR) )
#if defined(NEED_TIOCSCTTY)
    ec_neg1( ioctl(p->pt_fd_s, TIOCSCTTY, 0) )
#endif
#if defined(NEED_STREAM_SETUP)
    ec_neg1( ioctl(p->pt_fd_s, I_PUSH, "ptem") )
    ec_neg1( ioctl(p->pt_fd_s, I_PUSH, "ldterm") )
#endif
    /*
        Changing mode not that important, so don't fail if it doesn't
        work only because we're not superuser.
    */
    if (fchmod(p->pt_fd_s, PERM_FILE) == -1 && errno != EPERM)
        EC_FAIL
    return true;
```

```
EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

Again, `pt_open_slave` must be called in the child, not in the parent. (We'll see an example shortly.) Systems for which we defined `NEED_STREAM_SETUP`, such as Solaris, require two modules to be pushed onto the stream: `ptem`, the pseudo-terminal emulator, and `ldterm`, the normal terminal module.[11] (The `ioctl` system calls and the `I_PUSH` command are standardized but not the module names that are used to set up a pty.)

At the end of `pt_open_slave` we change the mode of the pty, since on some systems it may not have suitable permissions. However, as it may not be owned by the user-ID of the process (that's system-dependent), the `fchmod` call (Section 3.7.1) may fail, in which case we proceed anyway hoping for the best.

Steps 7, 8, and 9 aren't done with the pt-library but with system calls introduced in Chapters 5 and 6. I'll show the code in the example that appears later in this section, however.

As the child (slave) and parent (master) processes are running independently, the parent has to make sure the child is completely set up before starting to read and write the pty in step 9. So, the pt-library has a call for the parent's use that doesn't return until it's safe to proceed:

```
bool pt_wait_master(PTINFO *p)
{
    fd_set fd_set_write;

    FD_ZERO(&fd_set_write);
    FD_SET(PT_GET_MASTER_FD(p), &fd_set_write);
    ec_neg1( select(PT_GET_MASTER_FD(p) + 1, NULL, &fd_set_write, NULL,
      NULL) )
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

11. To read about these on a Solaris system (and perhaps others that implement STREAMS), run `man ptem` and `man ldterm`.

All we're doing is using `select` (Section 4.2.3) to wait until the pty is writable.

The next function is `pt_close_master`, called in the parent to close the file descriptors when the pty is no longer needed and to free the `PTINFO` structure.

```
bool pt_close_master(PTINFO *p)
{
    ec_neg1( close(p->pt_fd_m) )
    free(p);
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

There's also a `pt_close_slave`, but it's usually not called because the child process is overlaid by another program in step 8. Also, since we redirected file descriptors, the child's end of the pty is by this time represented by the standard file descriptors (0, 1, and 2), and those are almost always closed automatically when the process terminates. Here it is anyway:

```
bool pt_close_slave(PTINFO *p)
{
    (void)close(p->pt_fd_s); /* probably already closed */
    free(p);
    return true;
}
```

If you find yourself calling `pt_close_slave` in your code, you're probably doing something wrong.

Note that both `pt_close_master` and `pt_close_slave` free the `PTINFO` structure. That's the way it should be because when they are executed the structure is allocated in both parent and child. This will be more clear when `fork` is explained in Section 5.5.

Putting it all together, here's the overall framework for using the pt-library calls:

```
    PTINFO *p = NULL;
    bool ok = false;

    ec_null( p = pt_open_master() ) /* Steps 1, 2, and 3 */
    switch (fork()) { /* Step 4 */
    case 0:
        ec_false( pt_open_slave(p) ) /* Steps 5 and 6 */
        /*
            Redirect fds and exec (not shown) - steps 7 and 8
        */
```

```
        break;
    case -1:
        EC_FAIL
    }
    ec_false( pt_wait_master(p) ) /* Synchronize before step 9 */
    /*
        Parent (master) now reads and writes pty as desired - step 9
    */
    ok = true;
    EC_CLEANUP

EC_CLEANUP_BGN
    if (p != NULL)
        (void)pt_close_master(p);
    /*
        Other clean-up goes here
    */
    exit(ok ? EXIT_SUCCESS : EXIT_FAILURE);
EC_CLEANUP_END
```

This is what the application I'm about to present will look like.

4.10.2 Record and Playback Example

I'm going to use a pty to build a record/playback system that can record the output of a command and then play the recorded output back just as if the command were running. This means not only that the screen has to be written identically to the original but that the playback takes place at the same speed. To see what I mean, here's a script that shows the time, waits 5 seconds, and shows it again:

```
$ cat >script1
date
sleep 5
date
$ chmod +x script1
$ script1
Wed Nov  6 10:46:31 MST 2002 [5 second pause]
Wed Nov  6 10:46:36 MST 2002
```

Of course, as you can see from the times, there was a 5-second pause between the first and second executions of `date`. (The comment in brackets isn't part of the output.)

But if we just capture the output on a file and display that file, the text in the file is displayed very rapidly. Sure, the times in the file were exactly what the `date` command wrote, but there's no pause when the `cat` command prints the output:

```
$ script1 >tmp                    [5 second pause]
$ cat tmp
Wed Nov  6 10:55:24 MST 2002      [no pause]
Wed Nov  6 10:55:29 MST 2002
```

However, this is not what we want. We want to record the output so it plays back at the same speed as the recording, with a command we're going to show called `record` (its `-p` option plays the recording back):

```
$ record script1
Wed Nov  6 10:57:18 MST 2002      [5 second pause]
Wed Nov  6 10:57:23 MST 2002
$ record -p
Wed Nov  6 10:57:18 MST 2002      [5 second pause]
Wed Nov  6 10:57:23 MST 2002
```

You can't see it on the page, but after displaying the first line, there was a 5 second pause before displaying the second, which was the same speed at which the output was recorded.

So, at a minimum, what `record` needs to do is save *events:* the time along with the characters so the playback part will know how fast to write them to the standard output. We want it to work with interactive programs, too, so it can't just capture the output with a pipe—it has to set up the command with a pty because interesting commands, like `vi`, `emacs`, and other full-screen applications, will work only on terminals. How the recorder, the recording, and the command being recorded are all connected as shown in Figure 4.7. Whatever `record` reads from its standard input it writes to the pty, and whatever it reads from the pty it writes both to its standard output *and* to a file of events.

First, I'll show the data structures and functions that record the output from the command as a series of events. Then, I'll show the playback functions that read

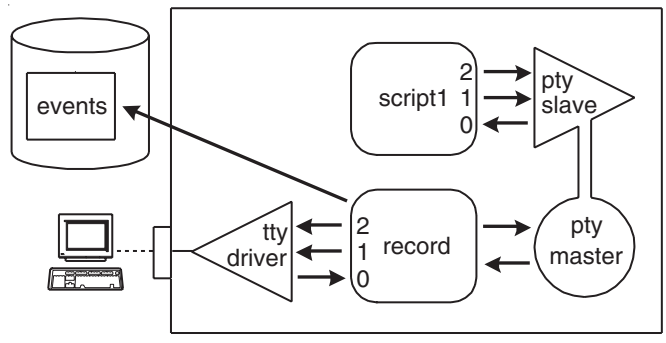


Figure 4.7 Recording interaction with `script1`.

the events and display the data in them at the appropriate times. Playback doesn't involve ptys, since the command isn't actually running. Finally, I'll show the recorder, which follows the framework for using the pt-library that I showed in the previous section.

Each time the process to be recorded writes some data, the master side of the pty reads it and treats it as one event. It stores the relative time since the recording began and the length of the data in an `event` structure. Then it writes the structure and the data itself to a file named recording.tmp. Figure 4.8 shows three such events in the file.

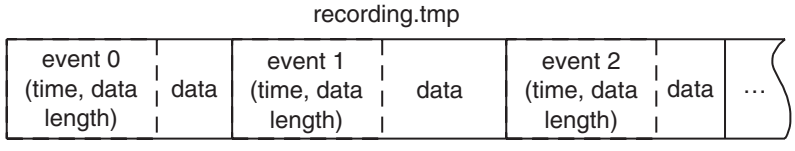


Figure 4.8 Layout of recording file.

This is the `event` structure along with a global file descriptor and macros for the file name and a buffer size for writing and reading data:

```
#define EVFILE "recording.tmp"
#define EVBUFSIZE 512

struct event {
    struct timeval e_time;
    unsigned e_datalen;
};
static int fd_ev = -1;
```

The `timeval` structure was explained in Section 1.7.1.

The recorder uses `ev_creat` to open the recording file (creating it if necessary), `ev_write` to write an event to it, and `ev_close` to close it:

```
static bool ev_creat(void)
{
    ec_neg1( fd_ev = open(EVFILE, O_WRONLY | O_CREAT | O_TRUNC,
      PERM_FILE) )
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

```
static bool ev_write(char *data, unsigned datalen)
{
    struct event ev = { { 0 } };

    get_rel_time(&ev.e_time);
    ev.e_datalen = datalen;
    ec_neg1( writeall(fd_ev, &ev, sizeof(ev)) )
    ec_neg1( writeall(fd_ev, data, datalen) )
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}

static void ev_close(void)
{
    (void)close(fd_ev);
    fd_ev = -1;
}
```

`writeall` is a simple way of writing a full buffer, even if it takes multiple calls to `write`; it's from Section 2.9.

The function `get_rel_time` does the work of calling `gettimeofday` (Section 1.7.1) to get the current time and subtracting it from the starting time (which it saves on its first call) using another function, `timeval_subtract`:

```
static void get_rel_time(struct timeval *tv_rel)
{
    static bool first = true;
    static struct timeval starttime;
    struct timeval tv;

    if (first) {
        first = false;
        (void)gettimeofday(&starttime, NULL);
    }
    (void)gettimeofday(&tv, NULL);
    timeval_subtract(&tv, &starttime, tv_rel);
}

static void timeval_subtract(const struct timeval *x,
  const struct timeval *y, struct timeval *diff)
{
    if (x->tv_sec == y->tv_sec || x->tv_usec >= y->tv_usec) {
        diff->tv_sec = x->tv_sec - y->tv_sec;
        diff->tv_usec = x->tv_usec - y->tv_usec;
    }
```

```
    else {
        diff->tv_sec = x->tv_sec - 1 - y->tv_sec;
        diff->tv_usec = 1000000 + x->tv_usec - y->tv_usec;
    }
}
```

Note that `timeval_subtract` assumes that `x >= y`.

Those are all the service functions needed to make a recording. Playing one back needs functions for opening the recording file and reading the `event` structure and the data that follows it in the file:

```
static bool ev_open(void)
{
    ec_neg1( fd_ev = open(EVFILE, O_RDONLY) )
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}

static bool ev_read(struct event *ev, char *data, unsigned datalen)
{
    ssize_t nread;

    ec_neg1( nread = read(fd_ev, ev, sizeof(*ev)) )
    if (nread != sizeof(*ev)) {
        errno = EIO;
        EC_FAIL
    }
    ec_neg1( nread = read(fd_ev, data, ev->e_datalen) )
    if (nread != ev->e_datalen) {
        errno = EIO;
        EC_FAIL
    }
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

One more thing: Once the player has an event, it needs to wait for the appropriate time before displaying it. This is the principal difference between real-time playback and just dumping the data out. The function `ev_sleep` takes a time from an `event` structure, calculates how much time to wait, and then sleeps for that interval. As a `timeval` stores the time in seconds and microseconds, we use the

Standard C function `sleep` for the seconds and the `usleep` system call (Section 9.7.3) for the microseconds.

```
static bool ev_sleep(struct timeval *tv)
{
    struct timeval tv_rel, tv_diff;

    get_rel_time(&tv_rel);
    if (tv->tv_sec > tv_rel.tv_sec ||
      (tv->tv_sec == tv_rel.tv_sec && tv->tv_usec >= tv_rel.tv_usec)) {
        timeval_subtract(tv, &tv_rel, &tv_diff);
        (void)sleep(tv_diff.tv_sec);
        ec_neg1( usleep(tv_diff.tv_usec) )
    }
    /* else we are already running late */
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

That's all we need to record and play back events. Assuming that a recording has been made already, here's the function that plays it back:

```
static bool playback(void)
{
    bool ok = false;
    struct event ev;
    char buf[EVBUFSIZE];
    struct termios tbuf, tbufsave;

    ec_neg1( tcgetattr(STDIN_FILENO, &tbuf) )
    tbufsave = tbuf;
    tbuf.c_lflag &= ~ECHO;
    ec_neg1( tcsetattr(STDIN_FILENO, TCSAFLUSH, &tbuf) )
    ec_false( ev_open() )
    while (true) {
        ec_false( ev_read(&ev, buf, sizeof(buf)) )
        if (ev.e_datalen == 0)
            break;
        ev_sleep(&ev.e_time);
        ec_neg1( writeall(STDOUT_FILENO, buf, ev.e_datalen) )
    }
    ec_neg1( write(STDOUT_FILENO, "\n", 1) )
    ok = true;
    EC_CLEANUP

EC_CLEANUP_BGN
```

```
    (void)tcdrain(STDOUT_FILENO);
    (void)sleep(1); /* Give the terminal a chance to respond. */
    (void)tcsetattr(STDIN_FILENO, TCSAFLUSH, &tbufsave);
    ev_close();
    return ok;
EC_CLEANUP_END
}
```

The reading of events and the writing to `STDOUT_FILENO` is pretty straightforward in this function. Certain escape sequences written to a terminal cause it to respond, however, and while we don't care about the response during playback (whatever it might have meant has already been captured in the recording), we do have to turn off echo to make sure the response doesn't mess up the output. At the end, we call `tcdrain` to get the last of the output out, wait a second for the terminal to process it, and then flush any remaining input when we restore the terminal's attributes. The functions `tcgetattr`, `tcsetattr`, and `tcdrain` are from Sections 4.5.1 and 4.6.

Now we can use the pt-library from the previous section along with the "ev" routines to make a recording. (You might want to review the framework at the end of the previous section.) I'll present the `main` function in pieces; it starts like this:

```
int main(int argc, char *argv[])
{
    bool ok = false;
    PTINFO *p = NULL;

    if (argc < 2) {
        fprintf(stderr, "Usage: record cmd ...\n       record -p\n");
        exit(EXIT_FAILURE);
    }
    if (strcmp(argv[1], "-p") == 0) {
        playback();
        ok = true;
        EC_CLEANUP
    }
```

If it's called with the `-p` option, it just calls the `playback` function I already showed and exits. Otherwise, for recording, it follows the framework:

```
    ec_null( p = pt_open_master() )
    switch (fork()) {
    case 0:
        ec_false( pt_open_slave(p) )
        ec_false( exec_redirected(argv[1], &argv[1], PT_GET_SLAVE_FD(p),
          PT_GET_SLAVE_FD(p), PT_GET_SLAVE_FD(p)) )
        break;
```

```
case -1:
    EC_FAIL
}
ec_false( ev_creat() )
ec_false( pt_wait_master(p) )
```

I put the stuff having to do with redirecting file descriptors and executing a command in the function `exec_redirected` because I'm going to explain it all in Chapters 5 and 6. I will show the code for `exec_redirected`, however, without explaining it. You can come back to it after reading those chapters if you need to.

The code so far in `main` takes us to step 9 (explained in the previous section), which is when we begin reading and writing the pty's master-side file descriptor. Bear in mind that the subject command (`script1`, `vi`, or whatever) is already running and is probably blocked in a `read` waiting for some input from what it thinks is a terminal. Here's the rest of the `main` function:

```
tc_setraw();
while (true) {
    fd_set fd_set_read;
    char buf[EVBUFSIZE];
    ssize_t nread;

    FD_ZERO(&fd_set_read);
    FD_SET(STDIN_FILENO, &fd_set_read);
    FD_SET(PT_GET_MASTER_FD(p), &fd_set_read);
    ec_neg1( select(FD_SETSIZE, &fd_set_read, NULL, NULL, NULL) )
    if (FD_ISSET(STDIN_FILENO, &fd_set_read)) {
        ec_neg1( nread = read(STDIN_FILENO, &buf, sizeof(buf)) )
        ec_neg1( writeall(PT_GET_MASTER_FD(p), buf, nread) )
    }
    if (FD_ISSET(PT_GET_MASTER_FD(p), &fd_set_read)) {
        if ((nread = read(PT_GET_MASTER_FD(p), &buf,
          sizeof(buf))) > 0) {
            ec_false( ev_write(buf, nread) )
            ec_neg1( writeall(STDOUT_FILENO, buf, nread) )
        }
        else if (nread == 0 || (nread == -1 && errno == EIO))
            break;
        else
            EC_FAIL
    }
}
ec_neg1( ev_write(NULL, 0) )
fprintf(stderr,
  "EOF or error reading stdin or master pseudo-terminal; exiting\n");
ok = true;
```

```
    EC_CLEANUP

EC_CLEANUP_BGN
    if (p != NULL)
        (void)pt_close_master(p);
    tc_restore();
    ev_close();
    printf("\n");
    exit(ok ? EXIT_SUCCESS : EXIT_FAILURE);
EC_CLEANUP_END
}
```

Comments on the last part of `main`:

- As `record` is mostly used for screen-oriented commands, we put the terminal in raw mode with `tc_setraw` (in Section 4.5.10). The attributes are restored by `tc_restore` in the clean-up code. We didn't use those functions in `playback` because all we wanted there was to turn off echo.
- `record` has two inputs: what the user types and what the pty sends back, as shown in Figure 4.5 in the previous section. It uses `select` to wait until one of those two inputs has some data, deals with it, and then loops back to the `select`. Note that it sets up `fd_set_read` each time, since `select` changes the bits to report its results.
- We break out of the loop when we get an end-of-file or I/O error from the pty, not from `record`'s standard input. After all, it's up to the subject command to decide when we're done, not us. An end-of-file or I/O error means that the slave side of the pty has all its file descriptors closed; that is, the subject command has terminated. Which indicator we get is implementation dependent, so we take either.

Finally, as promised, here's `exec_redirected`, which you will understand once you've finished Chapter 6:

```
bool exec_redirected(const char *file, char *const argv[], int fd_stdin,
  int fd_stdout, int fd_stderr)
{
    if (fd_stdin != STDIN_FILENO)
        ec_neg1( dup2(fd_stdin, STDIN_FILENO) )
    if (fd_stdout != STDOUT_FILENO)
        ec_neg1( dup2(fd_stdout, STDOUT_FILENO) )
    if (fd_stderr != STDERR_FILENO)
        ec_neg1( dup2(fd_stderr, STDERR_FILENO) )
    if (fd_stdin != STDIN_FILENO)
        (void)close(fd_stdin);
    if (fd_stdout != STDOUT_FILENO)
```

```
        (void)close(fd_stdout);
    if (fd_stderr != STDERR_FILENO)
        (void)close(fd_stderr);
    ec_neg1( execvp(file, argv) )

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

It's hard to really appreciate `record` in action in a book—we'd need a video to do it justice. Try it out for yourself! You'll smile when you see how it plays back the recording with the same timing as the original—almost as though you're watching a ghost at the keyboard.

Exercises

4.1. Change `getln` to return lines ended with an EOF, as discussed in Section 4.2.1.

4.2. Implement `Bfdopen`, as discussed in Section 4.2.1.

4.3. Implement a simplified version of the `stty` command that just prints out the current terminal status.

4.4. Implement a version of the `stty` command that allows the user to specify only the 10 most commonly used operands. You have to decide which 10 these are.

4.5. Implement a simple screen editor. Decide on a small selection of functions that allow reasonable editing but without too many bells and whistles. Keep the text being edited in internal memory. Use Curses (or equivalent) to update the screen and read the keyboard. Write some documentation using the `man`-page macros if you can.

4.6. Implement a `telnetd` server. For full details of what it's supposed to do, see [RFC854]. But, for this exercise, you can implement just enough so that you can connect from another machine that's running `telnet` and, through your server, execute both line-oriented commands and `vi` or `emacs`.

5

Processes and Threads

5.1 Introduction

We now leave the subject of input and output to begin investigating the multitasking features of UNIX. This chapter deals with techniques for invoking programs and processes, using `exec`, `fork`, `wait`, and related system calls. The next chapter explains simple interprocess communication using pipes. Chapters 7 and 8 continue with more advanced interprocess communication mechanisms.

My presentation is organized with the goal of implementing a fairly complete command interpreter, or shell. We'll start with a limited shell that's barely usable. Then we'll keep adding features until we end up, in the next chapter, with a shell that can handle I/O redirection, pipelines, background processes, quoted arguments, and environment variables.

5.2 Environment

I'll begin with a discussion of the environment, which most UNIX users are already familiar with at the shell level. When a UNIX program is executed, it receives two collections of data from the process that invoked it: the *arguments* and the *environment.* To C programs, both are in the form of an array of character pointers, all but the last of which point to a NUL-terminated character string. The last pointer is `NULL`. A count of the number of arguments is also passed on. Other languages use a different interface, but we're concerned here only with C and C++.

A C or C++ program begins in one of two ways:[1]

1. Some implementations allow a third array-of-string argument that contains the environment, but this is both nonstandard and unnecessary.

main—C or C++ program entry point

```
int main(
    int argc,              /* argument count */
    char *argv[]           /* array of argument strings */
)

int main(void)
```

The count `argc` doesn't include the `NULL` pointer that terminates the `argv` array. If the program doesn't take arguments, the count and array can be omitted, as in the second form. I've already shown examples of both forms in this book.

In addition, the global variable `environ` points to the array of environment strings, also `NULL` terminated (there's no associated count variable):

environ—environment strings

```
extern char **environ;     /* environment array (not in any header) */
```

Each argument string can be anything at all, as long as it's NUL-terminated. Environment strings are more constrained. Each is in the form *name=value,* with the NUL byte after the value. Of course, the name can't contain a = character.

I'll get to how the environment is passed to `main` (causing `environ` to be set) in Section 5.3; here I just want to talk about retrieving values from and modifying `environ`.

One way to get at the environment is just to access `environ` directly, like this:

```
extern char **environ;

int main(void)
{
    int i;

    for (i = 0; environ[i] != NULL; i++)
        printf("%s\n", environ[i]);
    exit(EXIT_SUCCESS);
}
```

On Solaris here are some of the lines that this program printed (the rest were omitted to save space):

```
HOME=/home/marc
HZ=100
LC_COLLATE=en_US.ISO8859-1
LC_CTYPE=en_US.ISO8859-1
```

```
LC_MESSAGES=C
LC_MONETARY=en_US.ISO8859-1
LOGNAME=marc
MAIL=/var/mail/marc
```

Listing the whole environment is an unusual requirement; usually a program wants the value of a specific variable, and for that there's the Standard C function `getenv`:

getenv—get value of environment variable

```
#include <stdlib.h>

char *getenv(
    const char *var        /* variable to find */
);
/* Returns value or NULL if not found (errno not defined) */
```

`getenv` returns just the value part, to the right of the = sign, as in this example:

```
int main(void)
{
    char *s;

    s = getenv("LOGNAME");

    if (s == NULL)
        printf("variable not found\n");
    else
        printf("value is \"%s\"\n", s);
    exit(EXIT_SUCCESS);
}
```

which printed:

```
value is "marc"
```

Updating the environment is not as easy as reading it. Although the internal storage it occupies is the exclusive property of the process, and may be modified freely, there is no guarantee of extra room for any new variables or for longer values. So unless the update is trivial, a completely new environment must be created. If the pointer `environ` is then made to point to this new environment, it will be passed on to any programs subsequently invoked (as we shall see in Section 5.3), and it will be used by subsequent calls to `getenv`. Any updates have no effect on any other processes, including the shell (or whatever) that invoked the process doing the updating. Thus, if you want to modify the shell's environment, you have to do so with commands built into the shell.

Rather than messing with the environment directly, you can use some standard functions:

putenv—change or add to environment

```
#include <stdlib.h>

int putenv(
     char *string           /* string of form name=value */
);
/* Returns 0 on success or non-zero on error (sets errno) */
```

setenv—change or add to environment

```
#include <stdlib.h>

int setenv(
     const char *var,       /* variable to be changed or added */
     const char *val,       /* value */
     int overwrite          /* overwrite? */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

unsetenv—remove environment variable

```
#include <stdlib.h>

int unsetenv(
     const char *var        /* variable to be removed */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

All of the functions modify the storage pointed to by `environ`, perhaps setting `environ` to a new value if the array of pointers isn't long enough. If you've modified `environ` yourself or any parts of the environment it points to, the behavior of these functions is undefined, so decide whether you're going to do it yourself or use the functions but don't mix the two approaches.

`putenv` takes complete environment strings of the form name=value and makes a pointer in the `environ` array point to the storage you pass in, which then becomes part of the environment, so don't pass in any automatically allocated data (local, nonstatic variables), and don't modify the string after you've called `putenv`.

`setenv` is more sophisticated: It copies the variable name and value you pass in and allocates its own storage for it. If the variable already exists, its value is changed if the third `overwrite` argument is nonzero; otherwise, the old value

stays. If the variable doesn't already exist, it's added to the environment and the value of `overwrite` doesn't matter.

`unsetenv` comes with `setenv`, and it removes a variable and value from the environment. If your system doesn't have `unsetenv`, the best you can do to remove a variable from the environment is to set its value to the empty string, which may or may not be acceptable, depending on the application. (Some systems, such as Linux, FreeBSD, and Darwin, define `unsetenv` as a `void` function, so there's no error return.)

While the interfaces to these functions is standardized, their presence isn't required. FreeBSD, Linux, and Solaris all have `putenv`, and the first two have all three. SUS3 requires `setenv` and `unsetenv`, but as of this writing there aren't any SUS3 systems, so there's no simple, portable way to determine which are present. `setenv` and `unsetenv` are from BSD and should be present in all systems derived from it, including FreeBSD; `putenv` is from System V and should be present in all systems derived from it, including Solaris. Since the SUS recommends `setenv` and `unsetenv`, perhaps the best approach is to code with those functions and just include your own implementations for those systems that don't have them already.

It's easy to implement `setenv` and `unsetenv` if you don't care about memory leaks. The problem is that they may have to allocate or, in the case of `unsetenv`, orphan memory, but they can't free the memory because they can't assume anything about how it was allocated in the first place. That defect notwithstanding, here's our version of `setenv`:

```
int setenv(const char *var, const char *val, int overwrite)
{
    int i;
    size_t varlen;
    char **e;

    if (var == NULL || val == NULL || var[0] == '\0' ||
      strchr(var, '=') != NULL) {
        errno = EINVAL;
        return -1;
    }
    varlen = strlen(var);
    for (i = 0; environ[i] != NULL; i++)
        if (strncmp(environ[i], var, varlen) == 0 &&
          environ[i][varlen] == '=')
            break;
```

```
    if (environ[i] == NULL) {
        if ((e = malloc((i + 2) * sizeof(char *))) == NULL)
            return -1;
        memcpy(e, environ, i * sizeof(char *));
        /* possible memory leaks with old pointer array */
        environ = e;
        environ[i + 1] = NULL;
        return setnew(i, var, val);
    }
    else {
        if (overwrite) {
            if (strlen(&environ[i][varlen + 1]) >= strlen(val)) {
                strcpy(&environ[i][varlen + 1], val);
                return 0;
            }
            return setnew(i, var, val);
        }
        return 0;
    }
}
```

Comments about this function:

- The first thing we do is check the arguments, which is required by the SUS.
- Next we search the environment to see if the variable is already defined. Note that we have to look for a string that starts with the name followed by an = sign.
- If the variable is found, we exit if the `overwrite` argument is false. Otherwise, there are two cases: The new value fits, in which case we just copy it in, or it doesn't, in which case we call the function `setnew` (below) to put it in.
- If the variable is not found, we have to grow the array. We can't use `realloc` because we can't assume how the old memory was allocated, as we said. So we use `malloc` and copy the old contents in ourselves. Then we terminate the new array with a `NULL` pointer and call `setnew` to put the new entry in.

Here's `setnew`, which allocates space for a name, an = sign, and a value and stores it in the array:

```
static int setnew(int i, const char *var, const char *val)
{
    char *s;

    if ((s = malloc(strlen(var) + 1 + strlen(val) + 1)) == NULL)
        return -1;
    strcpy(s, var);
```

```
    strcat(s, "=");
    strcat(s, val);
    /* possible memory leak with old value of environ[i] */
    environ[i] = s;
    return 0;
}
```

Lastly, we have `unsetenv` that checks the argument, finds it, and, if it's there, slides the rest of the array down to effectively remove it. This also creates a possible memory leak, as we can't assume that the memory for the unset variable can be freed.

```
int unsetenv(const char *var)
{
    int i, found = -1;
    size_t varlen;

    if (var == NULL || var[0] == '\0' || strchr(var, '=') != NULL) {
        errno = EINVAL;
        return -1;
    }
    varlen = strlen(var);
    for (i = 0; environ[i] != NULL; i++)
        if (strncmp(environ[i], var, varlen) == 0 &&
          environ[i][varlen] == '=')
            found = i;
    if (found != -1)
        /* possible memory leak with old value of environ[found] */
        memmove(&environ[found], &environ[found + 1],
          (i - found) * sizeof(char *));
    return 0;
}
```

Note that we use `memmove` rather than `memcpy` because the former is guaranteed to work if the source and target memory areas overlap, as they certainly do in this case.

You may be wondering why we didn't use our "ec" error-checking macros in these functions. The reason is that we wanted their behavior to be as close to the standard functions as possible.

It's fairly straightforward to correct the memory-leak problems in `setenv` and `unsetenv`; see Exercise 5.1.

5.3 exec System Calls

It's impossible to understand the `exec` or `fork` system calls without fully understanding the distinction between a process and a program. If these terms are new to you, you may want to go back and review Section 1.1.2. If you're ready to proceed now, we'll summarize the distinction in one sentence: A process is an execution environment that consists of instruction, user-data, and system-data segments, as well as lots of other resources acquired at runtime, whereas a program is a file containing instructions and data that are used to initialize the instruction and user-data segments of a process.

The `exec` system calls reinitialize a process from a designated program; the program changes while the process remains. On the other hand, the `fork` system call (the subject of Section 5.5) creates a new process that is a clone of an existing one, by just copying over the instruction, user-data and system-data segments; the new process is not initialized from a program, so old and new processes execute the same instructions.

Individually, `fork` and `exec` are of limited use—mostly, they're used together. Keep this in mind as I present them separately, and don't be alarmed if you think they're useless—just try to understand what they do. In Section 5.4, when we use them together, you'll see that they are a powerful pair.

Aside from booting the UNIX kernel itself, `exec` is the only way programs get executed on UNIX. Not only does the shell use `exec` to execute our programs, but the shell and its ancestors were invoked by `exec`, too. And `fork` is the only way new processes get created.

Actually, there is no system call named "exec." The so-called "exec" system calls are a set of six, with names of the form `exec`*AB*, where *A* is either `l` or `v`, depending on whether the arguments are directly in the call (list) or in an array (vector), and *B* is either absent, a `p` to indicate that the `PATH` environment variable should be used to search for the program, or an `e` to indicate that a particular environment is to be used. (Oddly, you can't get both the `p` and `e` features in the same call.) Thus, the six names are `execl`, `execv`, `execlp`, `execvp`, `execle`, and `execve`.[2] We'll start with `execl`, and then do the other five.

2. The same features could be easily provided in two system calls instead of six, but it's a little late to change. See Exercise 5.6.

execl—execute file with argument list

```
#include <unistd.h>

int execl(
     const char *path,      /* program pathname */
     const char *arg0,      /* first arg (file name) */
     const char *arg1,      /* second arg (if needed) */
     ...,                   /* remaining args (if needed) */
     (char *)NULL           /* arg list terminator */
);
/* Returns -1 on error (sets errno) */
```

The argument `path` must name a program file that is executable by the effective user-ID (mode 755, say) and has the correct contents for executable programs. The process's instruction segment is overwritten by the instructions from the program, and the process's user-data segment is overwritten by the data from the program, with a re-initialized stack. Then the process executes the new program from the top (that is, its `main` function is called).

There can be no return from a successful `execl` because the return location is gone. An unsuccessful `execl` does return, with a value of –1, but there's no need to test for that value, for no other value is possible. The most common reasons for an unsuccessful `execl` is that the path doesn't exist or it isn't executable.

The arguments to `execl` that follow `path` are collected into an array of character pointers; the last argument, which must be `NULL`, stops the collection and terminates the array. The first argument, by convention, is the name of the program file (not the entire path). The new program may access these arguments through the familiar `argc` and `argv` arguments of `main`. The environment pointed to by `environ` is passed on, too, and it is accessible through the new program's `environ` pointer or with `getenv`, as explained in the previous section.

Since the process continues to live, and since its system-data segment is mostly undisturbed, almost all of the process's attributes are unchanged, including its process-ID, parent-process-ID, process-group-ID, session-ID, controlling terminal, real user-ID and group-ID, current and root directories, priority, accumulated execution times, and, usually, open file descriptors. It's simpler to list the principal attributes that *do* change, all for good reasons:

- If the process had arranged to catch any signals, they are reset to the default action, since the instructions designated to catch them are gone. Ignored or defaulted signals stay that way. (More about signals in Chapter 9.)
- If the set-user-ID or set-group-ID bit of the new program file is on, the effective user-ID or group-ID is changed to the owner-ID or group-ID of the file. There's no way to get the former effective IDs back, if they were themselves different from the real IDs.
- Any functions registered with `atexit` (Section 1.3.4) are unregistered, since their code is gone.
- Shared memory segments (Section 7.12) are detached (unmapped), as the attachment point is no longer valid.
- POSIX named semaphores (Section 7.10) are closed. System V semaphores (Section 7.9) are undisturbed.

For the complete list, see Appendix A. But you get the idea: If retaining an attribute or resource isn't meaningful with an entirely new program running, it's reset to its default or closed.

To show how `execl` is used, here's a rather contrived example:

```
void exectest(void)
{
    printf("The quick brown fox jumped over ");
    ec_neg1( execl("/bin/echo", "echo", "the", "lazy", "dogs.",
      (char *)NULL) )
    return;

EC_CLEANUP_BGN
    EC_FLUSH("exectest");
EC_CLEANUP_END
}
```

Here's the output we got when we called the function:

```
the lazy dogs.
```

What happened to the fox? Well, it turns out it wasn't quick enough: The standard I/O Library, of which `printf` is a part, buffers its output, and any partially full last buffer is automatically flushed when the process exits. But the process didn't exit before `execl` was called, and the buffer, being in the user-data segment, got overlaid before it could be flushed.

The problem can be fixed by either forcing unbuffered output, like this:

```
setbuf(stdout, NULL);
```

or by flushing the buffer just before calling `execl`, like this:

```
fflush(stdout);
```

As I've mentioned, open file descriptors normally stay open across an `execl`. If we don't want a file descriptor to stay open, we close it first, with `close`. But sometimes we can't. Suppose we are invoking a program that requires all file descriptors to be closed (an extremely unconventional requirement). We might therefore try this:

```
errno = 0;
ec_neg1( open_max = sysconf(_SC_OPEN_MAX) )
for (fd = 0; fd < open_max; fd++)
    (void)close(fd); /* ignore errors */
ec_neg1( execl(path, arg0, arg1, arg2, (char *)NULL) )
```

We call `sysconf` (Section 1.5.5) to get the maximum number of files that a process can have open, and the highest-numbered file descriptor is one less than that. We want so badly to close all file descriptors here that we don't even care if they're all open—we close them anyhow. It's one of the rare instances where ignoring system-call error returns is appropriate.

All is fine if the `execl` succeeds, but if it fails we'll never see an error message because file descriptor 2 has been closed! Here's where the close-on-exec flag that we encountered in Section 3.8.3 can be used. When set, with `fcntl`, the file descriptor is closed upon a successful `execl` but left alone otherwise. We can either set this flag for only the important file descriptors or just set it for all of them, like this:

```
for (fd = 0; fd < open_max; fd++) {
    ec_neg1( flags = fcntl(fd, F_GETFD) )
    ec_neg1( fcntl(fd, F_SETFD, flags | FD_CLOEXEC) )
}
ec_neg1( execl(path, arg0, arg1, arg2, (char *)NULL) )
```

A problem with closing or setting close-on-exec for all file descriptors is that there may be a thousand or more of them, and this is a lot of processing, mostly wasted. At the same time, it's wrong to leave any but the standard file descriptors open across an `exec`, unless the new program is expecting more than the standard three, which is unusual. So, in most cases, you need to keep track of what file descriptors your program has opened and close them explicitly before an `exec`.

Because the standard file descriptors are normally left open, close-on-exec is rarely used. An example where it is useful would be on a file descriptor open to a

log file, which you would want to keep open in case the `exec` fails and you want to log that fact.

The other five `exec` system calls provide three features not available with `execl`:

- Putting the arguments into an array instead of listing them explicitly. This is necessary when the number of arguments is unknown at compile time, as in programming a shell, for example.
- Searching for the program file using the value of the `PATH` environment variable, as a shell does.
- Manually passing an explicit environment pointer, instead of automatically using `environ`. Since a successful `exec` overwrites the existing environment anyhow, there's seldom an advantage to this feature.[3]

Here are the synopses for the `exec` variants I haven't shown yet:

execv—execute file with argument vector

```
#include <unistd.h>

int execv(
     const char *path,      /* program pathname */
     char *const argv[]     /* argument vector */
);
/* Returns -1 on error (sets errno) */
```

execlp—execute file with argument list and PATH search

```
#include <unistd.h>

int execlp(
     const char *file,      /* program file name */
     const char *arg0,      /* first arg (file name) */
     const char *arg1,      /* second arg (if needed) */
     ...,                   /* remaining args (if needed) */
     (char *)NULL           /* arg list terminator */
);
/* Returns -1 on error (sets errno) */
```

execvp—execute file with argument vector and PATH search

```
#include <unistd.h>

int execvp(
     const char *file,      /* program file name */
     char *const argv[]     /* argument vector */
);
/* Returns -1 on error (sets errno) */
```

3. But see Exercise 5.2.

execle—execute file with argument list and environment

```
#include <unistd.h>

int execle(
     const char *path,      /* program pathname */
     const char *arg0,      /* first arg (file name) */
     const char *arg1,      /* second arg (if needed) */
     ...,                   /* remaining args (if needed) */
     (char *)NULL,          /* arg list terminator */
     char *const envv[]     /* environment vector */
);
/* Returns -1 on error (sets errno) */
```

execve—execute file with argument vector and environment

```
#include <unistd.h>

int execve(
     const char *path,      /* program pathname */
     char *const argv[],    /* argument vector */
     char *const envv[]     /* environment vector */
);
/* Returns -1 on error (sets errno) */
```

Note that the `argv` argument as used here is identical in layout to the `argv` argument of a `main` function. Don't forget that the last pointer has to be `NULL`.

If a `file` argument to `execlp` or `execvp` has no slashes, the strings listed in the value of the `PATH` variable are prepended to it one by one until an ordinary file with the resulting path name that has execute permission is located. If this file contains a program (as indicated by a code number in its first word), it is executed. If not, it is assumed to be a script file; usually, to run it a shell is executed with the path as its first argument.

For example, if the `file` argument is `echo`, and a search of the `PATH` strings finds an executable whose path is /bin/echo, then that file is executed as a program because it contains binary instructions in the right format. But if the `file` argument is, say, `myscript`, and a search turns up /home/marc/myscript which is not a binary executable, `execlp` or `execvp` instead executes the equivalent of:

```
sh /home/marc/myscript arg1 arg2 …
```

It's up to the implementation as to what shell it uses, but it must be standards-conforming.

There is one other universally supported, but nonstandard, convention: If the first line of the script is of the form

```
#! pathname [arg]
```

the interpreter specified by `pathname` is executed, instead of the shell, with the pathname of the script file as its first argument. If there's an optional `arg` on the `#!` line, it becomes the first argument to the program, and the script pathname becomes the second. The interpreter has to bypass the `#!` line, which is why most UNIX scripting languages use `#` to start a comment.

For example, you can put a script like this in the file `mycmd`:

```
#! /usr/bin/python2.2
print "Hello World!"
```

make it executable, and then execute it like this:

```
$ mycmd
Hello World!
$
```

On nearly all systems, the work to deal with the `#!` line is done by `execvp`, not by the shell itself.

If the `PATH` search turns up nothing executable, the `exec` fails. If the `file` argument has slashes in it, no search is done—the path is assumed to be complete. It still may be a script, however.

It's possible to program `execvp` and `execlp` as library functions that call on `execv` or `execl`, and we can learn a lot by doing so. I'll show a version of `execvp` that's close to the standard version, but which uses the "ec" error-checking macros.

Recall that for each path found by searching `PATH`, `execvp` first has to try to execute it as a binary executable and then again as a shell script. Here's a function that does that much (omitting the `#!` feature):

```
int exec_path(const char *path, char *const argv[], char *newargv[])
{
    int i;

    execv(path, argv);
    if (errno == ENOEXEC) {
        newargv[0] = argv[0];
        newargv[1] = (char *)path;
        i = 1;
        do {
            newargv[i + 1] = argv[i];
        } while (argv[i++] != NULL);
        return execv("/bin/sh", (char *const *)newargv);
    }
```

```
    return -1;
}
```

`exec_path` returns –1 with `errno` set if it fails to execute the path one way or another.

Our version of `execvp`, called `execvp2`, has to try each of the paths in `PATH` in turn, calling `exec_path` each time, unless the passed-in file contains a slash or `PATH` is nonexistent, in which case it just uses it as is:

```
int execvp2(const char *file, char *const argv[])
{
    char *s, *pathseq = NULL, *path = NULL, **newargv = NULL;
    int argc;

    for (argc = 0; argv[argc] != NULL; argc++)
        ;
    /* If shell script, we'll need room for one additional arg and NULL. */
    ec_null( newargv = malloc((argc + 2) * sizeof(char *)) )
    s = getenv("PATH");
    if (strchr(file, '/') != NULL || s == NULL || s[0] == '\0')
        ec_neg1( exec_path(file, argv, newargv) )
    ec_null( pathseq = strdup(s) )
    /* Following line usually allocates too much */
    ec_null( path = malloc(strlen(file) + strlen(pathseq) + 2) )
    while ((s = strtok(pathseq, ":")) != NULL) {
        pathseq = NULL; /* tell strtok to keep going */
        if (s[0] == '\0')
            s = ".";
        strcpy(path, s);
        strcat(path, "/");
        strcat(path, file);
        exec_path(path, argv, newargv);
    }
    errno = ENOENT;
    EC_FAIL

EC_CLEANUP_BGN
    free(pathseq);
    free(path);
    free(newargv);
    return -1;
EC_CLEANUP_END
}
```

Some comments about the memory allocations follow:

- It's not necessary or even possible to free up allocated memory if an `exec` is successful, as all user data will be overwritten.
- Taking the length of `PATH` plus the length of the passed-in file name plus two more bytes for a slash and a NUL is definitely enough for every path we'll synthesize.
- If we're going to try executing a shell script, we need an argument vector with one more slot than the one passed in, so we count the number of arguments passed in (in the `for` loop) and then allocate a vector with two more than that many slots (one for the additional argument—the pathname for the shell to execute—and one for the `NULL` pointer at the end).
- We use `strdup` to allocate a new string to hold the value of `PATH` because `strtok` will write into it. The pointer we got directly from `getenv` points into the environment, and we don't want to modify those strings directly, in case `execvp2` fails. (Setting the first argument of `strtok` to `NULL` is something you have to do when calling that function—that's how it knows to continue scanning the original string.)

As we've said, normally `exec` is paired with `fork`, but occasionally it's useful by itself. Sometimes a large program is split into phases, and `exec` brings in the next phase. But since *all* instructions and user-data are overlaid, the phases would have to be quite independent for this to be workable—data can be passed only through arguments, the environment, or files. So this application is rare but not unknown. More commonly, `exec` is used by itself when we need to do a very small amount of preliminary work before invoking a command. An example is the `nohup` command, which suppresses hangup signals before calling `execvp` to invoke the user's command. Another example is the `nice` command, which changes a command's priority (see Section 5.15 for an example).

5.4 Implementing a Shell (Version 1)

We now know enough system calls to write a shell of our own, although not a very good one. Its chief deficiency is that it has to commit suicide in order to execute a command, since there is no return from `exec`. Perversely, if you type in bad commands, you can keep running. While we're at it, we'll implement two built-in commands to modify and access the environment: assignment (e.g., `BOOK=/usr/marc/book`) and `set`, which prints the environment.[4]

4. There's no `export` command; the whole environment is passed to executed commands.

The first task is to break up a command line into arguments. For simplicity, we'll do without quoted arguments. Moreover, since this shell can't handle background processes, sequential execution, or piping, we don't have to worry about the special characters `&`, `;`, and `|`. We'll leave out redirection (`>`, `>>`, and `<`) also. Therefore, a command line is just a series of words separated by blanks or tabs. We have to gather them up and put them into an `argv` array:

```
#define MAXLINE 200

static bool getargs(int *argcp, char *argv[], int max, bool *eofp)
{
    static char cmd[MAXLINE];
    char *cmdp;
    int i;

    *eofp = false;
    if (fgets(cmd, sizeof(cmd), stdin) == NULL) {
        if (ferror(stdin))
            EC_FAIL
        *eofp = true;
        return false;
    }
    if (strchr(cmd, '\n') == NULL) {
        /* eat up rest of line */
        while (true) {
            switch (getchar()) {
            case '\n':
                break;
            case EOF:
                if (ferror(stdin))
                    EC_FAIL
            default:
                continue;
            }
            break;
        }
        printf("Line too long -- command ignored\n");
        return false;
    }
    cmdp = cmd;
    for (i = 0; i < max; i++) {
        if ((argv[i] = strtok(cmdp, " \t\n")) == NULL)
            break;
        cmdp = NULL; /* tell strtok to keep going */
    }
    if (i >= max) {
        printf("Too many args -- command ignored\n");
        return false;
    }
```

```
    *argcp = i;
    return true;

EC_CLEANUP_BGN
    EC_FLUSH("getargs")
    return false;
EC_CLEANUP_END
}
```

Comments:

- `getargs` returns `true` if it parsed the arguments OK, and otherwise `false`.
- It uses the standard C function `fgets` to read a line of input. The function is safe enough if a long line is typed, since it won't overrun the buffer we pass in. But in that case we don't want to leave the unread characters to be read later, as they would then be an unintended, and potentially damaging, shell command in their own right. So in that case (no newline terminator) we read in and throw away the rest of the line before printing an error message.
- Then we use `strtok` to break up the line into its arguments. (We also used `strtok` in `execvp2`, in the previous section.)
- `getargs` displays its own error messages with `EC_FLUSH` in case of a genuine error from one of the library functions it calls, and otherwise—such as in the case of a line that's too long—prints a message for the user.

Next, we need functions to handle the built-in commands, assignment and `set`. These are easy, since we can use the environment-manipulation techniques from Section 5.2:

```
extern char **environ;

void set(int argc, char *argv[])
{
    int i;

    if (argc != 1)
        printf("Extra args\n");
    else
        for (i = 0; environ[i] != NULL; i++)
            printf("%s\n", environ[i]);
}

void asg(int argc, char *argv[])
{
    char *name, *val;
```

```
    if (argc != 1)
        printf("Extra args\n");
    else {
        name = strtok(argv[0], "=");
        val = strtok(NULL, ""); /* get all that's left */
        if (name == NULL || val == NULL)
            printf("Bad command\n");
        else
            ec_neg1( setenv(name, val, true) )
    }
    return;

EC_CLEANUP_BGN
    EC_FLUSH("asg")
EC_CLEANUP_END
}
```

Next, we complete the program with a `main` function that prints the prompt character (we use `@`), gets the arguments, checks to see if the command is built in, and if it's not, tries to execute it:

```
#define MAXARG 20

int main(void)
{
    char *argv[MAXARG];
    int argc;
    bool eof;

    while (true) {
        printf("@ ");
        if (getargs(&argc, argv, MAXARG, &eof) && argc > 0) {
            if (strchr(argv[0], '=') != NULL)
                asg(argc, argv);
            else if (strcmp(argv[0], "set") == 0)
                set(argc, argv);
            else
                execute(argc, argv);
        }
        if (eof)
            exit(EXIT_SUCCESS);

    }
}

static void execute(int argc, char *argv[])
{
    execvp(argv[0], argv);
    printf("Can't execute\n");
}
```

Note that we loop back to print another prompt only if `execvp` fails. That's why we're unlikely to find this shell useful. Here's a sample session anyhow (the listing of the environment has been abbreviated to save space):

```
$ sh0
@ set
SSH_CLIENT=192.168.0.1 4971 22
USER=marc
MAIL=/var/mail/marc
EDITOR=vi
@ LASTNAME=Rochkind
@ set
SSH_CLIENT=192.168.0.1 4971 22
USER=marc
MAIL=/var/mail/marc
EDITOR=vi
LASTNAME=Rochkind
@ echo Hello World!
Hello World!
$
```

Note the dollar-sign prompt at the end—after executing `echo`, our shell has exited. Actually, it was `echo` that exited, as it was the program that replaced the shell. Clearly, it would be better for a shell to execute commands in processes of their own, and that's exactly where we're headed.

5.5 `fork` System Call

`fork` is somewhat the opposite of `exec`: It creates a new process, but it does not initialize it from a new program. Instead, the new process's instruction, user-data, and system-data segments are almost exact copies of the old process's.

fork—create new process

```
#include <unistd.h>

pid_t fork(void);

/* Returns child process-ID and 0 on success or -1 on error (sets errno) */
```

After `fork` returns, *both* processes (parent and child) receive the return. The return value is different, however, which is crucial, because this allows their subsequent actions to differ. Usually, the child does an `exec`, and the parent either waits for the child to terminate or goes off to do something else.

The child receives a 0 return value from `fork`; the parent receives the process-ID of the child. As usual, a return of –1 indicates an error, but since `fork` has no

arguments, the caller could not have done anything wrong. The only cause of an error is resource exhaustion, such as insufficient swap space or too many processes already in execution. Rather than give up, the parent may want to wait a while (with `sleep`, say) and try again, although this is not what shells typically do—they print a message and reprompt.

Recall that a program invoked by an `exec` system call retains many attributes because the system-data segment was left mostly alone. Similarly, a child created by `fork` inherits most attributes from its parent because its system-data segment is copied from its parent. It is this inheritance that allows a user to set certain attributes from the shell, such as current directory, effective user-ID, and priority, that then apply to each command subsequently invoked. One can imagine these attributes belonging to the "immediate family," although that's not a formally defined UNIX term.

Only a few attributes are not inherited:

- Obviously, the child's process-ID and parent-process-ID are different, since it's a different process.
- If the parent was running multiple threads (Section 5.17), only the one that executed the `fork` exists in the child. All the threads remain intact in the parent, however.
- The child gets duplicates of the parent's open file descriptors. Each is opened to the same file, and the file pointer has the same value. The open file *description* (Section 2.2.3) and, hence, the file offset, is shared. If the child changes it with `lseek`, then the parent's next `read` or `write` will be at the new location. The file *descriptor* itself, however, is distinct: If the child closes it, the parent's copy is undisturbed.
- The child's accumulated execution times are reset to zero because it's at the beginning of its life.

(These are the main ones—see Appendix A for the complete list.)

Here's a simple example to show the effect of a `fork`:

```
void forktest(void)
{
    int pid;

    printf("Start of test\n");
    pid = fork();
    printf("Returned %d\n", pid);
}
```

The output:

```
$ forktest
Start of test
Returned 98657
Returned 0
$
```

In this case the parent executed its `printf` before the child, but in general you can't depend on which comes first because the processes are independently scheduled. If it matters, you'll have to synchronize the two yourself, as explained in Section 9.2.3. As shown there, you can do it with a pipe or, less easily, with a signal.

Let's run `forktest` again, but this time with the standard output redirected to a file:

```
$ forktest >tmp
$ cat tmp
Start of test
Returned 56807
Start of test
Returned 0
$
```

This time "Start of test" got written twice! Can you figure out why? (Spoiler in next paragraph.)

What happened was that `printf` buffered its output (as explained in Section 2.12) and the child inherited the unflushed buffer, along with everything else. When the child exited, it flushed the buffer, and so did the parent. Before, when the output wasn't redirected, `printf` didn't buffer because it knew that the standard output was a terminal device and that it should behave more interactively. I'll discuss how to control the side-effects of exiting in Section 5.7.

`fork` and `exec` are a perfect match because the child process created by `fork`, as it is a near clone of the parent, isn't usually useful. It's ideal, though, for being overlaid by a new program, which is precisely what `exec` does. We'll see that in the next section, as we improve on our shell.

The cost of a `fork` is potentially enormous. It goes to all the trouble of cloning the parent, perhaps copying a large data segment (the instruction segment is usually read-only and can be shared) only to execute a few hundred instructions before reinitializing the code and data segments. This seems wasteful and it is. A

clever solution is used in virtual-memory versions of UNIX, in which copying is particularly expensive, called *copy-on-write.* It works this way: On a `fork`, the parent and child share the data segment, which consists of a collection of *pages.* As long as a page stays unmodified, this shortcut works OK. Then as soon as parent or child attempts to write on a page, just that page is copied to give each its own version. This is transparent and efficient with the right kind of dynamic-address-translation hardware, which almost all modern-day computers have. Since the `exec` follows very quickly in almost all cases, very few pages will have to be copied. But even if they all have to be copied, we are no worse off—in fact, we're better off, since we were able to start running the child earlier. Remember that a copy-on-write scheme is internal to the kernel; it does not change the semantics of `fork`, and, apart from better performance, users are unaware of it.

Even more efficient than copy-on-write, but not as transparent, is a variation on `fork` called `vfork`:

vfork—create new process; share memory (*obsolete*)

```
#include <unistd.h>

pid_t vfork(void);

/* Returns child process-ID and 0 on success or -1 on error (sets errno) */
```

`vfork` behaves exactly like `fork` insofar as creating the child process is concerned, but parent and child share the same data segment. There's no copy-on-write—they both read and write the same variables, so havoc ensues unless the child immediately exits or does an `exec`, which is what normally happens anyway (see the example function `execute2` in the next section).[5] In early versions of BSD UNIX, `vfork` was important; however, its performance advantages over `fork` are no longer substantial and its use is dangerous, so you should not use it, which is why it's marked "obsolete" in the synopsis.

A more recent attempt to improve the efficiency of `fork` and `exec` is `posix_spawn`, part of a 1999 POSIX update. Because its end result is a child process running a different program, it avoids the problems of `vfork`. It doesn't offer all the flexibility of a separate `fork` and `exec`, but it does handle the most common cases (e.g., duplicating a file descriptor). The real purpose of

5. Two processes concurrently reading and writing the same variables is similar to threads, with similar potential dangers. Section 5.17 has more.

`posix_spawn` is to provide an efficient way to invoke a program as a child process in a realtime system when swapping would be too slow and the hardware lacks dynamic address translation. It's possible to implement `posix_spawn` as a library function that uses `fork` and `exec` (Exercise 5.8), although in a realtime system of the sort its designers envisioned `posix_spawn` would have to be implemented as a system call.

It's a good time to mention the Standard C function `system`:

system—run command

```
#include <stdlib.h>

int system(
    const char *command       /* command */
);
/* Returns exit status or -1 on error (sets errno) */
```

This function does a `fork` followed by an `exec` of the shell, passing the `command` argument as a shell command line. It's a handy way to execute a command line from within a program, but it doesn't provide the efficiency (the shell is always invoked) or flexibility of a separate `fork` followed by an `exec`, or of `posix_spawn`.

5.6 Implementing a Shell (Version 2)

Now let's put `fork` and `execvp` together to make our shell from the previous section far more useful—it won't just exit after running a command but will actually prompt for another one! We replace the function `execute` with `execute2`:

```
static void execute2(int argc, char *argv[])
{
    pid_t pid;

    switch (pid = fork()) {
    case -1: /* parent (error) */
        EC_FAIL
    case 0: /* child */
        execvp(argv[0], argv);
        EC_FAIL
    default: /* parent */
        ec_neg1( wait(NULL) )
    }
    return;
```

```
EC_CLEANUP_BGN
    EC_FLUSH("execute2")
    if (pid == 0)
        _exit(EXIT_FAILURE);
EC_CLEANUP_END
}
```

I'll present the details of `wait` fully in Section 5.8*;* for now, you just need to know that it waits for the child process to terminate before returning.

This function treats an error from `fork` and `wait` differently from an error from `execvp`. If `fork` or `wait` returns –1, we're in the parent, so we just print the error information (with `EC_FLUSH`) and return. The caller (`main` from the previous section) will then prompt the user again, which is what we want—a shell shouldn't terminate just because there's been an error. But if `execvp` returns, we are in the *child* process, not the parent, and we had better exit. Otherwise, the child would keep running and we would have *two* shells prompting for commands. If we actually tried to type a command, the two processes would compete for characters and each would get some of them, since a single character can only be read once. This could have serious consequences if, for example, we typed `rm t*` and one of the shells read `rm *` while the other read `t`.

We called `_exit` instead of `exit` for reasons I'm that going to explain next.

5.7 `exit` System Calls and Process Termination

There are exactly four ways that a process can terminate:

1. Calling `exit`. Returning a value from `main` is the same as calling `exit` with that value as an argument, and returning without a value is the same as returning 0.
2. Calling `_exit` or its equivalent, `_Exit`, two variants of `exit` that will be explained shortly.
3. Receiving a terminating signal.
4. A system crash, caused by anything from a power failure to a bug in the OS.

This section is concerned with the first two, Chapter 9 deals with signals, and there's really nothing to say about crashes, upon which processes just stop.[6]

6. Some computers send a signal to processes when power fails, but that just changes it to the third case.

The differences between the three variants of `exit` are:

- `_exit` and `_Exit` behave identically in all respects, although technically `_exit` is from UNIX and `_Exit` is from Standard C. From now on, I'll refer only to `_exit`, and you can assume everything I say about it applies to `_Exit` as well.
- `exit` (without the underscore) is also from Standard C. It does everything that `_exit` does and also some higher-level cleanup, including at least calling functions registered with `atexit` (Section 1.3.4) and flushing standard I/O buffers, as though `fflush` or `fclose` were called. (Whether `_exit` flushes the buffers is implementation defined, so don't count on it.)

As `exit` is a superset of `_exit`, everything I say here about the latter applies also to the former, so I'll mostly just talk about `_exit`.

Usually, you call `_exit` in a child process that hasn't done an `exec`, rather than `exit`, as I showed in `execute2` in the previous section, because whatever cleanup processing the child inherited usually should be done only once. But this isn't always true, so you'll have to examine each case to decide what's appropriate. In the most common case, when the child is overwritten by an `exec`, a program starts fresh, so whatever cleanup it does when it exits is OK. Any buffers or `atexit` functions from the parent are long gone.

If you're using the "ec" macros for error checking, as I do in this book, remember that the automatic display of an error message and the function trace-back are done in a function registered with `atexit` that won't be called if you terminate with `_exit` (Section 1.4.2). In `execute2` in the previous section we took care of this by using the `EC_FLUSH` macro in both parent and child, although only the child calls `_exit`.

Here are the synopses for the `exit` functions. Note that there are two different include files involved:

_exit—terminate process without cleanup

```
#include <unistd.h>

void _exit(
    int status          /* exit status */
);
/* Does not return */
```

_Exit—terminate process without cleanup

```
#include <stdlib.h>

void _Exit(
    int status              /* exit status */
);
/* Does not return */
```

exit—terminate process with cleanup

```
#include <stdlib.h>

void exit(
    int status              /* exit status */
);
/* Does not return */
```

`_exit` and the other two variants terminate the process that issued it, with a status code equal to the rightmost (least significant) byte of `status`. There are lots of additional side-effects; the principal ones are listed here, and you can find the complete list in [SUS2002]. Actually, these side-effects apply to process termination in general (except for a crash):

- All open file descriptors are closed.
- As explained in Section 4.3, if the process was the controlling process (the session leader), the session loses its controlling terminal. Also, every process in the session gets a hangup (`SIGHUP`) signal, which, if not caught or ignored, causes them to terminate as well.
- The parent process is notified of the exit status in a manner explained in the next section.
- Child processes aren't affected in any way directly, except that their new parent is a special system process, and if they execute the `getppid` (Section 5.13) system call, they'll get the process-ID of that system process (usually 1).

The exiting process's parent receives its status code via one of the `wait` system calls, which are in the next section. The status code is a number from 0 through 255. By convention, 0 means successful termination, and nonzero means abnormal termination of some sort, as defined by the program doing the exiting. Two standard macros are defined, which I've already used in many examples in this book: `EXIT_SUCCESS` is defined as 0, and `EXIT_FAILURE` is defined as some nonzero value, typically 1. Using the macros instead of integers like 0 and 1 makes programs a bit more readable, which is why we do it that way.

Once a process terminates with an `exit` variant or via a signal, it stops executing but doesn't go away completely until its exit status is reported to its parent, unless the system has determined (in a manner described in the next section) that its parent isn't interested in the status. A terminated process that hasn't yet reported is called a *zombie*. [7]

5.8 `wait`, `waitpid`, and `waitid` System Calls

`wait`, `waitpid`, or `waitid` waits for a child process to change state (stop, continue, or terminate) and then retrieves that process's status. How a process terminates was explained in the previous section; stopped and continued processes were explained in Section 4.3. I'll start with `waitpid` and then cover the other two variants.

Several of the features described in this and the next section are only in X/Open-conforming systems and, as of this writing, are not yet in FreeBSD or Linux; they're marked with the notation "[X/Open]." (X/Open was discussed in Sections 1.2 and 1.5.1; although Linux as of this writing claims to be X/Open when we test it with feature-test macros, it doesn't have the X/Open functionality described in this section.)

waitpid—wait for child process to change state

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

pid_t waitpid(
     pid_t pid,             /* process or process-group ID */
     int *statusp,          /* pointer to status or NULL */
     int options            /* options (see below) */
);
/* Returns process ID or 0 on success or -1 on error (sets errno) */
```

The `pid` argument can be used for one of four things:

`>0` wait for the child process with that process ID

`-1` wait for any child process

7. A dictionary definition of *zombie* that fits very well is "a person of the lowest order of intelligence suggestive of the so-called walking dead." (*Webster's New Collegiate Dictionary.* 6th ed. [Springfield, Mass.: G. & C. Merriam Co., 1960].)

`0` wait for any child process in the same process group as the calling process

`<-1` wait for any child process in the process group whose process-group ID is `-pid`

Process groups were explained in Section 4.3. Typically, a shell that runs each pipeline in its own process group will set `pid` to the negative of that process group's ID when waiting for the pipeline to complete so that any process in the pipeline will cause `waitpid` to return.

When `waitpid` returns because a child matching the `pid` argument changed state, the child's process ID is returned. Zero is returned in one case that will be explained below, under the `WNOHANG` option.

Only direct children—those created with `fork`—can be waited for. Grandchildren may not be, even if their parent (the direct child) has already terminated. As explained in the previous section, orphaned processes are inherited by a special system process, not by the grandparent.

Normally, it's important for a process to wait for every child it creates—otherwise terminated children remain in the system as zombies until the parent terminates, at which point the system process that inherits them will do the wait and clean them out for good. As some processes don't terminate for a long time (months, sometimes), keeping zombies around for that long really clogs up the system tables. If waiting is too troublesome, a process can use signals to eliminate zombies, as explained in the next section.

A child that changes state, called *waitable*, can cause at most one return from `waitpid`. In other words, a waitable child that's waited for is no longer waitable. This means that if one section of the program gets the status and finds out it wasn't from the process it was expecting, there's no way to stuff the result back into the system so some other `waitpid` will get the status for that process later. (The call `waitid` can do this, though—see below.)

If `waitpid` is executed and there's a waitable child that meets the `pid` specification, `waitpid` returns immediately. If there is such a child but it hasn't yet changed state, `waitpid` blocks until there is a suitable waitable child. If there is no child matching `pid` at all, `waitpid` returns `-1` and sets `errno` to `ECHILD`. There might be no child because the `pid` is simply wrong or because the child has already been waited for, in which case it's no longer waitable (but, again, see `waitid`).

If statusp is non-NULL, the integer it points to is set to the status of the child. This is a combination of the child's argument to _exit or exit (if that's how it terminated) and a code that indicates how it terminated or stopped. Macros are supplied for interpreting the integer:

WIFEXITED(status)	true if the child terminated normally (with _exit or exit[8])
WEXITSTATUS(status)	if WIFEXITED, the low-order 8 bits of the argument to _exit or exit
WIFSIGNALED(status)	true if the child terminated abnormally (because of a signal)
WTERMSIG(status)	if WIFSIGNALED, the number of the signal that caused termination
WIFSTOPPED(status)	true if the child stopped; possible only if the WUNTRACED option is set (see below)
WSTOPSIG(status)	if WIFSTOPPED, the number of the signal that caused the child to stop
WIFCONTINUED(status)	true if the process was continued; possible only if the WCONTINUED option is set [X/Open]
WCOREDUMP(status)	true if a memory-dump file (called a "core dump" in UNIX) was produced; useful sometimes for post-mortem analysis (macro is nonstandard, but commonly implemented)

The last argument, options, is one or more of the flags ORed together:

WCONTINUED	Report on continued children, in addition to those that terminated. [X/Open]
WNOHANG	Do not wait for a child if status is not immediately available; return 0 instead of a process ID.
WUNTRACED	Report on stopped children, in addition to those that terminated.[9]

8. Remember that _Exit is the same as _exit, and returning from main is the same as calling exit.
9. Should be called WSTOPPED, but called WUNTRACED for historical reasons.

Here are some example usages of `waitpid`:

```
/* Wait for child pid to terminate and get its status. */
ec_neg1( waitpid(pid, &status, 0) )

/* Wait for any child to terminate, without getting its status. */
ec_neg1( pid = waitpid(-1, NULL, 0) )

/* Report on any child in process group pgid to terminate or stop,
   and get its status. Don't wait if no status is available immediately. */
ec_neg1( pid = waitpid(-pgid, &status, WNOHANG | WUNTRACED) )
```

Here's an example program that creates three child processes, each of which terminates differently. For each termination, the status is reported.

```
int main(void)
{
    pid_t pid;

    /* Case 1: Explicit call to _exit */
    if (fork() == 0) /* child */
        _exit(123);
    /* parent */
    ec_false( wait_and_display() )

    /* Case 2: Termination by kernel */
    if (fork() == 0) { /* child */
        int a, b = 0;

        a = 1 / b;
        _exit(EXIT_SUCCESS);
    }
    /* parent */
    ec_false( wait_and_display() )

    /* Case 3: External signal */
    if ((pid = fork()) == 0) { /* child */
        sleep(100);
        _exit(EXIT_SUCCESS);
    }
    /* parent */
    ec_neg1( kill(pid, SIGHUP) )
    ec_false( wait_and_display() )

    exit(EXIT_SUCCESS);
```

```
EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
EC_CLEANUP_END
}

static bool wait_and_display(void)
{
    pid_t wpid;
    int status;

    ec_neg1( wpid = waitpid(-1, &status, 0) )
    display_status(wpid, status);
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}

void display_status(pid_t pid, int status)
{
    if (pid != 0)
        printf("Process %ld: ", (long)pid);
    if (WIFEXITED(status))
        printf("Exit value %d\n", WEXITSTATUS(status));
    else {
        char *desc;
        char *signame = get_macrostr("signal", WTERMSIG(status), &desc);
        if (desc[0] == '?')
            desc = signame;
        if (signame[0] == '?')
            printf("Signal #%d", WTERMSIG(status));
        else
            printf("%s", desc);
        if (WCOREDUMP(status))
            printf(" - core dumped");
        if (WIFSTOPPED(status))
            printf(" (stopped)");
#if defined(_XOPEN_UNIX) && !defined(LINUX)
        else if (WIFCONTINUED(status))
            printf(" (continued)");
#endif
        printf("\n");
    }
}
```

As we mentioned, the line

```
#if defined(_XOPEN_UNIX) && !defined(LINUX)
```

tests separately for Linux because it claims falsely to be X/Open conforming.

The function `get_macrostr` provides a printable version of a signal number (among other things), and it works a lot like `errsymbol` in Section 1.4.1. The code is on the Web site [AUP2003] if you're interested.

This was the output:

```
Process 9585: Exit value 123
Process 9586: Erroneous arithmetic operation - core dumped
Process 9587: Hangup
```

As the output shows, in case 1, the child terminated itself with a call to `_exit`, passing the exit value 123. In case 2, the child was terminated by a `SIGFPE` (floating-point error) signal when it tried to divide by zero. In case 3, the sleeping child was killed by a `SIGHUP` signal sent from its parent. (The `kill` system call, which sends signals, is in Section 9.1.9.)

We're going to use function `display_status` in the next version of our shell, later in this chapter.

A second variant, the `wait` system call, is shorthand for `waitpid` with a `pid` of `-1` and no `options`:

wait—wait for child process to terminate

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

pid_t wait(
    int *statusp,          /* pointer to status or NULL */
);
/* Returns process ID or -1 on error (sets errno) */
```

Because `wait` and `waitpid` with `pid` set to `-1` wait for any child, they're often inappropriate when a function that's part of a larger program creates a child that it wants to wait for. It might accidentally get a return for some other child entirely, and as a child can be waited for only once, that would deprive some other part of the program of the ability to wait for that child, causing potential confusion. It's much better to use `waitpid` to wait for a specific child, or at least a member of a process group, and for that reason `wait` is seldom used outside of simple programs. It should never be used in library functions that create child processes.

But suppose a process has two children and can't predict which will terminate first. If all it cares about is waiting for both of them to terminate, it can simply wait for either one of them and then, when that call to `waitpid` returns, wait for the other. Or, the parent can proceed with its own work and periodically issue a `waitpid` call for each process with the `WNOHANG` option. But, as we said, it should not issue a blanket `waitpid` call (`pid` argument of `-1`) unless it can guarantee that there are no other children. Generally, in big, complicated programs where parts came from different development groups (e.g., an imaging library, a database interface), there can be no such guarantee. It would be nice if one could pass an array of process IDs to some `wait` variant, but one can't.

The third variant, `waitid`, entered UNIX with SUS1, and so as of this writing isn't in Linux or FreeBSD.[10] It offers one important new feature: You can get the status of a process while keeping it waitable, so no harm is done if you query the wrong process. In addition, it supplies more information than `waitpid` or `wait`. Here's the synopsis:

waitid—wait for child process to change state [X/Open]

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

int waitid(
     idtype_t idtype,       /* interpretation of id */
     id_t id,               /* process or process-group ID */
     siginfo_t *infop,      /* returned info */
     int options            /* options */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

siginfo_t—structure for waitid[11]

```
typedef struct {            /* only waitid-relevant members shown */
     int si_code;           /* code (see below) */
     pid_t si_pid;          /* child process ID */
     uid_t si_uid;          /* real user ID of child process */
     int si_status;         /* exit value or signal */
} siginfo_t;
```

The first argument to `waitid`, `idtype`, is one of:

`P_PID` wait for child with process ID of `id` (like `waitpid` with `pid` > 0)

10. It seems to be only partially implemented on my version of Linux (it seems to work, but only the options for `waitpid` are defined, and there's no `man` page). You'll have to check your own version.
11. This structure has been augmented by the Realtime Signals Extension; see Section 9.5.

`P_PGID` wait for any child in process group `id` (like `waitpid` with `pid` < –1)

`P_ALL` wait for any child (like `waitpid` with `pid` == –1); `id` is ignored

There's no direct equivalent to `waitpid` with `pid` == 0, but the same thing can be done with `P_PID` and `id` equal to the caller's process-group ID.

The `options` argument, as with `waitpid`, is one or more flags ORed together:

`WEXITED` Report on processes that have exited. (Always the case with `waitpid`, which has no such flag.)

`WSTOPPED` Report on stopped children (similar to `waitpid` with the `WUNTRACED` flag).

`WCONTINUED` Report on continued children (same flag as for `waitpid`).

`WNOHANG` Do not wait for a child if status is not immediately available; return 0 (same flag as for `waitpid`).

`WNOWAIT` Leave the reported-on process waitable, so a subsequent `waitid` (or other variant) call can be used on it.

Through its `infop` argument, `waitid` provides more information than does `waitpid`. This argument points to a `siginfo_t` structure; the members relevant to its use with `waitid` are shown in the synopsis. On return, the child's process ID is in the `si_pid` member. You check `si_code` to see why the child terminated; the most commonly occurring codes are:

`CLD_EXITED` Exited with `_exit` or `exit`

`CLD_KILLED` Terminated abnormally (via a signal)

`CLD_DUMPED` Terminated abnormally and created a memory-dump file (called a "core" file in UNIX)

`CLD_STOPPED` Stopped

`CLD_CONTINUED` Continued

If the code is `CLD_EXITED`, the `si_status` member holds the number passed to `_exit` or `exit`. Otherwise, the state change was caused by a signal, and `si_status` is the signal number.

Here are the `wait_and_display` and `display_status` functions from the previous `waitpid` example rewritten for `waitid`:

```
static bool wait_and_display(void)
{
    siginfo_t info;

    ec_neg1( waitid(P_ALL, 0, &info, WEXITED) )
    display_status(&info);
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}

void display_status(siginfo_t *infop)
{
    printf("Process %ld terminated:\n", (long)infop->si_pid);
    printf("\tcode =  %s\n",
      get_macrostr("sigchld-code", infop->si_code, NULL));
    if (infop->si_code == CLD_EXITED)
        printf("\texit value =  %d\n", infop->si_status);
    else
        printf("\tsignal = %s\n",
          get_macrostr("signal", infop->si_status, NULL));
}
```

The output:

```
Process 9580 terminated:
        code =  CLD_EXITED
        exit value =  123
Process 9581 terminated:
        code =  CLD_DUMPED
        signal = SIGFPE
Process 9582 terminated:
        code =  CLD_KILLED
        signal = SIGHUP
```

With `waitid`, the problem of inadvertently getting a report on the wrong child can be averted with an algorithm like this:

1. Execute `waitid` with `P_ALL` and the `WNOWAIT` option.
2. If the returned process ID isn't one this part of the program is concerned with, go back to step 1.

3. If the process ID is of interest, reissue `waitid` for that process ID without the `WNOWAIT` option (or just use `waitpid`), to clear it out so it's no longer waitable.
4. If there are any other un-waited-for children to be waited for, go back to step 1.

The only problem with `waitid` is that it's not available on pre-SUS1 systems, which as of this writing include Linux, FreeBSD, and Darwin. So, the more awkward approaches described earlier in the `waitpid` discussion have to be used.

The behavior of `wait`, `waitpid`, and `waitid` is further affected by what the parent has done with the `SIGCHLD` signal, as explained in the next section.

5.9 Signals, Termination, and Waiting

Most of the details on signals are in Chapter 9, but it makes sense to talk about `SIGCHLD` here. The quick introduction to signals in Section 1.1.3 should be enough to understand this section; if not, skip it until after you've read Chapter 9.

As explained in the previous section, a parent can get the status of a child that changed state (terminated, stopped, or continued) with one of the `wait` variants. If a child isn't waited for at all, it stays waitable (a zombie) until the parent terminates.

I didn't mention it earlier, but a child that changes state normally sends a `SIGCHLD` signal to its parent. Unless the parent has made other arrangements, the default action is for the signal to be ignored, which is why it wasn't important to deal with it in the earlier examples.

The parent can catch the signal if it wants to be notified about changes in the state of a child. Unfortunately, the parent isn't told *which* child caused the signal.[12] If it uses one of the `wait` variants to get the process ID, there's a danger of getting the status of the wrong process, as explained in the previous section, because some other process could have changed state between the sending of the signal and the call to the `wait` variant. The safe thing to do is to use `waitid` (as explained in the previous section) or to use `waitpid` to check all of the possible children, one by one.

12. Unless the advanced features of the Realtime Signals Extension are used, but this extension isn't universally available.

Instead of waiting at all, a parent can prevent an unwaited-for child from becoming a zombie (becoming waitable) by calling `sigaction` with a flag of `SA_NOCLDWAIT` for the `SIGCHLD` signal. Then a child's status is simply tossed out when it terminates, and the parent therefore can't get it (and presumably doesn't care to). If the parent executes `wait` or `waitpid` anyway, regardless of its arguments (even `WNOHANG`), it blocks until *all* children terminate and then returns –1 with `errno` set to `ECHILD`. The parent can still be notified if it catches the `SIGCHLD` signal.

If the parent *explicitly* sets `SIGCHLD` signals to be ignored (say, calling `sigaction` with an action of `SIG_IGN`), as opposed to accepting the default action, it's the same as using the `SA_NOCLDWAIT` flag [X/Open]. But as not all systems support this, it's a good idea to set the `SA_NOCLDWAIT` flag also, which is more widely supported.

5.10 Implementing a Shell (Version 3)

With the `waitpid` version of the `display_status` function from Section 5.8, we can improve on `execute2` from Section 5.6:

```
static void execute3(int argc, char *argv[])
{
    pid_t pid;
    int status;

    switch (pid = fork()) {
    case -1: /* parent (error) */
        EC_FAIL
    case 0: /* child */
        execvp(argv[0], argv);
        EC_FAIL
    default: /* parent */
        ec_neg1( waitpid(pid, &status, 0) )
        display_status(pid, status);
    }
    return;

EC_CLEANUP_BGN
    EC_FLUSH("execute3")
    if (pid == 0)
        _exit(EXIT_FAILURE);
EC_CLEANUP_END
}
```

Here's some output; the command `fpe`, which does nothing more than try to divide by zero, is one written just for this example:

```
$ sh3
@ date
Tue Feb 25 12:49:52 MST 2003
Process 9954: Exit value 0
@ echo The shell is getting better!
The shell is getting better!
Process 9955: Exit value 0
@ fpe
Process 9956: Erroneous arithmetic operation - core dumped
@ EOT $
```

I've shown where I typed a Ctrl-d with the letters *EOT*.

5.11 Getting User and Group IDs

There are a bunch of systems calls to get the real and effective user and group IDs (concepts explained in Section 1.1.5):

getuid—get real user ID

```
#include <unistd.h>

uid_t getuid(void);
/* Returns user ID (no error return) */
```

geteuid—get effective user ID

```
#include <unistd.h>

uid_t geteuid(void);
/* Returns user ID (no error return) */
```

getgid—get real group ID

```
#include <unistd.h>

gid_t getgid(void);
/* Returns group ID (no error return) */
```

getegid—get effective group ID

```
#include <unistd.h>

gid_t getegid(void);
/* Returns group ID (no error return) */
```

A user or group ID is just some number. If you want a name, you have to call `getpwuid` or `getgrgid`, which were described in Section 3.5.2. Here's an example program that displays the real and effective user and group IDs:

```
int main(void)
{
    uid_t uid;
    gid_t gid;
    struct passwd *pwd;
    struct group *grp;

    uid = getuid();
    ec_null( pwd = getpwuid(uid) )
    printf("Real user = %ld (%s)\n", (long)uid, pwd->pw_name);

    uid = geteuid();
    ec_null( pwd = getpwuid(uid) )
    printf("Effective user = %ld (%s)\n", (long)uid, pwd->pw_name);

    gid = getgid();
    ec_null( grp = getgrgid(gid) )
    printf("Real group = %ld (%s)\n", (long)gid, grp->gr_name);

    gid = getegid();
    ec_null( grp = getgrgid(gid) )
    printf("Effective group = %ld (%s)\n", (long)gid, grp->gr_name);

    exit(EXIT_SUCCESS);

EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
EC_CLEANUP_END
}
```

To make the output interesting, I changed the user and group owners of the program file (`uidgrp`) and then turned on the set-ID-on-execution bits for the user and group:

```
$ su
Password:
# chown adm:sys uidgrp
# chmod +s uidgrp
$ uidgrp
Real user = 100 (marc)
Effective user = 4 (adm)
Real group = 14 (sysadmin)
Effective group = 3 (sys)
$
```

5.12 Setting User and Group IDs

The rules for how the real and effective user and group IDs can be changed are complicated, and different depending on whether the process is running as superuser. Before explaining the rules, I need to note that in addition to its current real and effective user and group IDs, the kernel keeps for each process a record of the *original* effective user and group IDs that were set by the last `exec`. These are called the *saved set-user-ID* and *saved set-group-ID*.

Here now are the rules for setting user IDs; group IDs have similar rules:

1. Other than by an `exec` (which can change the saved ID), an ordinary (nonsuperuser) process can never explicitly change the real user ID or saved ID.
2. An ordinary process can change the effective ID to the real or saved ID.
3. A superuser process can change the real and effective user IDs to any user-ID value.
4. When a superuser process changes the real user ID, the saved ID changes to that value as well.

The superuser rules aren't that interesting—anything goes. The two rules for ordinary users essentially mean this: If an `exec` causes the real and effective user-IDs (or group IDs) to be different, the process can go back and forth between them.

Here's one scenario where going back and forth is useful: Imagine a utility `putfile` that transfers files between computers on a network. The utility needs access to a log file that only the administrative user (call it pfadm) can access. The program file is owned by pfadm, with the set-user-ID bit on. It starts running and can access the log file, since its effective user ID is pfadm. Then, to write the real user's files, it sets the effective user ID to the real user ID so that the real user's permissions will control the access. When it's done, it sets the effective user ID back to pfadm to access the log file again. (If the original effective user ID weren't saved, it wouldn't be able to go back.) Note that while it sounds like pfadm is special as far as this utility goes, it is just an ordinary user to the kernel—not a superuser.

Now for the calls themselves. The only useful ones for ordinary-user processes are `seteuid` and `setegid`.

seteuid—set effective user ID

```
#include <unistd.h>

int seteuid(
    uid_t uid              /* effective user ID */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

setegid—set effective group ID

```
#include <unistd.h>

int setegid(
    gid_t gid              /* effective group ID */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

According to the rules we laid out, for an ordinary-user process, the argument must be equal to the real user (or group) ID or to the saved ID. For a superuser, the arguments can be any user (or group) ID.

Superusers can use two additional calls:

setuid—set real, effective, and saved user ID

```
#include <unistd.h>

int setuid(
    uid_t uid              /* real, effective, and saved user ID */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

setgid—set real, effective, and saved group ID

```
#include <unistd.h>

int setgid(
    gid_t gid              /* real, effective, and saved group ID */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

For a superuser, the argument can be any user (or group) ID, and it sets the real, effective, and saved values. For an ordinary user, these two calls act the same as the "e" versions I just showed, which is very confusing, so they should be avoided.

Two more calls, `setreuid` and `setregid`, overlap the functionality of the four calls I already presented and add additional confusion, so they should also be avoided. (They were important in older systems, before `seteuid` and `setegid` were introduced.)

5.13 Getting Process IDs

A process can get its process ID and its parent's process ID with these calls:

getpid—get process ID

```
#include <unistd.h>

pid_t getpid(void);
/* Returns process ID (no error return) */
```

getppid—get parent process ID

```
#include <unistd.h>

pid_t getppid(void);
/* Returns parent process ID (no error return) */
```

I already used `getpid` in an example back in Section 2.4.3.

5.14 `chroot` System Call

chroot—change root directory

```
#include <unistd.h>

int chroot(
    const char *path          /* path name */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

This system call, reserved for superuser processes, changes the root directory of the process, the directory whose path is /. It's nonstandard but implemented on essentially all UNIX-like systems (because it's very old).

There are two uses for `chroot` that come to mind:

- When a command with built-in path names (such as /etc/passwd) is to be run on other than the usual files. A new tree can be constructed wherever convenient; the root can be changed, and then the command can be executed. This is primarily useful in testing new commands before they are installed.
- For added security, such as when a web-server process begins processing an HTML file. It can set the root to the base path, which guarantees that no paths to other files can be accessed. Even the path ".." won't work because at the root it's interpreted as a reference right back to the root.

Once the root is changed, there's no way to get back with `chroot`. However, some systems provide a sister function `fchroot` that takes a file descriptor. (Analogous to the relationship between `fchdir` and `chdir`, as explained in Section 3.6.2.) You can capture a file descriptor open to the current root by opening it for reading, execute `chroot`, and then get back with `fchroot` using the saved file descriptor as an argument.

5.15 Getting and Setting the Priority

As I mentioned back in Section 1.1.6, each process has a *nice valu*e, which is a way for a process to influence the kernel's scheduling priority. Higher values mean that the process wants to be nice—that is, run at a lower priority. Lower values mean less nice—higher priority. To keep the nice values positive, they are offset from a number that varies with the system (strangely, it's referred to as NZERO in UNIX documentation); a typical value is 20. A process starts out with 20 and can be as nice as 39 or as not-nice as 0.

To be nice or not-nice, a process executes the `nice` system call:

nice—change nice value

```
#include <unistd.h>

int nice(
    int incr                /* increment */
);
/* Returns old nice value - NZERO or -1 on error (sets errno) */
```

The `nice` system call adds `incr` to the nice value. The resulting nice value must be between 0 and 39, inclusive; if an invalid value results, the nearest valid value is used. Only the superuser can lower the nice value, getting better-than-average service.

`nice` actually returns the new nice value minus 20, so the returned value is in the range –20 through 19 if NZERO is 20. However, the returned value is rarely of much use. This is just as well because a new nice value of 19 is indistinguishable from an error return (19 – 20 = –1). That this bug has remained unfixed, if not unnoticed, for so many years is indicative of how little most UNIX system programmers care about error returns from minor system calls like `nice`.[13]

13. Finally fixed in the latest version of [SUS2002]. You can set `errno` to zero before the call and then test it afterwards.

Most UNIX users are familiar with the nice *command,* which runs a program at a lower priority (or a higher one if run by a superuser). Here's our version:

```
#define USAGE "usage: aupnice [-num] command\n"

int main(int argc, char *argv[])
{
    int incr, cmdarg;
    char *cmdname, *cmdpath;

    if (argc < 2) {
        fprintf(stderr, USAGE);
        exit(EXIT_FAILURE);
    }
    if (argv[1][0] == '-') {
        incr = atoi(&argv[1][1]);
        cmdarg = 2;
    }
    else {
        incr = 10;
        cmdarg = 1;
    }
    if (cmdarg >= argc) {
        fprintf(stderr, USAGE);
        exit(EXIT_FAILURE);
    }
    (void)nice(incr);
    cmdname = strchr(argv[cmdarg], '/');
    if (cmdname == NULL)
        cmdname = argv[cmdarg];
    else
        cmdname++;
    cmdpath = argv[cmdarg];
    argv[cmdarg] = cmdname;
    execvp(cmdpath, &argv[cmdarg]);
    EC_FAIL

EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
EC_CLEANUP_END
}
```

Note how we reuse the incoming `argv` in the call to `execvp`. The variable `cmdarg` holds the subscript of the command path (either 1 or 2, depending on whether an increment was specified). That part of `argv` starting with `cmdarg` is the part to be passed on. You might want to contrast this with the more extensive

manipulation needed for `execvp2` in Section 5.3. There we had to *insert* an argument, which required us to recopy the entire array.

The first string in the vector is supposed to be just the command name, not the whole path, so we had to strip off all but the last component of the path, in case the command was executed with a pathname containing slashes. The first argument to `execvp`, though, has to be whatever was typed on the `aupnice` command line.

Note also that we used `execvp` without a `fork`. Having changed the priority, there's no need to keep a parent alive just to wait around.

Here's `aupnice` in action on Solaris. For the example command, it uses `ps` reporting on itself. Note that the internal priority (whatever that is) changed from 58 to 28 when the nice value was increased by 10 (`aupnice`'s default).

```
$ ps -ac
   PID  CLS PRI TTY      TIME CMD
 10460   TS  58 pts/2    0:00 ps
$ aupnice ps -ac
   PID  CLS PRI TTY      TIME CMD
 10461   TS  28 pts/2    0:00 ps
```

There are some newer and fancier calls called `getpriority` and `setpriority` that you can use to manipulate the nice value; see [SUS2002] or your system's documentation for details.

5.16 Process Limits

The kernel enforces various limits on process resources, such as the maximum file size and the maximum stack size. Usually, when a limit is reached, whatever function (e.g., `write`, `malloc`) was the cause returns an error, or a signal is generated, such as `SIGSEGV` in the case of stack overflow (Chapter 9).[14] There are seven standard resources, listed below, whose limits can be set or gotten, and implementations may define others.

14. If `SIGSEGV` is caught, how can the signal handler execute when there's no stack? You can set up an alternate stack, as explained in Section 9.3. If you haven't done so, the signal is set back to its default behavior, which is to terminate the process.

For each resource, there's a *maximum* (or *hard*) limit, and a *current* (or *soft*) limit. The current limit is what's effective, and an ordinary process is allowed to set it to any reasonable number up to the maximum. An ordinary process can also irreversibly lower (but not raise) the maximum to whatever the current limit is. A superuser process can set either limit to whatever the kernel will support.

Here are the system calls to get and set the limits:

getrlimit—get resource limits

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

int getrlimit(
     int resource,              /* resource */
     struct rlimit *rlp         /* returned limits */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

setrlimit—set resource limits

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

int setrlimit(
     int resource,              /* resource */
     const struct rlimit *rlp   /* limits to set */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

struct rlimit—structure for getrlimit and setrlimit

```
struct rlimit {
     rlim_t rlim_cur;           /* current (soft) limit */
     rlim_t rlim_max;           /* maximum (hard) limit */
};
```

Each call is for a single resource; you get its limits with `getrlimit`, and set them, under the rules given previously, with `setrlimit`. The standard resources are:

`RLIMIT_CORE` Maximum size of a core (memory-dump) file in bytes; zero means no file is written at all. There's no error reported if it's exceeded—writing just stops at that size, probably leaving a useless file.

`RLIMIT_CPU` Maximum CPU time in seconds. Exceeding it generates a `SIGXCPU`, which by default terminates the process. The standard doesn't define what happens if you catch, ignore, or block this signal—the process may be terminated, as there's no way to continue without accumulating more CPU time.

RLIMIT_DATA	Maximum size of data segment, in bytes. Exceeding it causes the memory-allocation function (e.g., malloc) to fail.
RLIMIT_FSIZE	Maximum file size in bytes. Exceeding it generates a SIGXFSZ; if that signal is caught, ignored, or blocked, the offending function fails.
RLIMIT_NOFILE	Maximum number of file descriptors that a process may use.[15] Exceeding it causes the offending function to fail.
RLIMIT_STACK	Maximum stack size in bytes. Generates a SIGSEGV if it's exceeded.
RLIMIT_AS	Maximum total memory size, including the data segment, the stack, and anything mapped in with mmap (Section 7.14). Exceeding it causes the offending function to fail or, if it's the stack, the effect described for RLIMIT_STACK.

Process limits are preserved across an exec and inherited by the child after a fork, so if they're set once at login time, they'll affect all processes descended from the login shell. But, like cd, the command to set them has to be built into the shell—executing it in a child of the shell would be ineffective. Most shells have a ulimit command that sets the file-size limit, and other shells have a more general command called perhaps limit, limits, or plimit.[16]

There are some special macros for the rlim_cur and rlim_max members of an rlimit structure, in addition to actual numbers: RLIM_SAVED_CUR, RLIM_SAVED_MAX, and RLIM_INFINITY.

When you call getrlimit, the actual numbers are returned if they fit into an rlim_t, which is an unsigned type whose width isn't specified by the standard. If a number won't fit, the member is set to RLIM_SAVED_MAX if it's equal to the maximum. Otherwise it's set to RLIM_SAVED_CUR, which means "the limit is set to what the limit is set to." If a member is set to RLIM_INFINITY, it means there is no limit.

15. Technically, it's one greater than the maximum file-descriptor number. All the potential file descriptors count toward the limit even if they're not opened.

16. Don't confuse these *process* limits with *user* limits, such as user's disk quota. User limits are queried and set by nonstandard commands such as quota, implemented in terms of a nonstandard system call such as quotactl or ioctl.

When you call `setrlimit`, if a member is `RLIM_SAVED_CUR`, it's set to the current limit; if it's `RLIM_SAVED_MAX`, it's set to the maximum limit; if it's `RLIM_INFINITY`, no limit will be enforced. Otherwise, the limit is set to whatever number you've used. However, you can use `RLIM_SAVED_CUR` or `RLIM_SAVED_MAX` only if a call to `getrlimit` returned those values; otherwise, you have to use actual numbers. To say it differently: You can use those macros only when you're forced to because `rlim_t` won't hold the number.

The `RLIM_SAVED_CUR` and `RLIM_SAVED_MAX` rigmarole is really just this: You can manipulate the limits to a degree even if you can't get at their actual values.

If an implementation will never have a limit that won't fit in an `rlim_t`, the `RLIM_SAVED_CUR` or `RLIM_SAVED_MAX` macros will never be used, so they're allowed to have the same value as `RLIM_INFINITY`. We'll see the impact of this in the following example, which displays the limits, sets the file limit to a ridiculously low number, and then exceeds it, causing the program to be terminated by a `SIGXFSZ`:

```
int main(void)
{
    struct rlimit r;
    int fd;
    char buf[500] = { 0 };

    if (sizeof(rlim_t) > sizeof(long long))
        printf("Warning: rlim_t > long long; results may be wrong\n");
    ec_false( showlimit(RLIMIT_CORE, "RLIMIT_CORE") )
    ec_false( showlimit(RLIMIT_CPU, "RLIMIT_CPU") )
    ec_false( showlimit(RLIMIT_DATA, "RLIMIT_DATA") )
    ec_false( showlimit(RLIMIT_FSIZE, "RLIMIT_FSIZE") )
    ec_false( showlimit(RLIMIT_NOFILE, "RLIMIT_NOFILE") )
    ec_false( showlimit(RLIMIT_STACK, "RLIMIT_STACK") )
#ifndef FREEBSD
    ec_false( showlimit(RLIMIT_AS, "RLIMIT_AS") )
#endif
    ec_neg1( getrlimit(RLIMIT_FSIZE, &r) )
    r.rlim_cur = 500;
    ec_neg1( setrlimit(RLIMIT_FSIZE, &r) )
    ec_neg1( fd = open("tmp", O_WRONLY | O_CREAT | O_TRUNC, PERM_FILE) )
    ec_neg1( write(fd, buf, sizeof(buf)) )
    ec_neg1( write(fd, buf, sizeof(buf)) )
    printf("Wrote two buffers! (?)\n");
    exit(EXIT_SUCCESS);

EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
```

```
EC_CLEANUP_END
}

static bool showlimit(int resource, const char *name)
{
    struct rlimit r;

    ec_neg1( getrlimit(resource, &r) )
    printf("%s: ", name);
    printf("rlim_cur = ");
    showvalue(r.rlim_cur);
    printf("; rlim_max = ");
    showvalue(r.rlim_max);
    printf("\n");
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}

static void showvalue(rlim_t lim)
{
    /*
        All macros may equal RLIM_INFINITY; that test
        must be first; can't use switch statement.
    */
    if (lim == RLIM_INFINITY)
        printf("RLIM_INFINITY");
#ifndef BSD_DERIVED
    else if (lim == RLIM_SAVED_CUR)
        printf("RLIM_SAVED_CUR");
    else if (lim == RLIM_SAVED_MAX)
        printf("RLIM_SAVED_MAX");
#endif
    else
        printf("%llu", (unsigned long long)lim);
}
```

FreeBSD and Darwin don't implement the whole thing, as you can see from the code.[17]

Here's what I got on our four test systems, starting with Solaris:

```
RLIMIT_CORE: rlim_cur = RLIM_INFINITY; rlim_max = RLIM_INFINITY
RLIMIT_CPU: rlim_cur = RLIM_INFINITY; rlim_max = RLIM_INFINITY
```

17. Actually, these calls originated with 4.2BSD, but the BSD-derived systems haven't picked up some of the more recent POSIX extensions.

```
RLIMIT_DATA: rlim_cur = RLIM_INFINITY; rlim_max = RLIM_INFINITY
RLIMIT_FSIZE: rlim_cur = RLIM_INFINITY; rlim_max = RLIM_INFINITY
RLIMIT_NOFILE: rlim_cur = 256; rlim_max = 1024
RLIMIT_STACK: rlim_cur = 8683520; rlim_max = 133464064
RLIMIT_AS: rlim_cur = RLIM_INFINITY; rlim_max = RLIM_INFINITY
File Size Limit Exceeded - core dumped
```

Linux:

```
RLIMIT_CORE: rlim_cur = 0; rlim_max = RLIM_INFINITY
RLIMIT_CPU: rlim_cur = RLIM_INFINITY; rlim_max = RLIM_INFINITY
RLIMIT_DATA: rlim_cur = RLIM_INFINITY; rlim_max = RLIM_INFINITY
RLIMIT_FSIZE: rlim_cur = RLIM_INFINITY; rlim_max = RLIM_INFINITY
RLIMIT_NOFILE: rlim_cur = 1024; rlim_max = 1024
RLIMIT_STACK: rlim_cur = RLIM_INFINITY; rlim_max = RLIM_INFINITY
RLIMIT_AS: rlim_cur = RLIM_INFINITY; rlim_max = RLIM_INFINITY
File size limit exceeded
```

FreeBSD:

```
RLIMIT_CORE: rlim_cur = RLIM_INFINITY; rlim_max = RLIM_INFINITY
RLIMIT_CPU: rlim_cur = RLIM_INFINITY; rlim_max = RLIM_INFINITY
RLIMIT_DATA: rlim_cur = 536870912; rlim_max = 536870912
RLIMIT_FSIZE: rlim_cur = RLIM_INFINITY; rlim_max = RLIM_INFINITY
RLIMIT_NOFILE: rlim_cur = 957; rlim_max = 957
RLIMIT_STACK: rlim_cur = 67108864; rlim_max = 67108864
Filesize limit exceeded
```

Darwin:

```
RLIMIT_CORE: rlim_cur = 0; rlim_max = RLIM_INFINITY
RLIMIT_CPU: rlim_cur = RLIM_INFINITY; rlim_max = RLIM_INFINITY
RLIMIT_DATA: rlim_cur = 6291456; rlim_max = RLIM_INFINITY
RLIMIT_FSIZE: rlim_cur = RLIM_INFINITY; rlim_max = RLIM_INFINITY
RLIMIT_NOFILE: rlim_cur = 256; rlim_max = RLIM_INFINITY
RLIMIT_STACK: rlim_cur = 524288; rlim_max = 67108864
Filesize limit exceeded
```

There's a much older system call that's less functional and has problems with its return value:

ulimit—get and set process limits

```
#include <ulimit.h>

long ulimit(
     int cmd,               /* command */
     ...                    /* optional argument */
);
/* Returns limit or -1 with errno changed on error (sets errno) */
```

The cmd argument is one of:

UL_GETFSIZE Get the current file-size limit; equivalent to getrlimit with RLIMIT_FSIZE.

UL_SETFSIZE Set the file-size limit to the second long argument; it may be increased only by a superuser process. Equivalent to setrlimit with RLIMIT_FSIZE where rlim_max is adjusted and rlim_cur is adjusted to match.

The sizes set or gotten are in units of 512 bytes. Because the sizes can get big, the returned value might be negative, and even –1 is possible. So, the way to check for an error is to set errno to zero before the call. If it returns –1, it's an error only if errno changed.

There are some more peculiarities with ulimit, but there's no point going into them. It's basically a mess, and you should use getrlimit and setrlimit instead.

The getrusage system call gets information about resource usage by a process or by its terminated and waited-for children:

getrusage—get resource usage

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

int getrusage(
     int who,                    /* RUSAGE_SELF or RUSAGE_CHILDREN */
     struct rusage *r_usage      /* returned usage information */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

struct rusage—structure for getrusage

```
struct rusage {
     struct timeval ru_utime;   /* user time used */
     struct timeval ru_stime;   /* system time used */
     /* following members are nonstandard */
     long ru_maxrss;            /* maximum resident set size */
     long ru_ixrss;             /* integral shared memory size */
     long ru_idrss;             /* integral unshared data size */
     long ru_isrss;             /* integral unshared stack size */
     long ru_minflt;            /* page reclaims */
     long ru_majflt;            /* page faults */
     long ru_nswap;             /* swaps */
     long ru_inblock;           /* block input operations */
     long ru_oublock;           /* block output operations */
     long ru_msgsnd;            /* messages sent */
     long ru_msgrcv;            /* messages received */
     long ru_nsignals;          /* signals received */
     long ru_nvcsw;             /* voluntary context switches */
     long ru_nivcsw;            /* involuntary context switches */
};
```

[SUS2002] only specifies the first two members. Newer BSD systems, including FreeBSD, support them all. Solaris 8 (on Intel, anyway) supports only the standard members; Linux 2.4 supports the standard members plus `ru_minflt`, `ru_majflt`, and `ru_nswap`. For details about what the members mean, check your system's `man` page for `getrusage`; for the whole story, run it on a BSD-based system if you can.

The two times returned are similar to what `times` returns (Section 1.7.2), but to a higher resolution, since a `timeval` (Section 1.7.1) is in microseconds.

5.17 Introduction to Threads

This section gives a very brief introduction to threads, just enough to show some interesting example programs. My goals are to explain what they are, help you decide if your application design can benefit from them, and caution you about the hazards of using them if you're not careful.

For the whole truth about threads (there are about a hundred system calls to manage them), you'll have to pick up a book on the subject, such as [Nor1997] or [But1997].

5.17.1 Thread Creation

In our examples up to now, the processes we created (with `fork`) had only one flow of control, or *thread,* in them. Execution proceeded from instruction to instruction, in a single sequence, and the stack, global data, and system resources got modified by the various instructions, some of which may have executed system calls.

With the POSIX Threads feature of UNIX, a process can have multiple threads, each of which has its own flow of control (its own instruction counter and CPU clock) and its own stack. Essentially everything else about the process, including global data and resources such as open files or the current directory, is shared.[18] The following code shows what I mean.

18. A thread can also arrange to have some thread-specific data; see [SUS2002] or one of the referenced books for details.

```
static long x = 0;

static void *thread_func(void *arg)
{
    while (true) {
        printf("Thread 2 says %ld\n", ++x);
        sleep(1);
    }
}

int main(void)
{
    pthread_t tid;

    ec_rv( pthread_create(&tid, NULL, thread_func, NULL) )
    while (x < 10) {
        printf("Thread 1 says %ld\n", ++x);
        sleep(2);
    }
    return EXIT_SUCCESS;

EC_CLEANUP_BGN
    return EXIT_FAILURE;
EC_CLEANUP_END
}
```

The sequence in which the threads print is unpredictable; this is the output we happened to get:

```
Thread 2 says 1
Thread 1 says 2
Thread 2 says 3
Thread 1 says 4
Thread 2 says 5
Thread 2 says 6
Thread 1 says 7
Thread 2 says 8
Thread 2 says 9
Thread 1 says 10
Thread 2 says 11
Thread 2 says 12
```

So you can see that both threads—the one in `main` and the one in `thread_func`—accessed the same global variable, `x`. There are some problems with this that I'll address in Section 5.17.3.

The initial thread—the one containing `main`—kicked off the second thread with the `pthread_create` system call:

pthread_create—create thread

```
#include <pthread.h>

int pthread_create(
     pthread_t *thread_id,          /* new thread's ID */
     const pthread_attr_t *attr,    /* attributes (or NULL) */
     void *(*start_fcn)(void *),    /* starting function */
     void *arg                      /* arg to starting function */
);
/* Returns 0 on success, error number on error */
```

The new thread starts with a call to the start function specified in the `pthread_create` call, which must have this prototype:

pthread starting function

```
void *start_fcn(
     void *arg
);
/* Returns exit status */
```

Whatever you pass as the fourth argument to `pthread_create` is passed directly to the starting function. It's a `void` pointer for generality, and often it will indeed be a pointer to some data. Since the threads share the same address space, the pointer is valid for both threads. If you want, you can also pass in integer data, but you have to cast it to a `void` pointer. And, to be safe, you should check (with an `assert`, say) that the integer type you're using fits in a `void` pointer, with a line like this that you'll see in some of the examples that follow:

```
assert(sizeof(long) <= sizeof(void *));
```

The attributes that affect the new thread are not just some flags but rather an object of type `pthread_attr_t` that you have to set with calls like `pthread_attr_setscope` and `pthread_attr_setstacksize`, which I won't go into here. (But perhaps you're beginning to appreciate why there are a hundred thread-related calls.) In our examples we'll use just default attributes, so our second argument to `pthread_create` will be `NULL`.

The "pthread" functions uniformly return 0 on success and an error code on failure, rather than setting `errno`. We have a special "ec" macro for this, `ec_rv`, which takes the error code, if any, and treats it as if it were an `errno` value, which it is.[19]

19. The various functions in the error-checking package were modified to be thread-safe, but the code for that isn't shown in this book. You'll see it on the Web site, however.

5.17.2 Waiting for a Thread to Terminate

A thread can wait for another thread to terminate and get its exit status with `pthread_join` (analogous to `wait` and its variants).

pthread_join—wait for thread to terminate

```
#include <pthread.h>

int pthread_join(
    pthread_t thread_id,   /* ID of thread to join */
    void **status_ptr      /* returned exit status (if not NULL arg) */
);
/* Returns 0 on success, error number on error */
```

Here's my little example modified so that Thread 1 passes a limit to Thread 2, which then reports the value of `x` back to Thread 1:

```
static long x = 0;

static void *thread_func(void *arg)
{
    while (x < (long)arg) {
        printf("Thread 2 says %ld\n", ++x);
        sleep(1);
    }
    return (void *)x;
}

int main(void)
{
    pthread_t tid;
    void *status;

    assert(sizeof(long) <= sizeof(void *));
    ec_rv( pthread_create(&tid, NULL, thread_func, (void *)6) )
    while (x < 10) {
        printf("Thread 1 says %ld\n", ++x);
        sleep(2);
    }
    ec_rv( pthread_join(tid, &status) )
    printf("Thread 2's exit status is %ld\n", (long)status);
    return EXIT_SUCCESS;

EC_CLEANUP_BGN
    return EXIT_FAILURE;
EC_CLEANUP_END
}
```

The output:

```
Thread 1 says 1
Thread 2 says 2
Thread 2 says 3
Thread 1 says 4
Thread 2 says 5
Thread 2 says 6
Thread 1 says 7
Thread 1 says 8
Thread 1 says 9
Thread 1 says 10
Thread 2's exit status is 7
```

5.17.3 Thread Synchronization (Mutexes)

As we're going to see in the next few chapters, the challenge on UNIX of working with multiple processes (perhaps across a network) is allowing them to *share* data. With threads it's the complete opposite—the challenge is to keep their natural sharing of data *separated*. In fact, the two thread examples I've shown are defective in that both threads might simultaneously access the same data, the variable `x`. While it might seem that a simple increment operator is an atomic operation, there's no guarantee that it is. It's actually possible for Thread 1 to update half of a 32-bit `x` while Thread 2 reads the full 32 bits, getting a mishmash instead of a valid integer.[20] With more complicated shared data structures—a much more realistic situation—the problem is much worse. We want the access to shared data to be atomic. Two examples of what this means are:

- If updating a data structure leaves it in a temporarily inconsistent state, no thread other than the one updating sees it in that state, and
- If a thread has to read data, compute results, and write them back, no other thread will modify the data until the entire sequence is complete. Otherwise, the other thread's modification would be lost.

These requirements are provided for by some simple system calls that implement mutual-exclusion objects, called *mutexes* for short. Threads use them to protect critical sections where another thread may otherwise see inconsistent data or interfere with updating. The principal mutex system calls are `pthread_mutex_lock` and `pthread_mutex_unlock`:

20. And that's only one thing that can go wrong. Another is that compiler optimization might leave the integer in a register. You really can't ever let threads simultaneously access data without protection.

pthread_mutex_lock—lock mutex

```
#include <pthread.h>

int pthread_mutex_lock(
    pthread_mutex_t *mutex    /* mutex to lock */
);
/* Returns 0 on success, error number on error */
```

pthread_mutex_unlock—unlock mutex

```
#include <pthread.h>

int pthread_mutex_unlock(
    pthread_mutex_t *mutex    /* mutex to unlock */
);
/* Returns 0 on success, error number on error */
```

Mutual exclusion works like this: If a mutex is already locked, `pthread_mutex_lock` blocks until it is unlocked.

The simplest way to get a suitable mutex variable is just to declare it with an initializer, like this:

```
static pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;
```

It can be local to the file (`static`) or shared between files (`extern`). You can also have mutexes on the stack (automatic variables) or allocated dynamically, but then you need to initialize them with a call to `pthread_mutex_init`, which I won't detail here.[21]

Now we can modify our example to adequately protect the shared data:

```
static pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;
static long x = 0;

static void *thread_func(void *arg)
{
    bool done;

    while (true) {
        ec_rv( pthread_mutex_lock(&mtx) )
        done = x >= (long)arg;
        ec_rv( pthread_mutex_unlock(&mtx) )
```

21. You're not supposed to initialize an automatic mutex variable with the initializer `PTHREAD_MUTEX_INITIALIZER`, even though the compiler will let you, because on some systems that might invoke a function that's not thread safe. In C++, you can get into similar trouble if you use `PTHREAD_MUTEX_INITIALIZER` to initialize a static internal mutex, since C++ might delay the initialization until the function is called.

```
        if (done)
            break;
        ec_rv( pthread_mutex_lock(&mtx) )
        printf("Thread 2 says %ld\n", ++x);
        ec_rv( pthread_mutex_unlock(&mtx) )
        sleep(1);
    }
    return (void *)x;

EC_CLEANUP_BGN
    EC_FLUSH("thread_func")
    return NULL;
EC_CLEANUP_END
}

int main(void)
{
    pthread_t tid;
    void *status;
    bool done;

    assert(sizeof(long) <= sizeof(void *));
    ec_rv( pthread_create(&tid, NULL, thread_func, (void *)6) )
    while (true) {
        ec_rv( pthread_mutex_lock(&mtx) )
        done = x >= 10;
        ec_rv( pthread_mutex_unlock(&mtx) )
        if (done)
            break;
        ec_rv( pthread_mutex_lock(&mtx) )
        printf("Thread 1 says %ld\n", ++x);
        ec_rv( pthread_mutex_unlock(&mtx) )
        sleep(2);
    }
    ec_rv( pthread_join(tid, &status) )
    printf("Thread 2's exit status is %ld\n", (long)status);
    return EXIT_SUCCESS;

EC_CLEANUP_BGN
    return EXIT_FAILURE;
EC_CLEANUP_END
}
```

Note that we had to pull the conditionals out of the `while` expressions so that we could surround them with the locking and unlocking calls. We certainly don't want to protect the whole loop, for that would destroy the concurrency. It requires some care to protect enough to be safe but not so much to hurt performance. Unfortunately, while you may be able to detect a performance problem with the

right kind of testing, it's much harder to test whether you've prevented all race conditions.

We still have a defect: Suppose in `pthread_func` the call to `pthread_mutex_unlock` fails, causing the thread to exit. This might leave the mutex locked, causing the initial thread to block forever in one of its calls to `pthread_mutex_lock`. A potential solution is to make sure the mutex is unlocked in `pthread_func`'s cleanup code, with another call to `pthread_mutex_unlock`, like this:

```
EC_CLEANUP_BGN
    (void)pthread_mutex_unlock(&mtx);
    EC_FLUSH("thread_func")
    return NULL;
EC_CLEANUP_END
```

We're assuming that if the first call to `pthread_mutex_unlock` failed, the second one might succeed, which seems like a stretch. Since there is really no way either `pthread_mutex_lock` or `pthread_mutex_unlock` can fail as long as the mutex has a valid value, probably the easiest, safest, and clearest way to proceed is to check the error return from these functions during debugging, but not in the production program. Or, you might decide that for your application it's better to leave the error checks in, as long as you get a report of the error, so you don't waste time trying to find out why the application stalled. This is another example of why threads can be very tricky to use correctly. You may find that you spend 5% of your time implementing the threads and 95% of your time ensuring that you've done it right.

The proper way to proceed is to encapsulate the access to shared data with a collection of functions (or with a class, if you're using an object-oriented language like C++) with the mutexes in the access functions. Don't spread the locking and unlocking calls willy-nilly throughout your program, as I did.

So I'll show the example rewritten yet again, this time with all access to `x` going through a single function, `get_and_incr_x`, which does all the locking and unlocking. Both `x` and the mutex have been moved inside the function. Notice how much more readable this version is than the previous one. More readable almost always implies more reliable!

```
static long get_and_incr_x(long incr)
{
    static long x = 0;
    static pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;
    long rtn;
```

```
    ec_rv( pthread_mutex_lock(&mtx) )
    rtn = x += incr;
    ec_rv( pthread_mutex_unlock(&mtx) )
    return rtn;

EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
EC_CLEANUP_END
}

static void *thread_func(void *arg)
{
    while (get_and_incr_x(0) < (long)arg) {
        printf("Thread 2 says %ld\n", get_and_incr_x(1));
        sleep(1);
    }
    return (void *)get_and_incr_x(0);
}

int main(void)
{
    pthread_t tid;
    void *status;

    assert(sizeof(long) <= sizeof(void *));
    ec_rv( pthread_create(&tid, NULL, thread_func, (void *)6) )
    while (get_and_incr_x(0) < 10) {
        printf("Thread 1 says %ld\n", get_and_incr_x(1));
        sleep(2);
    }
    ec_rv( pthread_join(tid, &status) )
    printf("Thread 2's exit status is %ld\n", (long)status);
    return EXIT_SUCCESS;

EC_CLEANUP_BGN
    return EXIT_FAILURE;
EC_CLEANUP_END
}
```

This time I decided to make locking or unlocking errors fatal, which is another possible choice. My point certainly isn't that this is what you should be doing in all cases. Rather, the point is that there are many ways to handle these kinds of errors, and what's best depends on the needs of the specific application.

It's important to understand why these lines in function `get_and_incr_x` work:

```
ec_rv( pthread_mutex_lock(&mtx) )
rtn = x += incr;
ec_rv( pthread_mutex_unlock(&mtx) )
```

It's the access to `x`, which, as a global variable, is shared between the threads that we need to protect. The variables `rtn` and `incr` are on a stack unique to each thread (that is, each thread has its own copies) and need no protection.

I have one more observation about this example which I'll defer to Exercise 5.11.

There are three other kinds of thread-synchronization objects that you can read about in [SUS2002] or one of the books referenced at the start of Section 5.17:

- *Read-write locks* are like mutexes, but they distinguish between locking against reading the data and locking against writing, to gain additional concurrency. See Section 7.11.4 for more about read-write locks in general.
- *Spin locks* are also like mutexes, but they're faster and intended for short durations. They're usually implemented by testing the lock in a CPU loop instead by blocking the thread.
- *Barriers* are synchronization points at which one or more threads wait, ensuring that they have all completed some task before any one is allowed to continue.

5.17.4 Condition Variables

Suppose Thread A is doing some work (reading data from a network connection, say) and adding items to a queue, and Thread B is taking items from the queue and doing some additional work on them (updating a database, say). Using a mutex *M* to control access to the queue, you could organize the threads like this:

Thread A	*Thread B*
1. Read data [B]	1. Lock *M* [b]
2. Lock *M* [b]	2. If item on queue, remove it and update database
3. Put item on queue	3. Unlock *M*
4. Unlock *M*	4. Goto step 1
5. Goto step 1	

(The notation [B] means an indeterminate block, and [b] means a short-term block.)

This works OK, but Thread B wastes a lot of CPU time when the queue is empty, as it keeps checking while it loops and loops. You could slow it down a bit like this:

Thread A	*Thread B*
1. Read data [B]	1. Lock *M*
2. Lock *M* [b]	2. If item on queue, remove it and update database
3. Put item on queue	3. Unlock *M* [b]
4. Unlock *M*	4. Sleep for 1 sec. [b]
5. Goto step 1	5. Goto step 1

But this isn't much of an improvement: Thread B is now less responsive because it may be sleeping when there's work to do, and it still wastes CPU time because the queue may be empty when it wakes up. What we want to do is have Thread A signal Thread B when it puts an item on the queue and have Thread B block until it gets the signal. Let's try it with another mutex *Q* to represent the concept "queue is nonempty":

Thread A	*Thread B*
1. Read data [B]	1. Lock *M* [b]
2. Lock *M* [b]	2. If item on queue, remove it and update database
3. Put item on queue	3. Unlock *M*
4. Unlock *M*	4. Lock *Q* [B]
5. Unlock *Q*	5. Goto step 1
6. Goto step 1	

Thread B's attempt to lock *Q* (step 4) is an indeterminate block because *Q* becomes unlocked only when Thread A returns from its read (step 1).

Now the problem is that Thread A may try to unlock *Q* when Thread B hasn't got it locked, which would be a lost signal—the attempt to unlock isn't somehow remembered for the next attempt to lock (i.e., mutexes are not counting semaphores). So if this happens, Thread B will be held up waiting for the lock on *Q*, and the lock won't be released until the next time data is read, if ever. Meanwhile, the item on the queue goes unprocessed by Thread B.

A more concrete problem with *Q* is that only the thread that locks a mutex can unlock it. So, A's step 5 would result in an error.

We could try to replace *Q* with a counting semaphore (Section 7.8), but a better choice is to use another kind of signaling mechanism that POSIX Threads provides: *condition variables*. Here's how they're used in this example:

Thread A	*Thread B*
1. Read data [B]	1. Lock *M* [b]
2. Lock *M* [b]	2. while (queue is empty) {
3. Put item on queue	cond_wait(*C*, *M*) [B]
4. cond_signal(*C*)	}
5. Unlock *M*	3. Remove item; update database
6. Goto step 1	4. Unlock *M*
	5. Goto step 1

The second mutex, *Q*, is gone, and now there's a condition variable, *C*. In Thread B, cond_wait waits until condition *C* is signaled, which it is in Thread A's step 4. But cond_wait has a special interaction with mutex *M* (its second argument): The mutex must be locked when cond_wait is called. *It is unlocked during the waiting, and then relocked automatically when cond_wait returns.* That way there's no possibility of a missed signal or of deadlock, both of which were defects in our earlier attempts. And, all the code in Thread B's steps 2 and 3 is executed with *M* locked, as it should be.

Why is cond_wait in a loop that tests whether the queue is empty, when condition *C* gets signaled (by Thread A) only when the queue is nonempty? It's because cond_wait, like any UNIX blocking system call, is subject to being interrupted (e.g., by the arrival of a traditional UNIX signal, such as `SIGINT`), in which case there might be a return with the queue still empty. A loop that tests the predicate (empty queue, in this case) ensures that we'll go right back into cond_wait on such a spurious return. Because spurious returns are infrequent, the wasted CPU time is inconsequential.

Another reason for checking the queue before calling cond_wait is that cond_signal might have been called well before Thread B calls cond_wait, and a con_signal with no thread waiting is discarded (*not* held pending). So, it's important not to call cond_wait unless waiting is necessary, or the wait may be forever, even though the queue is nonempty.

Thus, to signal and wait for a condition, you need three things: a *condition variable*, a *mutex*, and a *predicate*. The first two are special data types, but the

predicate is just some ordinary code that makes sense for your program—it's the concrete details of what the condition variable represents abstractly.

Here are the synopses for the actual `pthread_cond_signal` and `pthread_cond_wait` system calls:

pthread_cond_signal—signal condition

```
#include <pthread.h>

int pthread_cond_signal(
    pthread_cond_t *cond      /* condition variable */
);
/* Returns 0 on success, error number on error */
```

pthread_cond_wait—wait for condition

```
#include <pthread.h>

int pthread_cond_wait(
    pthread_cond_t *cond,     /* condition variable */
    pthread_mutex_t *mutex    /* mutex */
);
/* Returns 0 on success, error number on error */
```

As with a mutex, you can declare and initialize a condition variable statically, like this:

```
static pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

A condition variable on the stack (automatic) or allocated dynamically needs to be initialized with a call to `pthread_cond_init`, which I won't go into in this book.

Here's an example program that has the initial thread putting nodes onto a queue while another thread takes them off and displays their contents. The initial thread signals a condition when a node is queued, and the other thread waits on that condition:

```
static pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;
static pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

struct node {
    int n_number;
    struct node *n_next;
} *head = NULL;
```

```
static void *thread_func(void *arg)
{
    struct node *p;

    while (true) {
        ec_rv( pthread_mutex_lock(&mtx) )
        while (head == NULL)
            ec_rv( pthread_cond_wait(&cond, &mtx) )
        p = head;
        head = head->n_next;
        printf("Got %d from front of queue\n", p->n_number);
        free(p);
        ec_rv( pthread_mutex_unlock(&mtx) )
    }
    return (void *)true;

EC_CLEANUP_BGN
    (void)pthread_mutex_unlock(&mtx);
    EC_FLUSH("thread_func")
    return (void *)false;
EC_CLEANUP_END
}

int main(void)
{
    pthread_t tid;
    int i;
    struct node *p;

    ec_rv( pthread_create(&tid, NULL, thread_func, NULL) )
    for (i = 0; i < 10; i++) {
        ec_null( p = malloc(sizeof(struct node)) )
        p->n_number = i;
        ec_rv( pthread_mutex_lock(&mtx) )
        p->n_next = head;
        head = p;
        ec_rv( pthread_cond_signal(&cond) )
        ec_rv( pthread_mutex_unlock(&mtx) )
        sleep(1);
    }
    ec_rv( pthread_join(tid, NULL) )
    printf("All done -- exiting\n");
    return EXIT_SUCCESS;

EC_CLEANUP_BGN
    return EXIT_FAILURE;
EC_CLEANUP_END
}
```

The guts of the initial thread that queues a new node follows the pattern shown earlier—it calls `pthread_cond_signal` with the mutex locked:[22]

```
ec_rv( pthread_mutex_lock(&mtx) )
p->n_next = head;
head = p;
ec_rv( pthread_cond_signal(&cond) )
ec_rv( pthread_mutex_unlock(&mtx) )
```

And the thread that removes a node also follows the pattern, calling `pthread_cond_wait` with the mutex locked:

```
ec_rv( pthread_mutex_lock(&mtx) )
while (head == NULL)
    ec_rv( pthread_cond_wait(&cond, &mtx) )
p = head;
head = head->n_next;
printf("Got %d from front of queue\n", p->n_number);
free(p);
ec_rv( pthread_mutex_unlock(&mtx) )
```

These two threads work successfully only because `pthread_cond_wait` unlocks the mutex while it's waiting and then relocks it before returning. So, the following is guaranteed to be true:

- Both critical sections that deal with the queue are executed under the protection of the mutex `mtx`.
- When the statement

  ```
  p = head;
  ```

  is executed in the second thread, `head` is non-`NULL`.

Here's the output we got:

```
Got 0 from front of queue
Got 1 from front of queue
Got 2 from front of queue
Got 3 from front of queue
Got 4 from front of queue
Got 5 from front of queue
Got 6 from front of queue
Got 7 from front of queue
Got 8 from front of queue
Got 9 from front of queue
```

What happened to the "All done – exiting" message? We never got it. Actually, the program hung after printing the "Got 9" line, and we typed Ctrl-C to stop it.

22. It's also OK to call `pthread_cond_signal` with the mutex unlocked, and in fact that might increase throughput a bit.

Looking at the program, we can see why: The second thread stays in its loop forever, looking for a node on the queue that will never arrive, and the initial thread never returns from its call to `pthread_join`.

An obvious fix is to put some sort of "end-of-file" node on the queue to tell the second thread to exit. Or, the initial thread can just cancel the second thread when there's going to be no more work. We'll do it that way so we have an excuse to talk about how to cancel a thread.

5.17.5 Canceling a Thread

One thread can cancel another thread with `pthread_cancel`:

pthread_cancel—cancel thread

```
#include <pthread.h>

int pthread_cancel(
    pthread_t thread_id       /* ID of thread to cancel */
);
/* Returns 0 on success, error number on error */
```

Normally, the thread to be cancelled doesn't stop right away, but only at a *cancellation point,* which is when the thread calls one of the 200 or so system calls or standard functions that can block, such as `read`, `waitpid`, or `pthread_cond_wait`.[23] If a thread calls a function defined elsewhere in the program, or in a library, there's a pretty good chance of it calling one of those 200+ system calls or functions; therefore, you should consider any function call as a potential, but not guaranteed, cancellation point unless it is specifically documented otherwise and you trust the documentation.

The significance of cancellation points is that you can safely execute ordinary code without worrying about it being cancelled. For example, you can modify a linked list (possibly protected by a mutex), knowing that the code sequence will be allowed to complete.

You'll be relieved to know that none of the "pthread" mutex calls are cancellation points, nor are `free`, `calloc`, `malloc`, or `realloc`. If the mutex calls were cancellation points, it would be very cumbersome to use mutexes at all, as you would have to add code to handle cancellation every time you called `pthread_mutex_lock`.

23. The SUS specifies 65 or so that are *always* cancellation points and another 150 or so that *may* be cancellation points.

If a thread has no cancellation points at all, or none that you can depend on, yet stays alive long enough so that allowing cancellation is an issue, you can put in one or more calls to `pthread_testcancel` at safe places to explicitly provide cancellation points. This call doesn't do anything if there's no pending cancellation.

pthread_testcancel—test for cancellation

```
#include <pthread.h>

void pthread_testcancel(void);
```

I said that *normally* a thread is cancelled only at a cancellation point. This is true when its cancellation type is `PTHREAD_CANCEL_DEFERRED`, which is the default. If you set the type to `PTHREAD_CANCEL_ASYNCHRONOUS` with the `pthread_setcanceltype` function (see [SUS2002] for details), it really will be cancelled immediately, which is a scary thought. Presumably, anyone who would do that knows what he or she is doing.

Now we can modify the `main` function from the previous example to cancel the other thread when no more nodes will be queued:

```
for (i = 0; i < 10; i++) {
    ec_null( p = malloc(sizeof(struct node)) )
    p->n_number = i;
    ec_rv( pthread_mutex_lock(&mtx) )
    p->n_next = head;
    head = p;
    ec_rv( pthread_cond_signal(&cond) )
    ec_rv( pthread_mutex_unlock(&mtx) )
    sleep(1);
}
ec_rv( pthread_cancel(tid) )
ec_rv( pthread_join(tid, NULL) )
printf("All done -- exiting\n");
return EXIT_SUCCESS;
```

With this improvement the program terminates cleanly:

```
Got 0 from front of queue
Got 1 from front of queue
Got 2 from front of queue
Got 3 from front of queue
Got 4 from front of queue
Got 5 from front of queue
```

```
Got 6 from front of queue
Got 7 from front of queue
Got 8 from front of queue
Got 9 from front of queue
All done -- exiting
```

We're not quite done, though. We need to look carefully at the code for the thread that got cancelled to make sure that cancellation won't leave the queue in an inconsistent state. Here it is again:

```
static void *thread_func(void *arg)
{
    struct node *p;

    while (true) {
        ec_rv( pthread_mutex_lock(&mtx) )
        while (head == NULL)
            ec_rv( pthread_cond_wait(&cond, &mtx) )
        p = head;
        head = head->n_next;
        printf("Got %d from front of queue\n", p->n_number);
        free(p);
        ec_rv( pthread_mutex_unlock(&mtx) )
    }
    return (void *)true;

EC_CLEANUP_BGN
    (void)pthread_mutex_unlock(&mtx);
    EC_FLUSH("thread_func")
    return (void *)false;
EC_CLEANUP_END
}
```

Indeed, there are two problems:

- `pthread_cond_wait` is a cancellation point. If the thread is cancelled there, the queue is OK, as nothing yet has been done to it, but the mutex is relocked, which always is the case when leaving `pthread_cond_wait`. We don't want the thread to terminate with the mutex locked, as it can then never be unlocked.
- `printf` may be a cancellation point. If the thread terminates there, the node just removed won't be freed, resulting in a memory leak.

True, this program exits after cancelling the thread, but that may not always be the case so it makes sense to fix the bugs. It's important to protect against what *could* happen, not only against what usually happens.

A solution for the second problem is to store the number to be printed in a local variable, free the node, and then call `printf`.

But the first problem has no obvious solution. We have to install a *cancellation cleanup handler,* which is a function that's called just before cancellation to clean thing up. We'll fix the second problem in the cancellation cleanup handler as well.

A cancellation cleanup handler always has this prototype (the name doesn't matter):

```
void cleanup_handler(void *arg);
```

A thread installs a cleanup handler with `pthread_cleanup_push` and uninstalls it with `pthread_cleanup_pop`:

pthread_cleanup_push—install cleanup handler

```
#include <pthread.h>

void pthread_cleanup_push(
    void (*handler)(void*),    /* pointer to cleanup-handler function */
    void *arg                  /* data to pass to function */
);
```

pthread_cleanup_pop—uninstall cleanup handler

```
#include <pthread.h>

void pthread_cleanup_pop(
    int execute                /* execute handler? */
);
```

When cancellation occurs, the cleanup handler is called. If there's more than one, they're all called, in the reverse order of how they were pushed. Each function is popped after it's called, which is fine, since the thread will be gone.

These functions must be paired, and they must be at the same C or C++ block level. If you don't do this, you'll probably get very strange *compile-time* errors, as the functions are usually implemented as macros with embedded braces (like `EC_CLEANUP_BGN` and `EC_CLEANUP_END`; see Section 1.4.2).

Depending on how you've done things, it may make sense to call the cleanup handler even when the thread exits normally. To make that convenient, you can make the `execute` argument to `pthread_cleanup_pop` true.

Here's the improved version of `thread_func`. Note that I explicitly initialized `p` to `NULL` to make sure its value is always valid for `free`.

```
static void cleanup_handler(void *arg)
{
    free(arg);
    (void)pthread_mutex_unlock(&mtx);
}

static void *thread_func(void *arg)
{
    struct node *p = NULL;

    pthread_cleanup_push(cleanup_handler, p);
    while (true) {
        ec_rv( pthread_mutex_lock(&mtx) )
        while (head == NULL)
            ec_rv( pthread_cond_wait(&cond, &mtx) )
        p = head;
        head = head->n_next;
        printf("Got %d from front of queue\n", p->n_number);
        free(p);
        ec_rv( pthread_mutex_unlock(&mtx) )
    }
    pthread_cleanup_pop(false);
    return (void *)true;

EC_CLEANUP_BGN
    (void)pthread_mutex_unlock(&mtx);
    EC_FLUSH("thread_func")
    return (void *)false;
EC_CLEANUP_END
}
```

Now we should be OK. Note that the whole complication of dealing with cancellation could have been avoided if I chose some other way to terminate the thread once the initial thread had stopped putting nodes on the queue. That's something to consider in your own applications—don't use cancellation when a less ruthless action will do the job. On the other hand, if the thread to be terminated is blocked (in `read`, say) and you can't very easily get it unblocked, thread cancellation may be the best choice. It's certainly a better choice than a signal, which also can unblock a system call; as we shall see in Chapter 9, signals have much worse side effects.

5.17.6 Threads vs. Processes

You're probably wondering when you should use threads vs. processes. Generally, you use threads when you want concurrent processing on the same complex data structures. Three common situations are:

- You want the user interface to be active while other computations are going on in the background. An example would be background printing or background page formatting in a word-processing program that allows the document to be viewed and edited concurrently with those background operations.
- You want to organize the algorithm to take advantage of a multiprocessing computer—one with multiple CPUs. On such a system, the UNIX scheduler can automatically assign different threads to different CPUs.
- You need to deal with several kinds of events that cause systems calls (e.g., `read`, `waitpid`, `msgrcv`) to block. This is the subject of the next section.

Processes would be used for less tightly coupled applications, where some data is passed between the processes, but there's no need for the same data structure to be manipulated directly. (With some trouble, it is possible to share a data structure using shared memory, as we'll see in Chapter 7.) Separate processes have their own effective user ID, file descriptors, global variables, etc., whereas separate threads don't. Processes can more easily be developed, tested, and debugged separately, and there's much less need for locking, so much less likelihood of undetected race conditions or deadlock. Another way to draw the comparison is to say that processes are for the big pieces of the applications, whereas threads provide more fine-grained concurrency.

Warning: As of this writing, the thread packages commonly shipped with Linux and FreeBSD don't adhere to the POSIX standard, chiefly because they implement a thread either too weakly (doing everything in user space) or too strongly (using a process for each thread). The main problem with the former is that a blocking system call, like `msgrcv`, blocks all threads, not just the one containing the call. The problem with the latter is that some systems calls like `waitpid` don't work right because they're in the wrong process (only the parent may wait for a child process). Another irritant is that you may have to find, compile, and install the thread package you want yourself if the package that came with the system is inadequate.

Look for the newest Linux thread implementation, called Native POSIX Thread Library for Linux (NPTL), which is now finding its way into Linux systems. It fixes all of the important POSIX-compliance bugs.

5.18 The Blocking Problem

We've encountered several system calls that can block, like `read`, `write`, `pthread_cond_wait`, and `waitpid`, and there are lots more in this book, especially in Chapters 7 and 8. One of the most difficult problems with UNIX programming is that your application may have to block in more than one system call because you don't know what's going to happen next. I call this the blocking problem.

For blocking system calls that take a file descriptor, like `read` and `write`, you can use a single system call, `select` or `poll` (Section 4.2), to block until one or more file descriptors are ready. But that doesn't help in general because lots of things you wait for, such as processes, signals, messages, semaphores, and condition variables, aren't associated with file descriptors. Yes, you can notify a thread blocked in `select` or `poll` when, say, a message arrives by writing to a pipe set up just for that purpose (as we'll see in the next section), but that doesn't help with the fact that `msgrcv` or `mq_receive` was itself blocked waiting for the message in the first place. There's no way to tie a message-waiting system call directly to a file descriptor. It's not a *notification* problem, it's a *blocking* problem.

5.18.1 Solutions Using Processes and Threads

Historically, the solution to the blocking problem was to create a child process to block.[24] When whatever it's blocking on unblocks, it then writes to a pipe to indicate to its parent process that an event has occurred. Pipes are a good choice because they're easy to set up (as we'll see in the next chapter) and, as they use file descriptors, the parent can incorporate them into a `select` or `poll`. What this does, in effect, is to transform the parent from blocking on the non-file-descriptor event to blocking on a file descriptor. Transform everything, use `select` or `poll` in the parent, and you're golden.

But processes are very heavyweight objects in UNIX—expensive to create, expensive to schedule, and in limited supply. Also, it's cumbersome for two processes to share the same data structures. It would be nice if the event could just be queued up when it occurs and then examined whenever it's convenient for the

24. This technique was invented by my mother around 1953. At the grocery store, she would have one child wait on the deli line and one on the fish line while she shopped for goods that didn't require a line. Occasionally even she got blocked, though, so she produced three more children.

parent. Setting up an event queue between processes is possible using shared memory and semaphores, but interprocess semaphores may be too slow, especially if the events are occurring rapidly.

Another problem with separate processes is that certain objects, like file descriptors, can't be passed to an existing process (not portably, anyway). They can be passed only through inheritance; therefore, if process A is in charge of blocking on I/O and is already running, process B that just opened a network connection can't get process A to wait on the newly acquired file descriptor.

A newer solution for the blocking problem is to use POSIX Threads. One simple approach is to create a new thread for each object on which you want to block (e.g., message queue, file-descriptor set, semaphore). Each thread simply issues an appropriate blocking system call (`O_NONBLOCK` clear). When the system call returns, the thread adds an event to a shared queue (using a mutex for protection) and then goes right back into the blocking system call. The main thread then has only one thing to block on: the presence of an event on the queue. The blocking threads use a condition variable to signal the main thread when the queue becomes nonempty, exactly as in the example in Section 5.17.4.

5.18.2 Unified Event Manager Prototype

I'll show a generalized approach to using threads to solve the blocking problem that I call a Unified Event Manager. It's a collection of library functions that any application can use. An application can register an event, which causes the library to create a thread to block. The application is organized around waiting on a single event queue. When an event shows up, the application takes it off the queue, processes it, and goes back to waiting.

I won't show all the code in this book, but it's on the Web site if you want to see it. It's a *prototype* because it's not efficient enough for critical applications and because it uses the same "ec" error-checking approach that all the examples in this book use, and that is itself a prototype.

We start with an enumeration for almost all events that can be waited for in UNIX:

```
enum UEM_TYPE {
    UEM_SVMSG,      /* System V message */
    UEM_PXMSG,      /* POSIX message */
    UEM_SVSEM,      /* System V semaphore */
    UEM_PXSEM,      /* POSIX semaphore */
    UEM_FD_READ,    /* file-descriptor set - read */
```

```
    UEM_FD_WRITE,   /* file-descriptor set - write */
    UEM_FD_ERROR,   /* file-descriptor set - error */
    UEM_SIG,        /* signal */
    UEM_PROCESS,    /* process */
    UEM_HEARTBEAT,  /* heartbeat */
    UEM_NONE        /* none */
};
```

The last two require some explanation: `UEM_HEARTBEAT` is a way to just get a periodic event at certain intervals, and `UEM_NONE` is there because it's always a good idea to have a value that you can use to indicate that something's empty.

When we register an event, we need a structure to keep track of what we'll need for the blocking system call. For example, to issue a `select`, we'll need the file-descriptor set, so this structure holds all of this kind of data for each `UEM_TYPE`:

```
struct uem_reg {
    enum UEM_TYPE ur_type;          /* type of registration */
    pthread_t ur_tid;               /* thread ID */
    union {
        int ur_mqid;                /* System V message-queue ID */
        struct {
            int s_semid;            /* System V semaphore-set ID */
            struct sembuf *s_sops;  /* semaphore operations */
        } ur_svsem;
#ifdef POSIX_IPC
        mqd_t ur_mqd;               /* POSIX message-queue descriptor */
        sem_t *ur_sem;              /* POSIX semaphore */
#endif
        int ur_signum;              /* signal number */
        pid_t ur_pid;               /* process ID */
        long ur_usecs;              /* microseconds (for heartbeat) */
        fd_set ur_fdset;            /* file-descriptor set */
    } ur_resource;
    void *ur_data;                  /* data to be queued with event */
    size_t ur_size;                 /* size (used for various purposes) */
};
```

The macro `POSIX_IPC` isn't a standard macro but one that we'll use in this prototype. It's set from the real feature-test macros in a very complicated way that's explained in Section 1.5.4. We need it because Linux and FreeBSD don't as yet support POSIX IPC.

When an application wants to register an event type, it calls a function of the form `uem_register_`*E,* where *E* is an abbreviation for the event type. For example, here's the call to register to wait for a process to terminate:

```
ec_false( uem_register_process(pid, NULL) )
```

This is instead of issuing a `waitpid` directly, which would block. When process `pid` terminates, an event containing its exit status will be put on the queue, and the application that registered the event can then get the status. We'll see those details shortly.

Here's the code for `uem_register_process`:

```
bool uem_register_process(pid_t pid, void *data)
{
    struct uem_reg *p;

    ec_null( p = new_reg() )
    p->ur_type = UEM_PROCESS;
    p->ur_resource.ur_pid = pid;
    p->ur_size = 0;
    p->ur_data = data;
    ec_rv( pthread_create(&p->ur_tid, NULL, thread_process, p) )
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

The function `new_reg`, used by all the `uem_register_` *E* calls, just allocates a structure. We put it in a separate function in case there's some common initialization to do, which there isn't with the current design.

```
static struct uem_reg *new_reg(void)
{
    struct uem_reg *p;

    ec_null( p = calloc(1, sizeof(struct uem_reg)) )
    return p;

EC_CLEANUP_BGN
    return NULL;
EC_CLEANUP_END
}
```

The registration info is just passed onto the thread function, which looks like this:

```
static void *thread_process(void *arg)
{
    struct uem_event *e = NULL;

    pthread_cleanup_push(cleanup_handler, e);
    ec_null( e = calloc(1, sizeof(struct uem_event)) )
```

```
    e->ue_reg = (struct uem_reg *)arg;
    if (waitpid(e->ue_reg->ur_resource.ur_pid, &e->ue_result, 0) == -1)
        e->ue_errno = errno;
    ec_false( queue_event(e) )
    pthread_cleanup_pop(false);
    return NULL;

EC_CLEANUP_BGN
    uem_free(e);
    EC_FLUSH("thread_process")
    return NULL;
EC_CLEANUP_END
}
```

This function first pushes a cleanup handler (explained in Section 5.17.5). Then it allocates an event structure to be queued when the event occurs (`waitpid` returning, in this case). Here's the event structure used by all the threads:

```
struct uem_event {
    struct uem_reg *ue_reg;
    void *ue_buf;
    ssize_t ue_result;
    int ue_errno;
    struct uem_event *ue_next;
};
```

Note that it points back to the registration, which contains data common to all the events of this type. The `ue_buf` member is in case data has to be returned (such as a message), but there's none in this case. We do have the status, which we put in the `ue_result` member. If an error occurred, `errno` goes into the `ue_errno` member; the application that gets this event needs to check that member against zero to see if an error was returned by the function that waited. The `ue_next` member is so that the `uem_event` structures can be linked into an event queue.

The cleanup handler is just like the one in Section 5.17.5 except it calls `uem_free` (not shown here) to free the event structure if one was allocated:

```
static void cleanup_handler(void *arg)
{
    (void)uem_free((struct uem_event *)arg);
}
```

An application that removes an event from the queue is also responsible for calling `uem_free`.

The actual work of putting an event on the queue is done by `queue_event`, which the thread calls when `waitpid` returns. Note that the event is queued even if `waitpid` reported an error, which is how the application finds out about such errors.

```
static pthread_mutex_t uem_mtx = PTHREAD_MUTEX_INITIALIZER;
static pthread_cond_t uem_cond_event = PTHREAD_COND_INITIALIZER;
static struct uem_event *event_head;

static bool queue_event(struct uem_event *e)
{
    struct uem_event *cur;

    ec_rv( pthread_mutex_lock(&uem_mtx) )
    if (event_head == NULL)
        event_head = e;
    else {
        for (cur = event_head; cur->ue_next != NULL; cur = cur->ue_next)
            /* queue same error only once */
            if (e->ue_errno != 0 &&
              cur->ue_reg->ur_type == e->ue_reg->ur_type &&
              cur->ue_errno == e->ue_errno) {
                ec_rv( pthread_mutex_unlock(&uem_mtx) )
                uem_free(e);
                return true;
            }
        cur->ue_next = e;
    }
    ec_rv( pthread_cond_signal(&uem_cond_event) )
    ec_rv( pthread_mutex_unlock(&uem_mtx) )
    return true;

EC_CLEANUP_BGN
    (void)pthread_mutex_unlock(&uem_mtx);
    return false;
EC_CLEANUP_END
}
```

This function follows the pattern from Section 5.17.4 for using a condition variable for signaling an event. However, an error event (`ue_errno` nonzero) could be generated repeatedly if the thread that calls `queue_event` keeps getting an error. For example, if the process ID passed to `waitpid` is invalid, `waitpid` will keep returning until the thread is cancelled. It's bad enough that that wastes CPU time, but we certainly don't want to fill the event queue with thousands of events all reporting the same thing. So, before queuing an error event, we make sure one just like it isn't already there. (This isn't the best approach, but this is a prototype, right?)

If the event is to be queued, the line

```
cur->ue_next = e;
```

queues it and then we signal the condition, unlock the mutex, and return.

That's basically the whole library except that there are a collection of registration functions like `uem_register_process`, which I won't show here because they all do pretty much the same thing, varying only in the system call that blocks. That is, `uem_register_svmsg` starts a thread that contains a call to `msgrcv`, `uem_register_pxmsg` starts a thread that contains a call to `mq_receive`, and so on. They all use `queue_event`.

The application that uses the library looks something like this:

```
struct uem_event *e;
...
ec_false( uem_register_process(pid, NULL) )
ec_false( uem_register_pxmsg(mqd, NULL) )
...
while (true) {
    ec_null( e = uem_wait() )
    if (e->ue_errno != 0)
        ... /* display error */
    else
        switch (e->ue_reg->ur_type) {
        case UEM_PXMSG:
            ... /* process received message */
            break;
        case UEM_PROCESS:
            ... /* process status from terminated process */
            break;
        ...
        }
}
```

The most important thing here is that the application blocks in exactly one place, the call to `uem_wait`, no matter how many different kinds of events need to be handled. Here's the code for `uem_wait`; note that, like `queue_event`, it follows the conditional-variable pattern in Section 5.17.4:

```
struct uem_event *uem_wait(void)
{
    struct uem_event *e = NULL;
```

```
    ec_rv( pthread_mutex_lock(&uem_mtx) )
    while (event_head == NULL)
        ec_rv( pthread_cond_wait(&uem_cond_event, &uem_mtx) )
    e = event_head;
    event_head = event_head->ue_next;
    ec_rv( pthread_mutex_unlock(&uem_mtx) )
    return e;

EC_CLEANUP_BGN
    (void)pthread_mutex_unlock(&uem_mtx);
    return NULL;
EC_CLEANUP_END
}
```

When the predicate is true (event present on queue), it removes the event from the queue and returns a pointer to it. It's the responsibility of the caller to free the memory with a call to `uem_free`, as I mentioned earlier.

Except for some additional bookkeeping details and the rest of the registration functions, that's the whole system. It solves the blocking problem!

Because it's so reliant on threads, this approach is only as good as the underlying thread implementation. Threads have to be fast, lightweight, and plentiful, and they have to follow the POSIX standard rigorously. Otherwise, there will be too much overhead, valuable system resources will be consumed, and there will be too many subtle bugs in the application that, because of all the multithreading, will be very hard to find.

If you're interested, try to rewrite the uem package using processes instead of threads. You'll find it difficult to write and extremely difficult to make efficient. But the effort should help you appreciate why threads are so important.

Exercises

5.1. Correct the memory-leak problems in `setenv` and `unsetenv` as suggested at the end of Section 5.2.

5.2. Rewrite the environment-manipulation functions in Section 5.2 to treat exported variables as the standard shell does. That is, an updated value for a variable is exported only if it's specifically declared to be exported in a function named, say, `env_export`. Give some thought to whether and when `environ` will be updated

and whether the existing `getenv` function can be used as is or whether it needs to be replaced.

5.3. Write a program that scans its arguments for assignments of the form *variable=value,* updates the environment appropriately, and then executes the program specified by the first nonassignment argument. The other nonassignment arguments become arguments to the invoked program. Don't use `fork`.

5.4. Write a function `execlp2` that's analogous to `execvp2` in Section 5.3. Use the Standard C variable-argument facilities (`va_arg`, etc.).

5.5. Enhance the `exec_path` function in Section 5.3 to support the `#!` feature discussed in that section.

5.6. Design and implement two functions, `execvx` and `execlx`, to replace the six `exec` system calls, as suggested by the footnote in Section 5.3. You can use as many of the `exec` system calls as you wish in your implementation.

5.7. Explain in your own words (perhaps with a diagram) how `fork` and `exec` are typically implemented. (You'll need to do some research; see, for example, [Bac1986], [McK1996], [Mau2001], or [Bov2001].) Then show how `posix_spawn` can be implemented more efficiently. Don't write the code—pseudo code or a clear step-by-step algorithm will do.

5.8. Research the semantics of `posix_spawn` (see [SUS2002]) and then implement it in terms of `fork` and `exec`, as suggested at the end of Section 5.5. To start with, skip both attributes and actions. Then do actions, and finally attributes. To do the whole job you'll need to implement the associated system calls (e.g., `posix_spawn_file_actions_init`). This is an extremely useful exercise, even if you don't care about realtime systems.

5.9. Design and run an experiment to measure the CPU time used by `fork`. (You may want to use `timestart` and `timestop` from Section 1.7.2.) If you have access to them, try different versions of UNIX and different hardware. Compare the time for `fork` with the time for `vfork`, if your systems support it. Ditto for `posix_spawn`.

5.10. Investigate the equivalent system calls to `exec` and `fork` that are provided in other operating systems, such as VMS, OS/390, Windows, and MacOS. Compare their features and summarize their advantages and disadvantages.

5.11. In Section 5.17.3, it's possible for `x` to be incremented by another thread between reading with `get_x(0)` and incrementing with `get_x(1)`, which might result in `x` being too big. Fix the problem by rewriting the example to use a single function call that both tests and increments `x`.

5.12. Write `fileview`, an interactive full-screen application (Section 4.8) that divides the screen with a horizontal line into two parts. An "s" command prompts for a search string, searches the standard include files (at least) for files that contain it on one or more lines, and displays the matching pathnames in the upper part. While they're being displayed or afterwards, a "v" command shows the contents of a selected file in the lower part with the matched strings highlighted. Choose two keys that can be used to scroll the upper part up and down, and two other keys for the lower part. To make the "v" command easy to implement, number the pathnames in the upper part and arrange for the "v" command to prompt for the file by its number. Note the word "While" in the phrase "While they're being displayed"; it implies that you'll have to use multithreading and, as Curses isn't necessarily thread-safe, you'll have to protect access to it with a mutex. There's also a "q" command that quits the application.

5.13. Explain how you could implement `fileview` (Exercise 5.12) without multithreading, using processes instead. You may have to explore Chapter 7 first. If you don't think you can do it without multithreading, explain why not. (As you would be trying to prove a negative statement, your explanation will have to be very persuasive.)

5.14. Write a program to display as many as the process attributes listed in Appendix A as you can, limiting yourself for now to those discussed in the first five chapters of this book. Later, you can extend your program to include them all. If there are any you can't display, explain why not. To make the output interesting, execute some system calls at the start to open some files, set some signal actions, and so on.

This page intentionally left blank

6

Basic Interprocess Communication

6.1 Introduction

Now that we know how to create processes, we want to connect them so they can communicate. We'll do this with pipes in this chapter, using basic techniques that have always worked on all versions of UNIX. In the next chapter we'll start exploring interprocess communication using techniques that are more efficient and robust, less universally available, and more troublesome to program.

Pipes are familiar to most UNIX users as a shell facility. For instance, to display a sorted list of who's logged in, you can type:

```
$ who | sort | more
```

There are three processes here, connected with two pipes. Data flows in one direction only, from `who` to `sort` to `more`. It's also possible to set up pipelines for two-way communication (from process A to B and from B back to A) and pipelines in a ring (from A to B to C to A) using system calls. Most shells, however, provide no notation for these more elaborate arrangements, so they are unknown to most UNIX users.[1]

I'll begin by showing some simple examples of processes connected for one-directional communication. Then we'll improve on the primitive shell we developed in Chapter 5. Our new shell will be complete enough to be called "real"—it will handle pipelines, background processes, I/O redirection, and quoted argu-

1. These arrangements can be set up with FIFOs, but the shell is an innocent bystander—it thinks they are regular files.

ments. It will lack file-name generation (e.g., `ls t*.?`) and programming constructs (e.g., `if` statements). At the end I'll show how to connect processes for two-way communication and expose the deadlock problems that can arise.

6.2 Pipes

This section is about unnamed pipes, although much of the behavior described applies to FIFOs (named pipes) as well. FIFOs are covered in detail in Section 7.2.

6.2.1 `pipe` System Call

pipe—create pipe

```
#include <unistd.h>

int pipe(
    int pfd[2]              /* file descriptors */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

The `pipe` system call creates a pipe, which is a communication channel represented by two file descriptors that are returned in the `pfd` array. Writing to `pfd[1]` puts data in the pipe; reading from `pfd[0]` gets it out.

There's a parameter called `PIPE_BUF` in UNIX documentation that you can think of as the buffer size of the pipe. If several processes or threads are writing to the same pipe, a write of `PIPE_BUF` or fewer bytes is guaranteed to be atomic—there will be no interleaving of data from other writers. This property is extremely important if several processes are writing structured data, since without it there would be no guarantee that a reader could read valid data. `PIPE_BUF` is always at least 512, but you can get the actual value for a particular pipe at run-time with a call to `fpathconf` (Section 1.5.6):

```
int pfd[2];
long v;

ec_neg1( pipe(pfd) )
errno = 0;
v = fpathconf(pfd[0], _PC_PIPE_BUF);
```

```
if (v == -1)
    if (errno != 0)
         EC_FAIL
    else
         printf("No limit for PIPE_BUF\n");
else
    printf("PIPE_BUF = %ld\n", v);
```

On my systems, I got 5120 on Solaris, 512 on FreeBSD, and 4096 on Linux.

Initially the `O_NONBLOCK` (Section 4.2.2) is clear for a pipe; that is, reading and writing may block. As you would guess, you can set the flag with `fcntl` (Section 3.8.3). How this flag affects `read`s and `write`s is explained below.

6.2.2 Pipe (and FIFO) I/O Behavior

Everything in this section applies to both pipes and to FIFOs (discussed further in Section 7.2), and here the term "pipe" means both.

Some I/O system calls act differently on pipe file descriptors from the way they do on ordinary files, and some do nothing at all, as summarized by the following list (these are the main ones; [SUS2002] has all the details):

`write` Data written to a pipe is sequenced in order of arrival. Normally (`O_NONBLOCK` clear), if the pipe becomes full, `write` will block until enough old data is removed by `read`; there are no partial writes. The capacity of a pipe varies with the UNIX implementation, but obviously it is always at least `PIPE_BUF` bytes. If `O_NONBLOCK` is set and the amount to be written is `PIPE_BUF` or less, `write` will either write the data immediately or return –1 with `errno` set to `EAGAIN`; there are no partial writes. But if the amount is greater than `PIPE_BUF`, a partial write is possible.

`read` Data is read from a pipe in order of arrival, just as it was written. Once read, data can't be reread or put back. Normally (`O_NONBLOCK` clear), if the pipe is empty, `read` will block until at least one byte of data is available, unless all writing file descriptors are closed, in which case the `read` will return with a 0 count (the usual end-of-file indication). But the byte count given as the third argument to read will not necessarily be satisfied—only as many bytes as are present at that instant will be read, and an appropriate count will be returned. The byte count will never be exceeded, of course; unread bytes will remain for the next read. If `O_NONBLOCK` is set, a read on an empty pipe will return –1 with `errno` set to `EAGAIN`.

`close` Means more on a pipe than it does on a file. Not only does it free up the file descriptor for reuse, but when all writing file descriptors are closed it acts as an end-of-file for a reader. If all reading file descriptors are closed, a `write` on a writing file descriptor will cause an error. A fatal signal is also normally generated; see Section 9.1.3.

`fstat` Not very useful on pipes, except to determine that the file descriptor is open to a pipe. The size returned is usually the number of bytes in the pipe, but this is not required by any UNIX standard.

`dup` This system call and `dup2` are explained in Section 6.3.

`lseek` Not used with pipes. This means that if a pipe contains a sequence of messages, it isn't possible to look through them for the message to be read next. Like toothpaste in a tube, you have to get it out to examine it, and then there's no way to put it back. This is one reason why pipes are awkward for application programs that pass messages between processes.

System calls not explicitly listed (e.g., `select`, `poll`) operate on pipes just as you would expect. For example, when a file descriptor tested by `select` is open to a pipe, `select` tests whether a `read` or `write` would block.

For writes, the relationships between atomic vs. nonatomic, blocking vs. nonblocking, and complete vs. partial vs. deferred (–1 return with an `errno` of `EAGAIN`) is a little complicated. Table 6.1 (based on tables in POSIX1990) should help. Columns one and two contain the possibilities for the state of the `O_NONBLOCK` flag and the amount being written, and the last three columns indicate what happens for a full pipe, a pipe that can receive part of the data immediately, and a pipe that can receive all the data.

In the table, the notation "complete write" means that `write` doesn't return until the full requested amount is written. "Nonatomic" means that the data is all in the pipe, but not necessarily contiguously (not even the first `PIPE_BUF` bytes are guaranteed to be contiguous in this case). "`EAGAIN`" means a return from write of –1 with `errno` set to `EAGAIN`. "Partial" means that the value returned by `write` is less than the requested amount.

The reason for the notation "might block" at the end of the second row is that while it's true that when the `write` starts all the data will fit, since it is nonatomic, another process or thread could fill some of the pipe before the `write` completes, forcing it to block.

Table 6.1 Writing to a Pipe

O_NONBLOCK?	Amount to write	None immediately writable	Some immediately writable (>=1 and < amt.)	All immediately writable
clear	<=PIPE_BUF	blocks; complete write; atomic	blocks; complete write; atomic	does not block; complete write; atomic
clear	>PIPE_BUF	blocks; complete write; nonatomic	blocks; complete write; nonatomic	might block; complete write; nonatomic
set	<=PIPE_BUF	EAGAIN	EAGAIN	does not block; complete write; atomic
set	>PIPE_BUF	EAGAIN	does not block; partial or EAGAIN; nonatomic	does not block; complete, partial, or EAGAIN; nonatomic

Summarizing writes to a pipe one more time:

- If the requested amount is PIPE_BUF or less, writes are always atomic (which implies never partial).
- If O_NONBLOCK is clear (the normal case), writes are never partial, even when they are nonatomic.
- The only partial writes occur when O_NONBLOCK is set and the requested amount is greater than PIPE_BUF.

Table 6.2 is for reads.

Table 6.2 Reading from a Pipe

O_NONBLOCK?	None immediately readable	Some or all immediately readable (>=1 and <= amt.)
clear	blocks unless no writers (0 returned)	does not block; possibly partial read
set	EAGAIN unless no writers (0 returned)	does not block; possibly partial read

Note that the `read` table is much simpler than the `write` table, because there's never a guarantee of atomicity or of a complete read. `Read`s work like this:

- If all writing file descriptors are closed, a `read` on an empty pipe always immediately returns 0, which most programs treat as an end-of-file.
- If a writing file descriptor is open, a `read` on an empty pipe blocks or not depending on whether `O_NONBLOCK` is set.
- A `read` on a nonempty pipe always returns immediately, no matter what the relationship between the requested amount and the amount of data in the pipe. The amount actually read is returned.
- Because there is no guarantee of atomicity, you must never allow multiple readers unless you have another concurrency-control mechanism (e.g., an interprocess semaphore) to prevent simultaneous reading. (In practice, you will seldom do this—you'll use something like message queues instead.)

Here are additional guidelines to keep in mind:

- If you have one writer and one reader (e.g., a shell pipeline) and the reader is prepared for partial reads (e.g., uses Standard C I/O functions), write whatever amount you like, although multiples of the block size are most efficient, as we saw in Section 2.12.
- If you have multiple writers (and one reader), always write `PIPE_BUF` or less. There's no practical way to get pipes to work correctly with a greater amount, unless you use another synchronization mechanism, such as a semaphore (Section 7.8).
- Don't assume anything about the value of `PIPE_BUF` other than that it is at least 512. If you really need to know what it is, use `fpathconf`.
- Even though the standard doesn't require atomic reads, essentially all implementations make them atomic if the requested amount is `PIPE_BUF` or less. Use this fact at your own risk.
- Remember that to get an end-of-file (0 return from `read`) all writing file descriptors must be closed, including any in the process that's doing the reading. This point will be clearer later on, when we show how to connect two processes with a pipe.

6.2.3 Pipe Examples

(Now I'm talking once again only about *unnamed* pipes; FIFO examples are in the next chapter.)

Considering a single process only, of what use is a pipe? None, but such an example is very informative:

```
void pipetest(void)
{
    int pfd[2];
    ssize_t nread;
    char s[100];

    ec_neg1( pipe(pfd) )
    ec_neg1( write(pfd[1], "hello", 6) )
    ec_neg1( nread = read(pfd[0], s, sizeof(s)) )
    if (nread == 0)
        printf("EOF\n");
    else
        printf("read %ld bytes: %s\n", (long)nread, s);
    return;

EC_CLEANUP_BGN
    EC_FLUSH("pipetest");
EC_CLEANUP_END
}
```

The output:

```
read 6 bytes: hello
```

We could safely write 6 bytes into the pipe without worrying about it filling up, since that's well below 512 (the minimum for `PIPE_BUF`). But if we wrote much more and did fill the pipe, the `write` would block until the `read` emptied it some. This could never happen, however, because the program would never get to the `read`. We would be stuck in a situation called *deadlock.* You must be careful to avoid deadlock when using pipes, but don't be overly concerned: When you connect processes with single pipes used for one-way communication (like the shell does), deadlock (due to pipes) is *impossible.* Only the more exotic arrangements can cause deadlock.

Given that we have two processes, how can we connect them so that one can read from a pipe what the other writes? We can't. Once the processes are created they can't be connected, because there's no way for the process that creates the pipe to pass a file descriptor to the other process.[2] It can pass the file descriptor number, of course, but that number won't be valid in the other process. But if we make a pipe in one process *before* creating the other process, it will inherit the pipe file

2. Some systems have a nonportable way of doing this.

descriptors, and they'll be valid in both processes. Thus, two processes communicating over a pipe can be parent and child, or two children, or grandparent and grandchild, and so on. They must be related, however, and the pipe must be passed on at birth. In practice, this may be a severe limitation because, if a process dies, there's no way to recreate it and reconnect it to its pipes—the survivors must be killed too, and then the whole family has to be recreated.

In the following example one process (running the function `pipewrite`) makes a pipe, creates a child process (running `piperead`) that inherits it, and then writes some data to the pipe for the child to read. Although the child has inherited the necessary reading file descriptor, it still doesn't know its number, so the number is passed as an argument.

```
void pipewrite(void)
{
    int pfd[2];
    char fdstr[10];

    ec_neg1( pipe(pfd) )
    switch (fork()) {
    case -1:
        EC_FAIL
    case 0: /* child */
        ec_neg1( close(pfd[1]))
        snprintf(fdstr, sizeof(fdstr), "%d", pfd[0]);
        execlp("./piperead", "piperead", fdstr, (char *)NULL);
        EC_FAIL
    default: /* parent */
        ec_neg1( close(pfd[0]) )
        ec_neg1( write(pfd[1], "hello", 6) )
    }
    return;

EC_CLEANUP_BGN
    EC_FLUSH("pipewrite");
EC_CLEANUP_END
}
```

Here is the code for the child process:

```
int main(int argc, char *argv[])
{
    int fd;
    ssize_t nread;
    char s[100];
```

```
    fd = atoi(argv[1]);
    printf("reading file descriptor %d\n", fd);
    ec_neg1( nread = read(fd, s, sizeof(s)) )
    if (nread == 0)
        printf("EOF\n");
    else
        printf("read %ld bytes: %s\n", (long)nread, s);
    exit(EXIT_SUCCESS);

EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
EC_CLEANUP_END
}
```

And here is the output:

```
reading file descriptor 3
read 6 bytes: hello
```

Some comments about this example are needed:

- Since the child will only be reading the pipe, not writing it, it closes the writing end (`pfd[1]`) right away (the statement after `case 0` in the parent) to conserve file descriptors. (If the child were going to read more, it would be crucial to close the writing end, or the child would never get an end-of-file.)
- The parent has no use for the reading end of the pipe, so it closes `pdf[0]`.
- `snprintf` converts the reading file descriptor from an integer to a string so it can be used as a program argument.
- Since we know that the child program is in the current directory, we code its path as `./pread` to stop `execlp` from searching. This saves time, but more importantly, it prevents the accidental execution of some other `piperead` instead (no telling what might be in the user's path). We could have accomplished the same thing by using `execl`, which doesn't search.
- We didn't bother coding a `wait` for the child, since in this simple example the parent is going to exit right away.
- The child runs `piperead`, which just converts its argument back to an integer and reads from that file descriptor. (Remember, knowing that integer doesn't make the file descriptor valid—it's valid because it was inherited.)

In general, then, here is how to connect two processes with a pipe for one-way communication:

1. Create the pipe.
2. Fork to create the reading child.

3. In the child, close the writing end of the pipe and do any other preparations that are needed. (We'll see what these are in subsequent examples.)
4. In the child, execute the child program.
5. In the parent, close the reading end of the pipe.
6. If a second child is to write on the pipe, create it, make the necessary preparations, and execute its program. If the parent is to write, go ahead and write.

All our examples of one-way piping will follow this paradigm.

Now we can see why `fork` and `exec` are separate system calls. Why not a single system call to do both jobs, to save the overhead of `fork`? The two are separated to allow us to perform Step 3 above. We've already discovered some work to do between `fork` and `exec` (closing `pfd[1]`), and further on in this chapter we'll see more. We can't do this work before the `fork` because we don't want it to affect the parent (the parent must not close the writing end of the pipe). We don't want the child's program to do the work because we want programs to be oblivious to how they got invoked and how their inputs and outputs are connected (this is a cornerstone of UNIX philosophy). So we do it in the perfect place: in the child, executing code cloned from the parent. An added benefit is that the connection code is localized, making it easier to debug and modify.

To persist: Since there are only a few ways that processes are typically connected, why not have a single `fork-exec` system call with various options for interprocess connection? After all, there are other system calls with lots of options, so why not here? The answer is simply that the original designers of UNIX, Thompson and Ritchie, wanted to minimize the number of system calls. `fork` and `exec` are simple, and yet they allow the caller to arrange a wide variety of customized connections. So why kludge up another system call?[3]

The program `piperead` was specially designed to read from the file descriptor whose number was passed in. It's a strange program—no standard UNIX command works that way. Many programs do read a particular file descriptor, which they assume to be already open, but that file descriptor is fixed at 0 (the standard input, `STDIN_FILENO`). It doesn't have to be passed as an argument. Similarly, many programs are designed to write to file descriptor 1 (the standard output, `STDOUT_FILENO`). To connect commands as the shell does we somehow need to

3. But now we have it: `posix_spawn`, which we briefly mentioned in Section 5.5.

force `pipe` to return particular file descriptors in the `pfd` array: 0 for the reading end and 1 for the writing end. Alas, `pipe` offers no such feature. We might try closing 0 and 1 before calling `pipe`, to make them available, but this isn't safe because the standard says nothing about which end is which, and, furthermore, we can't usually afford to sacrifice our standard input and output just to make a pipe. So how do we perform the trick? We use `dup` or `dup2`.

6.3 dup and dup2 System Calls

dup—duplicate file descriptor

```
#include <unistd.h>

int dup(
    int fd              /* file descriptor to duplicate */
);
/* Returns new file descriptor or -1 on error (sets errno) */
```

dup2—duplicate file descriptor

```
#include <unistd.h>

int dup2(
    int fd,             /* file descriptor to duplicate */
    int fd2             /* file descriptor to use */
);
/* Returns new file descriptor or -1 on error (sets errno) */
```

`dup` duplicates an existing file descriptor, returning a new file descriptor that is open to the same file (or pipe, etc.). The two share the same file description (see Section 2.2), just as an inherited file descriptor shares the file description with the corresponding file descriptor in the parent. The call fails if the argument is bad (not open) or if no file descriptors are available.

`dup` takes the lowest-numbered available file descriptor, so if you know what's open, you can control what it returns. It's easier, though, to use `dup2`, which allows you to specify, with the `fd2` argument, what file descriptor to return. To make `fd2` available, `dup2` closes it if it has to.

`dup(fd)` is equivalent to

```
fcntl(fd, F_DUPFD, 0)
```

and `dup2(fd, fd2)` is equivalent to

```
close(fd2);
fcntl(fd, F_DUPFD, fd2);
```

except that `dup2` is atomic—if there's a problem with the duplication, `fd2` is not closed. We'll use `dup2`, rather than `dup`, in all our examples, because it's easier.

Since the file description is shared, there is only one benefit to having a second file descriptor: Its number is different and perhaps better suited to the caller's purposes. Suppose we create a pipe and then use `dup2` to make `STDIN_FILENO` (file descriptor 0) a duplicate of the reading end of the pipe (whose file descriptor might be 3, 4, 27, or whatever). If we then `exec` a program designed to read `STDIN_FILENO` (there are lots of them!), it will have been tricked into reading the pipe. A similar algorithm can be used to force `STDOUT_FILENO` to be the writing end of a pipe.

If the arguments to `dup2` are equal, it just returns that file descriptor without closing or duplicating anything. This is helpful in the case where, say, the reading end of the pipe is somehow already equal to `STDIN_FILENO`.

To illustrate `dup2`, here is an example that makes a pipe, creates a child to read it, arranges for the child's `STDIN_FILENO` to be the reading end of the pipe, and then invokes the `cat` command to read the pipe. Now that we're able to use `STDIN_FILENO`, we don't need a special program like `piperead` as we did in the example in the previous section.

```
void pipewrite2(void) /* has a bug */
{
    int pfd[2];
    pid_t pid;

    ec_neg1( pipe(pfd) )
    switch (pid = fork()) {
    case -1:
        EC_FAIL
    case 0: /* child */
        ec_neg1( dup2(pfd[0], STDIN_FILENO) )
        ec_neg1( close(pfd[0]))
        ec_neg1( close(pfd[1]))
        execlp("cat", "cat", (char *)NULL);
        EC_FAIL
    default: /* parent */
        ec_neg1( close(pfd[0]) )
        ec_neg1( write(pfd[1], "hello", 6) )
        ec_neg1( waitpid(pid, NULL, 0) )
    }
```

```
    return;

EC_CLEANUP_BGN
    EC_FLUSH("pipewrite2");
EC_CLEANUP_END
}
```

We got this unsurprising output:

```
hello
```

Note that in the child we closed both file descriptors we got from `pipe` after duplicating one of them. The reading end of the pipe is still open, but now its file descriptor is `STDIN_FILENO`. The parent closed `pfd[0]` (the reading end) because it didn't need it. The call to `waitpid` waits for `cat` to terminate; we had left it out of the `piperead` example in the previous section.

Unfortunately, after displaying "hello," the program hung, and we didn't get a shell prompt until we killed it with Ctrl-c. Do you see why? (Spoiler in next paragraph.)

The program hung because `cat`, which reads until it gets an end-of-file, didn't get one, so it didn't terminate. Recall that a 0 return from `read` occurs only when *all* writing file descriptors open to a pipe are closed, and at the time of the call to `waitpid`, `pfd[1]` was still open. Thus the parent and child were deadlocked. The solution is to close the writing end after writing the data:

```
    default: /* parent */
        ec_neg1( close(pfd[0]) )
        ec_neg1( write(pfd[1], "hello", 6) )
        ec_neg1( close(pfd[1]) )
        ec_neg1( waitpid(pid, NULL, 0) )
```

We don't have to limit ourselves to piping from parent to child. The next example implements the equivalent of the shell command line:

```
$ who | wc
```

to see how many users are logged in.

```
void who_wc(void)
{
    int pfd[2];
    pid_t pid1, pid2;

    ec_neg1( pipe(pfd) )
```

```
    switch (pid1 = fork()) {
    case -1:
        EC_FAIL
    case 0: /* first child */
        ec_neg1( dup2(pfd[1], STDOUT_FILENO) )
        ec_neg1( close(pfd[0]))
        ec_neg1( close(pfd[1]))
        execlp("who", "who", (char *)NULL);
        EC_FAIL
    }
    /* parent */
    switch (pid2 = fork()) {
    case -1:
        EC_FAIL
    case 0: /* second child */
        ec_neg1( dup2(pfd[0], STDIN_FILENO) )
        ec_neg1( close(pfd[0]))
        ec_neg1( close(pfd[1]))
        execlp("wc", "wc", "-l", (char *)NULL);
        EC_FAIL
    }
    /* still the parent */
    ec_neg1( close(pfd[0]) )
    ec_neg1( close(pfd[1]) )
    ec_neg1( waitpid(pid1, NULL, 0) )
    ec_neg1( waitpid(pid2, NULL, 0) )
    return;

EC_CLEANUP_BGN
    EC_FLUSH("who_wc");
EC_CLEANUP_END
}
```

This is the output:

```
1
```

Instead of `who` and `wc` both being children of the same parent, `wc` could be a child of `who`, which is just as easy to set up:

```
void who_wc2(void)
{
    int pfd[2];
    pid_t pid1, pid2;

    ec_neg1( pipe(pfd) )
    switch (pid1 = fork()) {
    case -1:
        EC_FAIL
```

```
    case 0: /* child */
        switch (pid2 = fork()) {
        case -1:
            EC_FAIL
        case 0: /* grandchild */
            ec_neg1( dup2(pfd[0], STDIN_FILENO) )
            ec_neg1( close(pfd[0]))
            ec_neg1( close(pfd[1]))
            execlp("wc", "wc", "-l", (char *)NULL);
            EC_FAIL
        }
        /* still the child */
        ec_neg1( dup2(pfd[1], STDOUT_FILENO) )
        ec_neg1( close(pfd[0]))
        ec_neg1( close(pfd[1]))
        execlp("who", "who", (char *)NULL);
        EC_FAIL
    }
    /* parent */
    ec_neg1( close(pfd[0]) )
    ec_neg1( close(pfd[1]) )
    ec_neg1( waitpid(pid1, NULL, 0) )
    return;

EC_CLEANUP_BGN
    EC_FLUSH("who_wc2");
EC_CLEANUP_END
}
```

Two points about this example:

- The child has to create the grandchild before closing the reading file descriptor so the grandchild will inherit it.
- The parent waits only for `who` (`pid1`) to terminate. It can't wait for `wc` because `wc` is not its child. The child running `who` can't wait for `wc` either because the `who` program hasn't been designed to do that. Putting a call to `waitpid` after the call that `exec`s `wc` won't help because on a successful `exec` all the code is overwritten. What happens (as explained in Section 5.8) is that when `who` (`wc`'s parent) terminates, a system process will inherit `wc` and will wait for it.

We could just as easily have reversed things and made `wc` the parent of `who` (see Exercise 6.11).

Of course, it's much easier to use a shell to set up a command line like this than it is to write a custom program. In the next section we'll code such a shell.

6.4 A Real Shell

Our shell is a subset of the typical UNIX shell that you probably use. It has these features:

- A *simple command* consists of a command name followed by an optional sequence of arguments separated by spaces or tabs, each of which is a single word or a string surrounded by double quotes (`"`). If quoted, an argument may include any otherwise special characters (`|`, `;`, `&`, `>`, `<`, space, tab, and newline). An included quote or backslash must be preceded with a backslash (`\"` or `\\`). A command may have up to 50 arguments, each of which may have up to 500 characters.
- A simple command's standard input may be redirected to come from a file by preceding the file name with `<`. Similarly, the standard output may be redirected with `>`, which truncates the output file. If the output redirection symbol is `>>`, the output is appended to the file. An output file is created if it doesn't exist.
- A *pipeline* consists of a sequence of one or more simple commands separated with bars (`|`). Each except the last simple command in a pipeline has its standard output connected, via a pipe, to the standard input of its right neighbor.
- A pipeline is terminated with a newline, a semicolon (`;`), or an ampersand (`&`). In the first two cases the shell waits for the rightmost simple command to terminate before continuing. In the ampersand case, the shell does not wait. It reports the process number of each simple command in the pipeline, and each simple command is run with interrupt and quit signals ignored.
- Built-in commands are assignment, `set`, and `cd`. The first two of these were described in Section 5.4. `cd` works in the familiar way.

The first step is to parse input lines into *tokens,* which are groups of characters that form syntactic units; examples are words, quoted strings, and special symbols such as `&` and `>>`. Each token is represented by a symbolic constant, as follows:

`T_WORD` An argument or file name. If quoted, the quotes are removed after the token is recognized.

`T_BAR` The symbol `|`.

`T_AMP` The symbol `&`.

`T_SEMI` The symbol `;`.

`T_GT` The symbol `>`.

`T_GTGT` The symbol `>>`.

`T_LT` The symbol `<`.

`T_NL` A newline.

`T_EOF` A special token signifying that the end-of-file has been reached. If the standard input is a terminal, the user has typed an EOT (Ctrl-d).

`T_ERROR` A special token signifying an error.

The job of a *lexical analyzer* is to read the input and assemble the characters into tokens. Each time it is called, one token is returned. If the token is `T_WORD`, a string containing the actual characters composing it is also returned. (For the other tokens the actual characters are obvious.) The lexical analyzer should bypass irrelevant characters, such as spaces separating arguments, without returning anything.[4]

Our lexical analyzer is a finite-state-machine: As characters are read they are either recognized immediately as tokens or they are accumulated (characters of a word, for example). With each character, the lexical analyzer can switch into a new state which serves to remember what it is doing and how characters are to be interpreted. For example, when accumulating a quoted string, spaces are treated differently than when they appear outside quotes. For our shell we need four states:

`NEUTRAL` The starting state for each call to the lexical analyzer. Spaces and tabs are skipped. The characters `|`, `&`, `;`, `<`, and newline are recognized immediately as tokens. The character `>` causes a switch to state `GTGT`, which will see if it is followed by another `>`, since `>` and `>>` are two different tokens. A quote causes a switch to state `INQUOTE`, which gathers a quoted string. Anything else is taken as the beginning of an unquoted word; the character is saved in a buffer and the state is switched to `INWORD`.

4. Most UNIX systems include a command called `lex` that can automatically generate a lexical analyzer from a description of the tokens it is to recognize. Because `lex` is complicated to use, and because the lexical analyzers it generates are sometimes big and slow, it's usually preferable to code lexical analyzers by hand. They aren't difficult programs to write.

GTGT — This state means that > was just read. If the next character is also >, the token T_GTGT is returned. Otherwise, T_GT will be returned. But first, we've read one character too many, so we put it back on the input with the Standard C function ungetc.

INQUOTE — This state means that a starting quote was read. We accumulate characters into a buffer until the closing quote is read. Then we return the token T_WORD and the accumulated string. Special steps must be taken to process the escape character \.

INWORD — This state means that the first character of a word was read and has been put into the buffer. We keep accumulating characters until a nonword character is read (|, say). We put the gratuitous character back on the input and return the token T_WORD.

With this explanation, the code for our lexical analyzer, gettoken, should be readily understandable:

```
typedef enum {T_WORD, T_BAR, T_AMP, T_SEMI, T_GT, T_GTGT, T_LT,
  T_NL, T_EOF, T_ERROR} TOKEN;

static TOKEN gettoken(char *word, size_t maxword)
{
    enum {NEUTRAL, GTGT, INQUOTE, INWORD} state = NEUTRAL;
    int c;
    size_t wordn = 0;

    while ((c = getchar()) != EOF) {
        switch (state) {
        case NEUTRAL:
            switch (c) {
            case ';':
                return T_SEMI;
            case '&':
                return T_AMP;
            case '|':
                return T_BAR;
            case '<':
                return T_LT;
            case '\n':
                return T_NL;
            case ' ':
            case '\t':
                continue;
            case '>':
                state = GTGT;
                continue;
```

```
            case '"':
                state = INQUOTE;
                continue;
            default:
                state = INWORD;
                ec_false( store_char(word, maxword, c, &wordn) )
                continue;
            }
        case GTGT:
            if (c == '>')
                return T_GTGT;
            ungetc(c, stdin);
            return T_GT;
        case INQUOTE:
            switch (c) {
            case '\\':
                if ((c = getchar()) == EOF)
                    c = '\\';
                ec_false( store_char(word, maxword, c, &wordn) );
                continue;
            case '"':
                ec_false( store_char(word, maxword, '\0', &wordn) )
                return T_WORD;
            default:
                ec_false( store_char(word, maxword, c, &wordn) )
                continue;
            }
        case INWORD:
            switch (c) {
            case ';':
            case '&':
            case '|':
            case '<':
            case '>':
            case '\n':
            case ' ':
            case '\t':
                ungetc(c, stdin);
                ec_false( store_char(word, maxword, '\0', &wordn) )
                return T_WORD;
            default:
                ec_false( store_char(word, maxword, c, &wordn) )
                continue;
            }
        }
    }
    ec_false( !ferror(stdin) )
    return T_EOF;
```

```
EC_CLEANUP_BGN
    return T_ERROR;
EC_CLEANUP_END
}

static bool store_char(char *word, size_t maxword, int c, size_t *np)
{
    errno = E2BIG;
    ec_false( *np < maxword )
    word[(*np)++] = c;
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

When coding a large program, it's convenient to debug it in pieces. This not only provides early feedback, but it makes finding bugs easier since there are fewer lines of code to search. Here's a test program for `gettoken`:

```
int main(void)
{
    char word[200];

    while (1)
        switch (gettoken(word, sizeof(word))) {
        case T_WORD:
            printf("T_WORD <%s>\n", word);
            break;
        case T_BAR:
            printf("T_BAR\n");
            break;
        case T_AMP:
            printf("T_AMP\n");
            break;
        case T_SEMI:
            printf("T_SEMI\n");
            break;
        case T_GT:
            printf("T_GT\n");
            break;
        case T_GTGT:
            printf("T_GTGT\n");
            break;
        case T_LT:
            printf("T_LT\n");
            break;
```

```
        case T_NL:
            printf("T_NL\n");
            break;
        case T_EOF:
            printf("T_EOF\n");
            exit(EXIT_SUCCESS);
        case T_ERROR:
            printf("T_ERROR\n");
            exit(EXIT_SUCCESS);
        }
}
```

When we ran this program and typed this line (followed by an EOT)

```
sort <inf | pr -h "Sept. Results" >>outf&
```

we got this output:

```
T_WORD <sort>
T_LT
T_WORD <inf>
T_BAR
T_WORD <pr>
T_WORD <-h>
T_WORD <Sept. Results>
T_GTGT
T_WORD <outf>
T_AMP
T_NL
T_EOF
```

This test doesn't prove that `gettoken` is correct, but it's encouraging enough for us to go on with more of the shell.

The next step is to write a function to process a simple command, which is terminated by `|`, `&`, `;`, or a newline. We'll call this function `command`. Arguments are simply entered into an `argv` array for later use in a call to `execvp`. There are three possibilities for the standard input: the default (`STDIN_FILENO`), a file (if `<` was present), or the reading end of a pipe (if this simple command was *preceded* by `|`). The standard output has four possibilities: the default (`STDOUT_FILENO`), a file to be created or truncated (if `>` was present), a file to be created or appended to (if `>>` was present), or the writing end of a pipe (if this simple command was *followed* by a `|`). The tokens received from `gettoken` tell us which possibilities obtain.

As `command` processes the tokens, it uses these variables to record the standard input and output situation for a simple command:

`srcfd` The source file descriptor, initially `STDIN_FILENO`. If input is redirected with <, `srcfd` is set to –1 and `srcfile` records the file name. If the input is a pipe, `srcfd` will be set to a file-descriptor number other than `STDIN_FILENO`.

`srcfile` The source file, used only when the simple command includes a <.

`dstfd` The destination file descriptor, initially `STDOUT_FILENO`. If output is redirected with > or >>, it is set to –1 and `dstfile` records the file name. Like `srcfd`, it's set to a file descriptor other than `STDOUT_FILENO` if the output is a pipe.

`dstfile` The output file, used only when the simple command includes a > or >>.

`append` This Boolean variable is set to `true` only if output was redirected with >>.

`makepipe` This is an argument to `command`. If `true`, the caller is requesting that `command` make a pipe, use the reading end as its standard input, and pass the file descriptor for the writing end back to the caller, who will use it for its standard output. This scheme will be explained shortly.

As it goes through the tokens, `command` also checks for logical errors: two occurrences of < or >, the occurrence of > when the simple command terminates with |, the occurrence of < when the simple command is preceded with |, and so on. It's interesting that the typical UNIX shell doesn't check for some of these anomalies: You can actually run a command line like this:

```
$ who >outf | wc
```

The function `command` returns the token that terminated the pipeline because the type of terminator affects later processing: If the terminator is `&`, the rightmost simple command is not waited for, and all simple commands in the pipeline are run with interrupt and quit signals ignored. If the rightmost simple command is to be waited for, its process-ID must be returned, too, through the argument `wpid`. If the terminator is a newline, the user is prompted for a new command.

The most subtle thing about `command` is that it calls itself recursively when a simple command is followed by |. Recursive calls continue until one of the other terminators (`;`, `&`, or newline) is reached. Each recursive call is responsible for actually making the pipe (with the `pipe` system call), and its argument `makepipe`

is therefore set to `true`. The writing pipe file descriptor is passed back (through another argument, `pipefdp`). A simple command is not actually invoked (with the function `invoke`) until after the recursive call returns, since it can't be invoked until the writing end of the pipe is available. Also, as stated above, the pipeline terminator must be known before any of the constituent simple commands can be invoked so that `invoke` will know what to do about signals. Thus pipelines are processed from left to right, but the simple commands are invoked from right to left. There is one call to `command`, with `makepipe` set to `false`, for the leftmost simple command; and then one more recursive call, with `makepipe` set to `true`, for each additional simple command. As the stack of command calls returns, the simple commands are invoked—all needed information is available then.

Here is the code for `command`. It's worth careful study.

```
#define MAXARG 50       /* max args in command */
#define MAXFNAME 500    /* max chars in file name */
#define MAXWORD 500     /* max chars in arg */

static TOKEN command(pid_t *wpid, bool makepipe, int *pipefdp)
{
    TOKEN token, term;
    int argc, srcfd, dstfd, pid, pfd[2] = {-1, -1};
    char *argv[MAXARG], srcfile[MAXFNAME] = "", dstfile[MAXFNAME] = "";
    char word[MAXWORD];
    bool append;

    argc = 0;
    srcfd = STDIN_FILENO;
    dstfd = STDOUT_FILENO;
    while (true) {
        switch (token = gettoken(word, sizeof(word))) {
        case T_WORD:
            if (argc >= MAXARG - 1) {
                fprintf(stderr, "Too many args\n");
                continue;
            }
            if ((argv[argc] = malloc(strlen(word) + 1)) == NULL) {
                fprintf(stderr, "Out of arg memory\n");
                continue;
            }
            strcpy(argv[argc], word);
            argc++;
            continue;
```

```
case T_LT:
    if (makepipe) {
        fprintf(stderr, "Extra <\n");
        break;
    }
    if (gettoken(srcfile, sizeof(srcfile)) != T_WORD) {
        fprintf(stderr, "Illegal <\n");
        break;
    }
    srcfd = -1;
    continue;
case T_GT:
case T_GTGT:
    if (dstfd != STDOUT_FILENO) {
        fprintf(stderr, "Extra > or >>\n");
        break;
    }
    if (gettoken(dstfile, sizeof(dstfile)) != T_WORD) {
        fprintf(stderr, "Illegal > or >>\n");
        break;
    }
    dstfd = -1;
    append = token == T_GTGT;
    continue;
case T_BAR:
case T_AMP:
case T_SEMI:
case T_NL:
    argv[argc] = NULL;
    if (token == T_BAR) {
        if (dstfd != STDOUT_FILENO) {
            fprintf(stderr, "> or >> conflicts with |\n");
            break;
        }
        term = command(wpid, true, &dstfd);
        if (term == T_ERROR)
            return T_ERROR;
    }
    else
        term = token;
    if (makepipe) {
        ec_neg1( pipe(pfd) )
        *pipefdp = pfd[1];
        srcfd = pfd[0];
    }
    ec_neg1( pid = invoke(argc, argv, srcfd, srcfile, dstfd,
      dstfile, append, term == T_AMP, pfd[1]) )
    if (token != T_BAR)
        *wpid = pid;
```

```
            if (argc == 0 && (token != T_NL || srcfd > 1))
                fprintf(stderr, "Missing command\n");
            while (--argc >= 0)
                free(argv[argc]);
            return term;
        case T_EOF:
            exit(EXIT_SUCCESS);
        case T_ERROR:
            return T_ERROR;
        }
    }

EC_CLEANUP_BGN
    return T_ERROR;
EC_CLEANUP_END
}
```

Once `command` has sensed the need for a pipe by encountering the token `T_BAR`, why does it ask the *next* call to `command` to make the pipe instead of making it itself and just passing on the reading file descriptor? It is to conserve file descriptors. If the `pipe` system call were called before the recursive call to `command`, instead of after, each level of recursion—each simple command—would tie up two pipe file descriptors that could not be closed until after the recursive call returned. Since on some systems file descriptors are in short supply, pipelines would be limited to somewhat less than half that number of simple commands. By asking the reader to make the pipe, we can call `pipe` just before we call `invoke` (which we'll get to shortly), if our caller asked us to make one, thereby allowing pipelines to be of any length.

The child that `invoke` will create has no need for the writing end of the pipe (`pfd[1]`), so it's passed as the last argument to `invoke` so it can be closed in the child; we'll see this a bit later.

Here's the `main` program that makes the first call to `command`. It's modeled on the `main` program for the one-shot shell in Section 5.4.

```
int main(void)
{
    pid_t pid;
    TOKEN term = T_NL;

    ignore_sig();
    while (true) {
        if (term == T_NL)
            printf("%s", PROMPT);
        term = command(&pid, false, NULL);
```

```
        if (term == T_ERROR) {
            fprintf(stderr, "Bad command\n");
            EC_FLUSH("main--bad command")
            term = T_NL;
        }
        if (term != T_AMP && pid > 0)
            wait_and_display(pid);
        fd_check();
    }
}
```

The call to `ignore_sig` causes interrupt and quit signals to be ignored—we don't want to kill our shell when we hit the interrupt or quit keys. We'll show the code for `ignore_sig` in Section 9.1.6. The pipeline terminator returned by `command` tells us whether to wait for the rightmost simple command to terminate (we'll see this version of `wait_and_display` shortly) and whether to prompt.

We want to make sure that with all the redirection and piping we don't miss closing any file descriptors, so we call `fd_check` each time we process a command to make sure that only the standard file descriptors are open. (If we wanted to, we could remove `fd_check` from a production version of this shell.) We check only the first 20 file descriptors, as that's enough to reveal any failure-to-close bugs:

```
static void fd_check(void)
{
    int fd;
    bool ok = true;

    for (fd = 3; fd < 20; fd++)
        if (fcntl(fd, F_GETFL) != -1 || errno != EBADF) {
            ok = false;
            fprintf(stderr, "*** fd %d is open ***\n", fd);
        }
    if (!ok)
        _exit(EXIT_FAILURE);
}
```

`command` calls `invoke` to invoke a simple command. It passes on the command arguments (`argc` and `argv`) and the source and destination variables described earlier (`srcfd`, `srcfile`, `dstfd`, `dstfile`, and `append`). The next-to-last argument tells invoke whether the simple command is to be run in the background; the last argument is the file descriptor to close in the child, as we already described. The `fork` and `exec` scheme used by `invoke` should by now be familiar:

```
static pid_t invoke(int argc, char *argv[], int srcfd, const char *srcfile,
  int dstfd, const char *dstfile, bool append, bool bckgrnd, int closefd)
{
    pid_t pid;
    char *cmdname, *cmdpath;

    if (argc == 0 || builtin(argc, argv, srcfd, dstfd))
        return 0;
    switch (pid = fork()) {
    case -1:
        fprintf(stderr, "Can't create new process\n");
        return 0;
    case 0:
        if (closefd != -1)
            ec_neg1( close(closefd) )
        if (!bckgrnd)
            ec_false( entry_sig() )
        redirect(srcfd, srcfile, dstfd, dstfile, append, bckgrnd);
        cmdname = strchr(argv[0], '/');
        if (cmdname == NULL)
            cmdname = argv[0];
        else
            cmdname++;
        cmdpath = argv[0];
        argv[0] = cmdname;
        execvp(cmdpath, argv);
        fprintf(stderr, "Can't execute %s\n", cmdpath);
        _exit(EXIT_FAILURE);
    }
    /* parent */
    if (srcfd > STDOUT_FILENO)
        ec_neg1( close(srcfd) )
    if (dstfd > STDOUT_FILENO)
        ec_neg1( close(dstfd) )
    if (bckgrnd)
        printf("%ld\n", (long)pid);
    return pid;

EC_CLEANUP_BGN
    if (pid == 0)
        _exit(EXIT_FAILURE);
    return -1;
EC_CLEANUP_END
}
```

The horsing around with `cmdname` and `cmdpath` is caused by the need to keep the whole path that might have been typed as the first argument to `execvp`, but to make `argv[0]` point to only the file-name part.

The command to be invoked may be built in. If so, `builtin` (to be shown shortly) returns `true`. If not, a child process is created to run the simple command. Recall that interrupt and quit signals are already ignored. If the command is not to be run in the background, they are restored to the way they were on entry to the shell with a call to `entry_sig` (detailed in Section 9.1.6).

`invoke` calls `redirect` to redirect I/O and to ensure that the source and destination are duped to be `STDIN_FILENO` and `STDOUT_FILENO`, if necessary. Here is the code for `redirect`:

```
static void redirect(int srcfd, const char *srcfile, int dstfd,
  const char *dstfile, bool append, bool bckgrnd)
{
    int flags;

    if (srcfd == STDIN_FILENO && bckgrnd) {
        srcfile = "/dev/null";
        srcfd = -1;
    }
    if (srcfile[0] != '\0')
        ec_neg1( srcfd = open(srcfile, O_RDONLY, 0) )
    ec_neg1( dup2(srcfd, STDIN_FILENO) )
    if (srcfd != STDIN_FILENO)
        ec_neg1( close(srcfd) )
    if (dstfile[0] != '\0') {
        flags = O_WRONLY | O_CREAT;
        if (append)
            flags |= O_APPEND;
        else
            flags |= O_TRUNC;
        ec_neg1( dstfd = open(dstfile, flags, PERM_FILE) )
    }
    ec_neg1( dup2(dstfd, STDOUT_FILENO) )
    if (dstfd != STDOUT_FILENO)
        ec_neg1( close(dstfd) )
    fd_check();
    return;

EC_CLEANUP_BGN
    _exit(EXIT_FAILURE); /* we are in child */
EC_CLEANUP_END
}
```

If a background command does not have its standard input redirected to come from a file or a pipe, we take the precaution of redirecting it to the special file /dev/null, which gives an immediate end-of-file when read. The rest of `redirect` should be clear.

`wait_and_display` (called from `main`) is an extension to code presented in Section 5.8. It's told what process to wait for, but during the wait it may learn of other terminations. If so, it uses `display_status` to print their process IDs and a description of the reason for their termination. When the designated process terminates, `wait_and_display` returns after displaying its status. Here's the code:

```
static bool wait_and_display(pid_t pid)
{
    pid_t wpid;
    int status;

    do {
        ec_neg1( wpid = waitpid(-1, &status, 0) )
        display_status(wpid, status);
    } while (wpid != pid);
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

Finally, here is `builtin`. Most of it is taken from Section 5.4; we've added code to handle the `cd` command. `cd` without arguments changes to the directory whose path is given by the `HOME` environment variable.

```
static bool builtin(int argc, char *argv[], int srcfd, int dstfd)
{
    char *path;

    if (strchr(argv[0], '=') != NULL)
        asg(argc, argv);
    else if (strcmp(argv[0], "set") == 0)
        set(argc, argv);
    else if (strcmp(argv[0], "cd") == 0) {
        if (argc > 1)
            path = argv[1];
        else if ((path = getenv("HOME")) == NULL)
            path = ".";
        if (chdir(path) == -1)
            fprintf(stderr, "%s: bad directory\n", path);
    }
    else
        return false;
    if (srcfd != STDIN_FILENO || dstfd != STDOUT_FILENO)
        fprintf(stderr, "Illegal redirection or pipeline\n");
    return true;
}
```

You may have observed that some errors discovered by our shell are handled by the heavy "ec" machinery, while others just cause a message to be printed, after which the shell keeps running. We've treated the "impossible" errors as serious ones, since if they occur it means that the operating system is mortally wounded. This is a compromise: We certainly don't want to ignore these errors, because the impossible has been known to happen, but we don't want to code to recover from a situation that will probably never occur. Sometimes we guess wrong—a serious error keeps occurring because it's not impossible after all. Then we have to revise the code to handle the error differently.

I won't show any example of this shell in use. I don't have to because it behaves just like those of the shell you use every day.

6.5 Two-Way Communication with Unidirectional Pipes

Now we want to move beyond one-way pipeline communication, as used by the shell, to two-way. The typical shell offers no notation to set up two-way communication between processes. Two-way pipelines are set up from C programs.

We'll start with a fairly simple example. From within a program we want to invoke the `sort` command to sort some data. Of course, we could do it like this:

```
system("sort <datafile >outfile");
```

Then we could read the contents of outfile to access the sorted data. We don't want to do it that way, however—we want to pipe the data to `sort` and have `sort` pipe the sorted output back to us. Since `sort` can read and write its standard input and output (it's a filter), we should be able to use it the way we want. We already know how to force an arbitrary file descriptor to be the standard input or output of a process.

Just as a baby learns to fall down before learning to walk, we'll begin by doing the job incorrectly. That way we'll learn more about how to do it properly than if we just presented the solution right away.

Since every pipe has both a reading end and a writing end, and since both file descriptors are inherited by a child process, we'll use just one pipe. `sort` will

read it to get its input and write it to send back the sorted output. The parent has file descriptors that access the pipe too. It will write the unsorted data to the pipe and read the sorted data from the pipe. Here's a program that reads data from the file datafile and invokes sort to sort it; the sorted data is printed:

```
void fsort0(void)  /* wrong */
{
    int pfd[2], fd;
    ssize_t nread;
    pid_t pid;
    char buf[512];

    ec_neg1( pipe(pfd) )
    ec_neg1( pid = fork() )
    if (pid == 0) { /* child */
        ec_neg1( dup2(pfd[0], STDIN_FILENO) )
        ec_neg1( close(pfd[0]) )
        ec_neg1( dup2(pfd[1], STDOUT_FILENO) )
        ec_neg1( close(pfd[1]) )
        execlp("sort", "sort", (char *)NULL);
        EC_FAIL
    }
    /* parent */
    ec_neg1( fd = open("datafile", O_RDONLY) )
    while (true) {
        ec_neg1( nread = read(fd, buf, sizeof(buf)) )
        if (nread == 0)
            break;
        ec_neg1( write(pfd[1], buf, nread) )
    }
    ec_neg1( close(fd) )
    ec_neg1( close(pfd[1]) )
    while (true) {
        ec_neg1( nread = read(pfd[0], buf, sizeof(buf)) )
        if (nread == 0)
            break;
        ec_neg1( write(STDOUT_FILENO, buf, nread) )
    }
    ec_neg1( close(pfd[0]) )
    ec_neg1( waitpid(pid, NULL, 0) )
    return;

EC_CLEANUP_BGN
    EC_FLUSH("fsort0");
EC_CLEANUP_END
}
```

This is what's in datafile:

```
peach
apple
orange
strawberry
plum
pear
cherry
banana
apricot
tomato
pineapple
mango
```

When I ran this program, it printed the fruits just as they appeared in datafile—unsorted—and then hung. I had to hit the interrupt key to kill it. What went wrong?

There were two problems with using just one pipe. First of all, after writing the unsorted data to the pipe, the parent immediately began to read the pipe, assuming that it would read the output of `sort`. It got there before `sort` did, however, and just read its own output right back! So the data was printed in its unsorted form. This is reminiscent of the first example in Section 6.2.3.

The second problem caused the deadlock. The child process, running `sort`, began to read its standard input, which happened to be empty since its parent had already emptied it. Empty or not, `sort` would have blocked in a `read` system call waiting for an end-of-file, which would occur only when the writing end was closed. Sure enough, the parent had already closed the writing end, but the *child* still had it open—after all, the child was supposed to write its output there. So the child was stuck. Generally, any filter tricked into reading and writing the same pipe will deadlock.

One might try to fix the first problem by having the parent wait for the child to terminate before reading the pipe. At first, this sounds appealing. It won't work, however, if the child's output fills the pipe, which it might if we're sorting a large amount of data, because the child will block in the `write` system call. With the parent blocked in `waitpid`, we'll have deadlock again.

Next, one might try some semaphores to synchronize things without causing deadlock, but this is overkill. The problem goes away completely if one just uses *two* pipes, each of which handles only one-way traffic. One pipe handles data

flowing into `sort`, and the other pipe handles data flowing back. Here is a rewrite of `fsort0` that works correctly:

```
void fsort(void)
{
    int pfdout[2], pfdin[2], fd;
    ssize_t nread;
    pid_t pid;
    char buf[512];

    ec_neg1( pipe(pfdout) )
    ec_neg1( pipe(pfdin) )
    ec_neg1( pid = fork() )
    if (pid == 0) { /* child */
        ec_neg1( dup2(pfdout[0], STDIN_FILENO) )
        ec_neg1( close(pfdout[0]) )
        ec_neg1( close(pfdout[1]) )
        ec_neg1( dup2(pfdin[1], STDOUT_FILENO) )
        ec_neg1( close(pfdin[0]) )
        ec_neg1( close(pfdin[1]) )
        execlp("sort", "sort", (char *)NULL);
        EC_FAIL
    }
    /* parent */
    ec_neg1( close(pfdout[0]) )
    ec_neg1( close(pfdin[1]) )
    ec_neg1( fd = open("datafile", O_RDONLY) )
    while (true) {
        ec_neg1( nread = read(fd, buf, sizeof(buf)) )
        if (nread == 0)
            break;
        ec_neg1( write(pfdout[1], buf, nread) )
    }
    ec_neg1( close(fd) )
    ec_neg1( close(pfdout[1]) )
    while (true) {
        ec_neg1( nread = read(pfdin[0], buf, sizeof(buf)) )
        if (nread == 0)
            break;
        ec_neg1( write(STDOUT_FILENO, buf, nread) )
    }
    ec_neg1( close(pfdin[0]) )
    ec_neg1( waitpid(pid, NULL, 0) )
    return;

EC_CLEANUP_BGN
    EC_FLUSH("fsort");
EC_CLEANUP_END
}
```

Now we get:

```
apple
apricot
banana
cherry
mango
orange
peach
pear
pineapple
plum
strawberry
tomato
```

Not only is the list sorted, but the program stopped all by itself! The interesting thing about the correct version is that although it uses one more pipe, no additional file descriptors are consumed. In both versions parent and child need to read and write to each other, so each needs two pipe file descriptors. In the second version the parent can close the reading end of `pfdout` and the writing end of `pfdin`.

In general, deadlock is still possible with two pipes, although not in our example using `sort`. Deadlock would occur if the parent blocks writing its output pipe because it is full, and the child, instead of emptying it, writes enough output back to the parent to block on the other pipe. Each case must be examined carefully to ensure that it is always deadlock free, and not only for small amounts of test data.

There are other ways to get deadlocked. To explore one of these, we'll look at a more complex example of two-way interprocess communication. Here the child is the standard line editor, `ed`. The parent is using the editor as a server sending editor command lines to it and getting the output back. Such an arrangement might be used by a visual editor, where the parent handles the keyboard and screen but lets `ed` do the actual editing. I don't have space to show a visual editor, so my example will have to be ridiculously simple. It's an interactive search program much like `grep`. Here's a sample session (what I typed is underlined):

```
$ search

File? datafile

Search pattern? ^a
apple
apricot
```

```
Search pattern? apple
apple
pineapple

Search pattern? o$
tomato
mango

Search pattern? EOT
$
```

The fruit data file is from the sorting examples, above.

When I used `sort`, I knew when to stop reading its output: when I reached an end-of-file. The situation was simple because `sort` reads all its input, writes all its output, and then terminates. The editor, however, is interactive. It reads some input, may or may not write some output of indeterminate length, and then goes back to read some more input. We can capture its output by making its standard output a pipe, but how do we know how much to read at a time? We can't wait for an end-of-file, for the editor won't close its output file descriptor until it terminates. If we read too far we'll deadlock, and if we don't read far enough we'll lose synchronization between a command and its results.

If we could change the editor, we would have it send an unambiguous line of data whenever it's ready for more input. In fact, the editor does have the ability to prompt the user for input (enabled by the `P` command), but the prompt character is `*`, which is hardly unambiguous. (Although you can change the prompt; see Exercise 6.12.) Somehow we need to get the editor to tell us when it's done outputting the results of each command.

We'll use a trick, a kludge if there ever was one: After each command, issue an `r` (read file) command with a nonexistent file name and look for the resulting error message from `ed`. When this message shows up, we know the editor has responded to the bad `r` command and is ready for another command. The nonexistent file name should be such that the error message is unambiguous. It's probably best to use a name containing control characters, which rarely appear in text files, but for clarity we'll use the name "end-of-file." To see the exact form of the message, we run the editor:

```
$ ed
r end-of-file
?end-of-file
q
$
```

So we have to look for "?end-of-file."

To make things easier, we'll code functions to handle interaction with the editor. At the start of processing we call `edinvoke` to invoke the editor and set up the pipes. Instead of using the file descriptors directly, which would require us to do I/O with `read` and `write`, we create Standard C `FILE` pointers instead, using `fdopen`. We can then write to the editor on `sndfp` and read from the editor on `rcvfp`. Here is `edinvoke`, which is a lot like the first part of `fsort`:

```
static FILE *sndfp, *rcvfp;

static bool edinvoke(void)
{
    int pfdout[2], pfdin[2];
    pid_t pid;

    ec_neg1( pipe(pfdout) )
    ec_neg1( pipe(pfdin) )
    switch (pid = fork()) {
    case -1:
        EC_FAIL
    case 0:
        ec_neg1( dup2(pfdout[0], STDIN_FILENO) )
        ec_neg1( dup2(pfdin[1], STDOUT_FILENO) )
        ec_neg1( close(pfdout[0]) )
        ec_neg1( close(pfdout[1]) )
        ec_neg1( close(pfdin[0]) )
        ec_neg1( close(pfdin[1]) )
        execlp("ed", "ed", "-", (char *)NULL);
        EC_FAIL
    }
    ec_neg1( close(pfdout[0]) )
    ec_neg1( close(pfdin[1]) )
    ec_null( sndfp = fdopen(pfdout[1], "w") )
    ec_null( rcvfp = fdopen(pfdin[0], "r") )
    return true;

EC_CLEANUP_BGN
    if (pid == 0) {
        EC_FLUSH("edinvoke");
        _exit(EXIT_FAILURE);
    }
    return false;
EC_CLEANUP_END
}
```

Note that we called `_exit` on an error in the child, rather than `exit`, but then had to call `EC_FLUSH` to get the error message displayed, exactly as we did in Section 5.6.

To write and read the pipe we use `edsnd`, `edrcv`, and `turnaround`. `edrcv` returns `false` when no more editor output is available, which fact it knows because it has found the forced error message. (It treats an actual end-of-file, indicated by a `NULL` return from `fgets`, as an error.) To make sure the error message is there, `turnaround` must be called to switch from sending to receiving. Here are these three functions:

```
static bool edsnd(const char *s)
{
    ec_eof( fputs(s, sndfp) )
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}

static bool edrcv(char *s, size_t smax)
{
    ec_null( fgets(s, smax, rcvfp) )
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}

static bool turnaround(void)
{
    ec_false( edsnd("r end-of-file\n") )
    ec_eof( fflush(sndfp) )
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

We flush `sndfp` so that the editor gets all our output because normally output to pipes is buffered.

Since we'll usually want to display everything the editor has to say, a function to do that will come in handy:

```
static bool rcvall(void)
{
    char s[200];

    ec_false( turnaround() )
    while (true) {
        ec_false( edrcv(s, sizeof(s)) )
        if (strcmp(s, "?end-of-file\n") == 0)
            break;
        printf("%s", s);
    }
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

Last, the `main` program that conducts the dialogue with the user and communicates with the editor:

```
int main(void)
{
    char s[100], line[200];
    bool eof;

    ec_false( prompt("File", s, sizeof(s), &eof) )
    if (eof)
        exit(EXIT_SUCCESS);
    ec_false( edinvoke() )
    snprintf(line, sizeof(line), "e %s\n", s);
    ec_false( edsnd(line) )
    ec_false( rcvall() );
    while (true) {
        ec_false( prompt("Search pattern", s, sizeof(s), &eof) )
        if (eof)
            break;
        snprintf(line, sizeof(line), "g/%s/p\n", s);
        ec_false( edsnd(line) )
        ec_false( rcvall() );
    }
    ec_false( edsnd("q\n") )
    exit(EXIT_SUCCESS);

EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
EC_CLEANUP_END
}
```

```
static bool prompt(const char *msg, char *result, size_t resultmax,
  bool *eofp)
{
    char *p;

    printf("\n%s? ", msg);
    if (fgets(result, resultmax, stdin) == NULL) {
        if (ferror(stdin))
            EC_FAIL
        *eofp = true;
    }
    else {
        if ((p = strrchr(result, '\n')) != NULL)
            *p = '\0';
        *eofp = false;
    }
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

Is this program deadlock free? Since we know that the capacity of a pipe is at least 512 bytes, the parent will never block while writing to the editor's input pipe because our editor commands are quite short. The editor will never block, except temporarily, writing back to the parent because the parent will read everything it has to say. There's no reason why the bad-file message should be missed, unless somebody actually makes a file with that name, which they had better not. Not much of a proof, but at least the obvious pitfalls have been avoided.

The example output shown was generated on Solaris. The program didn't work at all on FreeBSD, Darwin, or Linux because their version of `ed` is different. See Exercise 6.12 if you're curious about the details.

6.6 Two-Way Communication with Bidirectional Pipes

I said when I first presented pipes (Section 6.2.1) that the first file descriptor (`pfd[0]`) is opened for reading, the second (`pfd[1]`) is opened for writing, and whatever is written on `pfd[1]` can be read from `pfd[0]`. Then I showed several useful examples where one process is the writer and another the reader, including some using two pipes (each unidirectional) for two-way communication, much like a divided highway made up, in effect, of two one-way roads.

Everything I said about pipes is true, but that doesn't mean that the file descriptors are opened *only* for reading or *only* for writing. To see what the story is, here's a little program that shows the access mode for a pipe's file descriptors:

```
void pipe_access_mode(void)
{
    int pfd[2], flags, i;

    ec_neg1( pipe(pfd) )
    for (i = 0; i < 2; i++) {
        ec_neg1( flags = fcntl(pfd[i], F_GETFL) )
        if ((flags & O_ACCMODE) == O_RDONLY)
            printf("pfd[%d] O_RDONLY\n", i);
        if ((flags & O_ACCMODE) == O_WRONLY)
            printf("pfd[%d] O_WRONLY\n", i);
        if ((flags & O_ACCMODE) == O_RDWR)
            printf("pfd[%d] O_RDWR\n", i);
    }
    return;

EC_CLEANUP_BGN
    EC_FLUSH("pipe_access_mode")
EC_CLEANUP_END
}
```

On Linux I got what you would probably expect:

```
pfd[0] O_RDONLY
pfd[1] O_WRONLY
```

But look what FreeBSD and Solaris said:

```
pfd[0] O_RDWR
pfd[1] O_RDWR
```

Both ends of the pipe are opened for reading and writing! It turns out that on those two systems (and on others) anything written on `pfd[0]` can be read on `pfd[1]`, anything written on `pfd[1]` can be read on `pfd[0]`, and the data is never mixed up. In fact, on those systems a single pipe is a divided highway, with no collisions. This means that we can set up two-way communication with only one pipe because it's bidirectional.

To see this, here's a revised version of `edinvoke` from the last example in the previous section:

```
static FILE *sndfp, *rcvfp;

static bool edinvoke(void)
{
    int pfd[2];
    pid_t pid;

    ec_neg1( pipe(pfd) )
    switch (fork()) {
    case -1:
        EC_FAIL
    case 0:
        ec_neg1( dup2(pfd[0], STDIN_FILENO) )
        ec_neg1( dup2(pfd[0], STDOUT_FILENO) )
        ec_neg1( close(pfd[0]) )
        execlp("ed", "ed", "-", (char *)NULL);
        EC_FAIL
    }
    ec_null( sndfp = fdopen(pfd[1], "w") )
    ec_null( rcvfp = fdopen(pfd[1], "r") )
    return true;

EC_CLEANUP_BGN
    if (pid == 0) {
        EC_FLUSH("edinvoke");
        _exit(EXIT_FAILURE);
    }
    return false;
EC_CLEANUP_END
}
```

That's all I changed—the rest of the program is identical. It worked exactly as before, but with one pipe instead of two.

There are two disadvantages of using bidirectional pipes instead of using two pipes unidirectionally:

- They're nonstandard, and therefore nonportable.
- A writer who also has to read can't close the writing end of the pipe to simulate an end-of-file because it has only one end. This means, for instance, that the `fsort` example from the previous section can't be written to use only one pipe. (Try it if you don't believe me.)

So, it's best to assume that pipes are unidirectional and to use two of them if you need two-way communication.

Exercises

6.1. Using as many shells as you have access to (e.g., `sh`, `ksh`, `bash`, `csh`), try typing bad command lines to see what error message you get, if any, and what happens. Try these examples for starters:

```
echo abc > f1 > f2
echo def | cat < f1
echo ghi > f1 | cat
cat < f1 < f2
echo jkl | | cat
echo jki | > f1
```

6.2. In the function `gettoken` in Section 6.4, in state `INQUOTE`, when `getchar` is called, not much attention is paid to an end-of-file. Is this a problem? Why or why not?

6.3. Add parameter replacement (e.g., `echo $PATH`) to the shell in Section 6.4.

6.4. Add wild-cards (file-name generation) to the shell in Section 6.4. Use the standard function `glob` if you have it (most systems do). If you don't have it, you can restrict matching to the last component of path names (i.e., `ls */*.c` is illegal).

6.5. Add a `goto` built-in statement to the shell of Section 6.4. Can you implement it as an external command to be run as a child process? (Hint: Review open file descriptions in Section 2.2.3.)

6.6. Add an `if` statement to the shell of Section 6.4.

6.7. Design and implement a menu shell. Instead of prompting for commands, it displays a menu of choices and the user simply picks one. You'll need menus for command arguments, too. Rather than building in the details of various commands, the shell should take its information about commands and their arguments from a database.

6.8. Design and implement a windowing shell. Use Curses if you have access to it. Divide the terminal screen into two halves (which is easier—a horizontal or vertical split?). Run a shell child process in each half. The user presses a function key or a control key (Ctrl-w, say) to select the active window. A command running in the inactive window continues to output to the screen, but when input is requested it blocks until the user activates the window and types a response. A window scrolls when it gets full, and the scrolled-off lines are gone forever. Only line-oriented input and output are supported. Design questions: Should the standard output of a com-

mand go directly to the CRT, to a specially designed filter, or back to the windowing shell? What about input? Optional extras (ranging from hard to extremely hard): The user can scroll a window back to see what disappeared (up to some limit). Screen-oriented output can be handled, as well as line-oriented.

6.9. Write a program that writes its process-ID to the standard output and then reads a list of process-IDs from its standard input. If its own process-ID is on the list, it prints out the list (using file descriptor 2). Otherwise, it adds its process-ID to the list and writes the entire list to its standard output. Then it repeats. Arrange five processes running this program into a ring, and let them play awhile (this is a computer game for a computer to play by itself).

6.10. Using a scheme similar to that of the interactive search program in Section 6.5, write a front-end to the desk calculator `dc` that allows the user to enter expressions in infix notation (2 + 3) instead of postfix (2 3+). (This is what the `bc` command does.)

6.11. Reverse the last example in Section 6.3 so that `wc` is the parent of `who`.

6.12. If you're using FreeBSD, Darwin, or Linux, try to figure out why the search program in Section 6.6 doesn't work. Hint: It generates errors on purpose, which is always a problematical technique, and their version of `ed` doesn't like it. Is this a bug, or is it documented on the `man` page, or both? Is there any way to fix it?

6.13. Extend the program you wrote in Exercise 5.14 to include process attributes from Appendix A that are explained in this chapter.

This page intentionally left blank

7

Advanced Interprocess Communication

7.1 Introduction

So far we've seen how to connect two processes with pipes, but only if the processes are related, and only if the pipes are created before one of the processes, since the only way the file descriptors can be used is if they were inherited. That's much too restrictive for the more realistic situation in which a server is running continuously, and clients that want to talk to it come and go. We want more flexibility in setting up the connections, more control over the information flow (e.g., queues and priorities), and more efficiency if there's a lot of data to be communicated.

Happily, today's UNIX systems have several interprocess communication (IPC) mechanisms besides the pipes we already know about: named pipes, message queues, semaphores and file locks, shared memory, and sockets. This chapter covers all those IPC mechanisms that pass data within a single system; sockets, which go between systems, are in the next chapter.

Most of the IPC mechanisms are flexible enough to accommodate a wide range of application architectures, but it's much easier for a beginner to understand them if we restrict our examples to the simple case of a single server exchanging messages with multiple clients. To make it straightforward to compare the mechanisms, I'll introduce a Simple Messaging Interface (SMI) and then implement it six different ways, using FIFOs (named pipes), System V message queues, POSIX message queues, System V shared memory and semaphores, POSIX shared memory and semaphores, and sockets (in the next chapter). This should give you a solid basis to pick the mechanisms appropriate for your own applica-

tions and then to build on as you investigate the advanced features of the mechanisms you've chosen.

With so many ways to do roughly the same thing, it's natural to wonder which is best. There are at least three dimensions to "best":

- Convenient to use and a good match for your application's needs
- Portable, in case you want to move someday from, say, HP/UX, Solaris, or AIX to Linux
- Efficient

I'll go into the first two criteria in depth as I present each IPC mechanism and summarize things in "critique" sections. Efficiency depends too much on the implementation of the mechanism on the systems of interest to you and how your application uses them, so, if feasible, you might consider implementing your own abstract layer (analogous to SMI). This would allow you to experiment with different mechanisms after your application is running. I'll provide the results of some timing tests at the end of this chapter. Of course, they're definitely not the last word, and the results you get may be different.

7.2 FIFOs, or Named Pipes

A FIFO combines features of a regular file and a pipe. Like a regular file, it has a name, and any process with appropriate permissions may open it for reading or writing. Unlike a pipe, then, unrelated processes may communicate over a FIFO since they need not rely on inheritance alone to access it. Once opened, however, a FIFO acts more like a pipe than a regular file. It follows the pipe behavior described in Section 6.2.2.

Normally, when a FIFO is opened for reading, the `open` waits until it is also opened for writing, usually by another process. Similarly, an `open` for writing blocks until the FIFO is opened for reading. Hence, whichever process, reader or writer, executes the `open` first will wait for the other one. This allows the processes to synchronize themselves before the actual data transmission starts.

If the `O_NONBLOCK` flag is set on `open`, an `open` for reading returns immediately (with an open file descriptor), without waiting for the writer, and an `open` for writing returns –1 with `errno` set to `ENXIO` if no reader has the FIFO open. This asymmetry implies that `O_NONBLOCK` is useful when opening for reading but

awkward when opening for writing since the error has to be processed and then perhaps the `open` has to be retried. The intent is to prevent a process from putting data into a FIFO that will not be read immediately because UNIX has no way of storing data in FIFOs permanently. As with a water pipe, you have no business turning on the water until both ends are soldered in. If any data is still in a FIFO when all readers and writers close their file descriptors, the data is discarded with no error indication. This is like a water pipe too: The water leaks out if you disconnect the pipe at both ends.

The `O_NONBLOCK` flag affects `read` and `write` on FIFOs just as it does on pipes. (Detailed in Section 6.2.2.)

7.2.1 Creating a FIFO

Unlike a regular file, you can't create a FIFO with `open`; you use a separate system call:

mkfifo—make FIFO

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

int mkfifo(
    const char *path,          /* pathname */
    mode_t perms               /* permissions */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

The `perms` argument is used like the third argument to `open`; that is, it is the new file's permissions.

An obvious use for a FIFO is as a replacement for a pipe. Instead of using the `pipe` system call, you make a FIFO and then open it twice, to get read and write file descriptors.[1] From then on you can treat it like a pipe. Used this way, however, FIFOs have absolutely no advantages over pipes and several disadvantages: extra overhead in making the FIFO, two system calls to get file descriptors, and the risk of a name clash just as with a temporary file.

FIFOs weren't added to UNIX to replace *pipes*—they were added to provide a simple way of passing data between server processes and clients. Recall that a limitation of pipes is that the file descriptors used to read and write them can be passed to a process only by inheritance. This won't do if the server is to be kept

1. Opening it for both reading and writing (`O_RDWR`) might work on your system, but it's nonstandard.

running while client processes come and go. Since any process with appropriate permissions can open a FIFO, it's easy to arrange for the server to create a process with a fixed name so clients can open it.

In this chapter we're going to use all of the interprocess communication mechanisms we introduce to pass messages—short pieces of structured data—between processes, rather than streams of data, as we did in Chapter 6. However, bear in mind that you can certainly use a FIFO to pass a stream of data as well.

7.2.2 A Simple FIFO Example

We'll start with two programs, a server and a client. The server runs continuously, waiting for a client to send it a message. In this example, the message is a string to be converted to upper case. The server converts it and sends it back to the client, and then it loops back for the next message, possibly from a different client. The example client sends a few strings to the server, displays the results, and then exits.

We start the server like this:

```
$ smsg_server&
server started
[1] 8725
$
```

Then we run a client like this:

```
$ smsg_client
client 8747 started
client 8747: applesauce --> APPLESAUCE
client 8747: tiger --> TIGER
client 8747: mountain --> MOUNTAIN
Client 8747 done
```

Figure 7.1 shows the arrangement of server, clients, and FIFOs. The server creates a single input FIFO, for use by all clients, named "fifo_server," and each client (two are shown) creates its own FIFO, with the process ID embedded in the name for uniqueness, to receive replies from the server. In the figure, client 8748 has sent a message (represented by a rectangle) containing the data to be converted ("tiger") and its process ID. The server reads that message from its input FIFO, converts the data (to "TIGER"), and then uses the sent process ID (8748) to determine the name of the FIFO to which to write the result message. Thus, there's only one common server FIFO but a separate reply FIFO for each client.

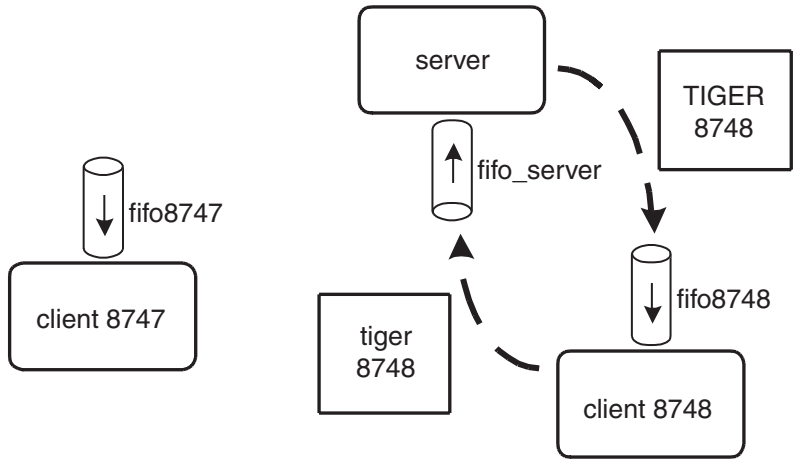


Figure 7.1 Server with two clients.

Client and server have to use the same algorithm for composing a FIFO name from a client process ID, so my example uses a function for that purpose that they share:

```
bool make_fifo_name(pid_t pid, char *name, size_t name_max)
{
    snprintf(name, name_max, "fifo%ld", (long)pid);
    return true;
}
```

A common header file defines the fixed server FIFO name and a structure for messages:

```
#define SERVER_FIFO_NAME "fifo_server"

struct simple_message {
    pid_t sm_clientpid;
    char sm_data[200];
};
```

Here's the code for the server program:

```
int main(void)
{
    int fd_server, fd_client, i;
    ssize_t nread;
    struct simple_message msg;
    char fifo_name[100];
```

```
    printf("server started\n");
    if (mkfifo(SERVER_FIFO_NAME, PERM_FILE) == -1 && errno != EEXIST)
        EC_FAIL
    ec_neg1( fd_server = open(SERVER_FIFO_NAME, O_RDONLY) )
    while (true) {
        ec_neg1( nread = read(fd_server, &msg, sizeof(msg)) )
        if (nread == 0) {
            errno = ENETDOWN;
            EC_FAIL
        }
        for (i = 0; msg.sm_data[i] != '\0'; i++)
            msg.sm_data[i] = toupper(msg.sm_data[i]);
        ec_false( make_fifo_name(msg.sm_clientpid, fifo_name,
          sizeof(fifo_name)) )
        ec_neg1( fd_client = open(fifo_name, O_WRONLY) )
        ec_neg1( write(fd_client, &msg, sizeof(msg)) )
        ec_neg1( close(fd_client) )
    }
    /* never actually get here */
    ec_neg1( close(fd_server) )
    exit(EXIT_SUCCESS);

EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
EC_CLEANUP_END
}
```

Here's what the server does:

- It makes its FIFO, failing only if `mkfifo` returned an error other than the FIFO already existing, as it may be left around from a previous execution of the server.
- It opens its FIFO for reading. This will block until there is a writer, so it's OK to start the server before any client runs.
- Each time through the loop, it reads a message, converts the data, opens the client's FIFO, writes the result message to that FIFO, and then closes it.
- As we didn't provide an explicit way to stop the server, it just stays in the loop indefinitely until it's terminated with the `kill` command (not shown in our examples).

You can probably already imagine what the client program has to do. Here's the code:

```
int main(int argc, char *argv[])
{
    int fd_server, fd_client = -1, i;
    ssize_t nread;
    struct simple_message msg;
    char fifo_name[100];
    char *work[] = {
        "applesauce",
        "tiger",
        "mountain",
        NULL
    };

    printf("client %ld started\n", (long)getpid());
    msg.sm_clientpid = getpid();
    ec_false( make_fifo_name(msg.sm_clientpid, fifo_name,
      sizeof(fifo_name)) )
    if (mkfifo(fifo_name, PERM_FILE) == -1 && errno != EEXIST)
        EC_FAIL
    ec_neg1( fd_server = open(SERVER_FIFO_NAME, O_WRONLY) )
    for (i = 0; work[i] != NULL; i++) {
        strcpy(msg.sm_data, work[i]);
        ec_neg1( write(fd_server, &msg, sizeof(msg)) )
        if (fd_client == -1)
            ec_neg1( fd_client = open(fifo_name, O_RDONLY) )
        ec_neg1( nread = read(fd_client, &msg, sizeof(msg)) )
        if (nread == 0) {
            errno = ENETDOWN;
            EC_FAIL
        }
        printf("client %ld: %s --> %s\n", (long)getpid(),
          work[i], msg.sm_data);
    }
    ec_neg1( close(fd_server) )
    ec_neg1( close(fd_client) )
    ec_neg1( unlink(fifo_name) )
    printf("Client %ld done\n", (long)getpid());
    exit(EXIT_SUCCESS);

EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
EC_CLEANUP_END
}
```

The client does this:

- Makes its FIFO.
- Opens the server FIFO for writing. This will block until the server opens it for reading, so it's OK to start a client before the server is started.

- For each string to be converted, it forms the message, writes it to the server, opens its own client for reading, if necessary, and reads the result.
- After all the strings have been converted, it closes the file descriptors it's opened and unlinks its FIFO since it's unlikely that any other client will use it.

There are two items of bad news to give you: First, client and server both have bugs, of a similar nature, and, second, I lied when I showed the output from the client. In fact, the client's output was this:

```
$ smsg_client
client 8747 started
client 8747: applesauce --> APPLESAUCE
ERROR:  0: main [/aup/c7/smsg_client.c:46] 0
                *** ? (100: "Network is down") ***
$
```

What happened? What's the bug? Well, the location and the `errno` value tell us that the client got an end-of-file from its own FIFO. Why wasn't there data since the client had just sent a message to the server? The problem is one of timing: The server closes its client-FIFO file descriptor after each message and didn't get around to re-opening it before the client went from its `write` to its `read`. So, since there were no file descriptors open for writing, the client got an end-of-file, as explained in Section 6.2.2.

The fix is very easy. If the client maintains a file descriptor open for writing to its own FIFO, even though it will never write, that will force any `read` of its FIFO to block until some data is written there (by the server), which is the behavior we want. So all we have to do to fix the client is to declare another file-descriptor variable like this:

```
int fd_client_w = -1;
```

and then open it along with the reading file descriptor, like this:

```
        if (fd_client == -1)
            ec_neg1( fd_client = open(fifo_name, O_RDONLY) )
        if (fd_client_w == -1)
            ec_neg1( fd_client_w = open(fifo_name, O_WRONLY) )
```

(We should also close it along with `fd_client`.) On most UNIX systems, you can also fix the client with just one file descriptor opened for both reading and writing, but that's nonstandard.

So, rerunning the client with this fix, we now get this output:

```
client 8937 started
client 8937: applesauce --> APPLESAUCE
client 8937: tiger --> TIGER
client 8937: mountain --> MOUNTAIN
Client 8937 done
ERROR:  0: main [/aup/c7/smsg_server.c:33] 0
               *** ? (100: "Network is down") ***
```

Still not right! Now the *server* is complaining that it got an end-of-file from *its* FIFO. The reason is that the client closed the writing end of the server's FIFO when it had no more work, and, as there weren't any other clients running, the server got an end-of-file. We don't want that—we want the server to keep running, ideally blocking in its `read` until there's a message. The fix is similar to the fix for the client—the server maintains a file descriptor open for writing its own FIFO, to avoid an end-of-file. (I won't show the code—you can easily figure it out.)

With the server fixed, not only do both programs run smoothly, but we can even run multiple clients:

```
$ smsg_client & smsg_client & smsg_client & smsg_client
client 9001 started
client 9001: applesauce --> APPLESAUCE
client 9001: tiger --> TIGER
[2] 9001
client 9002 started
client 9002: applesauce --> APPLESAUCE
client 9002: tiger --> TIGER
[3] 9002
[4] 9003
client 9004 started
client 9004: applesauce --> APPLESAUCE
client 9004: tiger --> TIGER
client 9003 started
client 9003: applesauce --> APPLESAUCE
client 9003: tiger --> TIGER
client 9002: mountain --> MOUNTAIN
client 9001: mountain --> MOUNTAIN
client 9004: mountain --> MOUNTAIN
client 9003: mountain --> MOUNTAIN
Client 9001 done
Client 9002 done
Client 9004 done
[2]   Done                    smsg_client
[3]-  Done                    smsg_client
$ Client 9003 done
[4]+  Done                    smsg_client
```

7.2.3 FIFOs Critiqued

The good things about FIFOs are:

- Easy to understand and use because they're really pipes, and you use the basic I/O systems calls (`open`, `read`, `write`, etc.).
- Available in all versions of UNIX.
- Fairly efficient (see Table 7.2 at the end of this chapter).
- Work well for streams of data and for discrete messages. To speed things up, a reader can read multiple messages in one gulp, and a writer can write a whole buffer of messages at once, as long as the maximum for an atomic write isn't exceeded.

The disadvantages are:

- A single FIFO can't have multiple readers because there are no atomicity guarantees when reading (see Section 6.2.2 for more about this problem).
- Data has to be copied from user space in one process to a kernel buffer and then back to another user space, which is expensive. (Message queues and sockets have the same disadvantage.) Thus, FIFOs aren't suitable for the most critical applications.
- If the message size is too big (see Section 6.2.2), a writer may block. If this isn't carefully handled, the application may deadlock.

7.3 An Abstract Simple Messaging Interface (SMI)

It's useful to generalize the messaging sending and receiving from the FIFO example in the previous section into an abstract interface, for two key reasons:

- Critical details like keeping a file descriptor open for writing can be handled by the implementation of the interface, allowing for easier and more reliable application programming.
- Many different implementations can be programmed for different IPC mechanisms. You can experiment with various implementations without changing the application program's source to find the one that works best.

I call the interface I'm going to use SMI, for Simple Messaging Interface.[2]

2. It was designed for this book; it's not part of any standard.

7.3.1 SMI Types and Functions

I'm just going to explain the interface in this section; the specific implementations come later. First, here's the generalized message structure:

struct smi_msg—structure for SMI

```
struct smi_msg {
     long smi_mtype; /* must be first */
     struct client_id {
         long c_id1;
         long c_id2;
     } smi_client;
     char smi_data[1];
};
```

The first two members, `smi_mtype` and `smi_client`, are required by some of the implementations, as we shall see. We've already seen, in the example in Section 7.2.2, that a client has to pass its process ID to the server; in an SMI message, this goes into the `client.cl_id1` member. We'll see `smi_mtype` and `smi_client.c_id2` used later on.

The message data (`smi_data`) can be anything at all, with these restrictions:

- Server and client may be on different machines, so no pointers or other data that isn't meaningful across a network should be passed.
- As different machines have different byte ordering, binary numbers need to be expressed in a network-standard format. This topic is explored in Section 8.1.4.

In this simple interface, server and client have to agree on the fixed message size, and the actual size of the `smi_data` array depends on that size. Once you've mastered the principles in this and the next chapter, you can investigate relaxing that limitation in your own version of the SMI that you use in your own applications.

Before sending or receiving, a server or client has to open a message queue, which should be closed upon termination. Conceptually, sent messages are put on the queue, and received messages are taken from the queue. What a queue actually is is implementation dependent and is hidden inside the SMI implementation, much as the internals of a `FILE` type are hidden by the Standard C I/O functions.

The opening and closing interfaces are:

SMI types—types for SMI

```
typedef void *SMIQ;             /* message queue */
typedef enum {
     SMI_SERVER,                /* server */
     SMI_CLIENT                 /* client */
} SMIENTITY;
#define SERVER_NAME_MAX 50      /* max size of server name */
```

smi_open—open SMI message queue

```
SMIQ *smi_open(
     const char *name,          /* server name */
     SMIENTITY entity,          /* entity being opened */
     size_t msgsize             /* fixed message size */
);
/* Returns pointer to message queue or NULL on error (sets errno) */
```

smi_close—close SMI message queue

```
bool smi_close(
     SMIQ *sqp                  /* queue */
);
/* Returns true on success or false on error (sets errno) */
```

The message-queue name passed to `smi_open` must be known to the server and all clients. If the implementation needs to create it before opening it, this is done behind the scenes inside `smi_open`. There aren't any permissions in this simple interface, which is one of the many reasons why it's called "simple." Whether the message queue is left around after it's closed isn't specified as part of the interface and is also up to the implementation.

The `entity` argument is either `SMI_SERVER` or `SMI_CLIENT`, depending on which kind of program is doing the opening. There can be only one server process but an undefined number of clients.

The `msgsize` argument is the size of the `smi_data` member of the `smi_msg` structure, for all messages. As I mentioned, there's no provision for messages being of different sizes.

For sending and receiving, we could just pass an address of a buffer to simple send and receive functions, something like `write` and `read`, like this (no size argument because it's a fixed number):

```
ec_false( smi_send(sqp, buffer) )
...
ec_false( smi_receive(sqp, buffer) )
```

But if we did that it would require that all implementations copy each message from the sending process to the kernel and again from the kernel to the receiving process. Instead, we'll use a slightly more elaborate interface that uses two calls for sending and two for receiving. The first call of each pair just gets the address of the message, and the second releases access to that address. That way the implementation can keep it in the `SMIQ` structure (implying that it is indeed copied), or keep it in shared memory, or whatever. All the callers know is that they have an address that they can dereference. The release functions are needed because the address may not be good forever—the memory may have to be deallocated, or the shared-memory segment may have to be reused—and because the underlying implementation needs to know when the sender is ready for the message to go.

It's easier to explain receiving before sending:

smi_receive_getaddr—get received SMI message address

```
bool smi_receive_getaddr(
     SMIQ *sqp,              /* queue */
     void **addr             /* message */
);
/* Returns true on success or false on error (sets errno) */
```

smi_receive_release—release received SMI message

```
bool smi_receive_release(
     SMIQ *sqp               /* queue */
);
/* Returns true on success or false on error (sets errno) */
```

The `addr` argument is a pointer to a `void` pointer rather than to a `struct smi_msg` pointer because in practice each application will define its own message structure whose first two members match the first two members of `smi_msg` (`smi_mtype` and `smi_client`).

An application uses these two calls like this:

```
struct my_msg *msg;
...
ec_false( smi_receive_getaddr(sqp, (void **)&msg) )
/* process data in msg->smi_data */
ec_false( smi_receive_release(sqp) )
```

Conceptually, the message is actually received when `smi_receive_getaddr` is called. Note that the application doesn't allocate space for the message; that's done by the implementation.

Sending is very similar, with one added wrinkle—identifying the client:

smi_send_getaddr—get sending SMI message address

```
bool smi_send_getaddr(
     SMIQ *sqp,                 /* queue */
     struct client_id *client,  /* client ID (server only)
     void **addr                /* message */
);
/* Returns true on success or false on error (sets errno) */
```

smi_send_release—release and send SMI message

```
bool smi_send_release (
     SMIQ *sqp                  /* queue */
);
/* Returns true on success or false on error (sets errno) */
```

Conceptually, the actual send occurs when `smi_send_release` is called.

The second argument to `smi_send_getaddr` is used by a server to identify the client it's sending to. It's a pointer to the `struct client_id` that was in a received message. This works because a server receives a message from a client that wants to do business with it before it sends a message to that client. A client that sends doesn't need to identify the server (that was done with `smi_open`), so the second argument is `NULL`.

A server, then, would do something like this:

```
struct my_msg *msg_in, *msg_out;
...
ec_false( smi_receive_getaddr(sqp, (void **)&msg_in) )
/* code to process data in msg_in->smi_data */
ec_false( smi_send_getaddr(sqp, &msg_in->smi_client, (void **)&msg_out) )
ec_false( smi_receive_release(sqp) )
/* code to put data into msg_out->smi_data */
ec_false( smi_send_release(sqp) )
```

Observe that:

- The server used two separate `my_msg` addresses (`msg_in` and `msg_out`) because two different buffers are involved.
- `msg_in` wasn't released until after the call to `smi_send_getaddr` because the `msg_in->smi_client` structure was needed until then. It's OK to interleave the calls in this way; in fact, `smi_receive_release` could have been called even after the call to `smi_send_release`—sending and receiving are totally independent. Alternatively, the `client_id` structure could

have been copied to a temporary variable, the address of which could be used in the call to `smi_send_getaddr`.

A client calls `smi_send_getaddr` and `smi_send_release` like this (there's probably a `msg_in`, but we're not showing it):

```
struct my_msg *msg_out;
...
ec_false( smi_send_getaddr(sqp, NULL, (void **)&msg_out) )
/* code to put data into msg_out->smi_data */
ec_false( smi_send_release(sqp) )
```

The next section has a more complete example.

7.3.2 Example Server and Client Using SMI

To show how the SMI functions are used in an application, here is the server from the example in Section 7.2.2 rewritten to use that interface (the `defines` are in a header included by server and client):

```
#define SERVER_NAME "smsg_server"
#define DATA_SIZE 200

int main(void)
{
    SMIQ *sqp;
    struct smi_msg *msg_in, *msg_out;
    int i;

    printf("server started\n");
    ec_null( sqp = smi_open(SERVER_NAME, SMI_SERVER, DATA_SIZE) )
    while (true) {
        ec_false( smi_receive_getaddr(sqp, (void **)&msg_in) )
        ec_false( smi_send_getaddr(sqp, &msg_in->smi_client,
          (void **)&msg_out) )
        for (i = 0; msg_in->smi_data[i] != '\0'; i++)
            msg_out->smi_data[i] = toupper(msg_in->smi_data[i]);
        msg_out->smi_data[i] = '\0';
        ec_false( smi_receive_release(sqp) )
        ec_false( smi_send_release(sqp) )
    }
    /* never actually get here */
    ec_false( smi_close(sqp) )
    exit(EXIT_SUCCESS);
```

```
EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
EC_CLEANUP_END
}
```

and here's the client:

```
int main(int argc, char *argv[])
{
    SMIQ *sqp;
    struct smi_msg *msg;
    int i;

    char *work[] = {
        "applesauce",
        "tiger",
        "mountain",
        NULL
    };

    printf("client %ld started\n", (long)getpid());
    ec_null( sqp = smi_open(SERVER_NAME, SMI_CLIENT, DATA_SIZE) )
    for (i = 0; work[i] != NULL; i++) {
        ec_false( smi_send_getaddr(sqp, NULL, (void **)&msg) )
        strcpy(msg->smi_data, work[i]);
        ec_false( smi_send_release(sqp) )
        ec_false( smi_receive_getaddr(sqp, (void **)&msg) )
        printf("client %ld: %s --> %s\n", (long)getpid(),
          work[i], msg->smi_data);
        ec_false( smi_receive_release(sqp) )
    }
    ec_false( smi_close(sqp) )
    printf("Client %ld done\n", (long)getpid());
    exit(EXIT_SUCCESS);

EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
EC_CLEANUP_END
}
```

Notice that all the monkey-business with keeping file descriptors opened for writing that we saw in Section 7.2.2 is gone, as are details like FIFO names. Maybe "hidden" would be more accurate than "gone," as we are about to discover.

7.3.3 FIFO Implementation of SMI

All of the complexity of messaging with FIFOs that was removed from the server and client now has to go into the implementation of the SMI functions. In addition, the implementation of the server will try to keep client-FIFO file descriptors

open across messages, to save the overhead of opening and closing them each time, as the example in Section 7.2.2 did.

The FIFO implementation uses function names of the form *smifcn*`_fifo`, where *smifcn* is the SMI name (e.g., `smi_send_getaddr_fifo` is the implementation of `smi_send_getaddr`). A small wrapper file in effect translates the names, with little functions like this:

```
bool smi_send_getaddr(SMIQ *sqp, struct client_id *client, void **addr)
{
    return smi_send_getaddr_fifo(sqp, client, addr);
}
```

Using the wrappers allows two kinds of applications to be linked:

- An implementation-independent application, like the one shown in the previous section, can be written in terms of the generic SMI functions (e.g., `smi_send_getaddr`) and then linked with a particular implementation and the appropriate wrapper to translate the calls from generic to specific.
- An implementation that wants to use several implementations can use the specific function names (e.g., `smi_send_getaddr_fifo`, `smi_send_getaddr_skt`), bypassing the wrappers, which aren't linked in. The main purpose of this is for writing a test program that, for example, runs the same application using different methods to compare their performance. I did exactly that to prepare Table 7.2, which closes this chapter.

I won't show the wrappers in this book, but they're on the Web site [AUP2003].

We'll begin the FIFO implementation with the message queue that it uses internally. Server and client use the same data structure, although each has its own data since they are in separate processes.

```
#define MAX_CLIENTS 20

typedef struct {
    SMIENTITY sq_entity;            /* entity */
    int sq_fd_server;               /* server read and ... */
    int sq_fd_server_w;             /* ... write file descriptors */
    char sq_name[SERVER_NAME_MAX];  /* server name */
    struct {
        int cl_fd;                  /* client file descriptor */
        pid_t cl_pid;               /* client process ID */
    } sq_clients[MAX_CLIENTS];
    struct client_id sq_client;     /* client ID */
    size_t sq_msgsize;              /* msg size */
    struct smi_msg *sq_msg;         /* msg buffer */
} SMIQ_FIFO;
```

There's room for only 20 clients per server. A linked list could be used to remove the limit, but I didn't bother. Both client and server use `sq_fd_server` for the server, and in addition the server uses `sq_fd_server_w` to keep a file descriptor open for writing, as explained in Section 7.2.2. The server uses the `sq_clients` array to hold the client file descriptors to avoid opening and closing them with each incoming message. A client needs just one of them for reading its own FIFO, for which it uses `sq_clients[0]`. It uses `sq_clients[1]` to keep a second file descriptor open for writing. For a client, the rest of the array and `sq_fd_server_w` are wasted space, but I thought it would be easier to use the same queue structure for both, as the wasted space is small.

The client-ID information passed by the server to `smi_send_getaddr_fifo` is kept for use by the following `smi_send_release_fifo` in the `sq_client` member. The size passed to `smi_open_fifo` and a pointer to a buffer for messages are in the `sq_msgsize` and `sq_msg` members. (Note that with FIFOs, as with message queues and sockets, a copy from user space to kernel and back into the receiving process is unavoidable.)

Server and client need to initialize the sq_clients array, with this internal function:

```
static void clients_bgn(SMIQ_FIFO *p)
{
    int i;

    for (i = 0; i < MAX_CLIENTS; i++)
        p->sq_clients[i].cl_fd = -1;
}
```

And, at the end, they call `clients_end`:

```
static void clients_end(SMIQ_FIFO *p)
{
    clients_close_all(p);
}

static void clients_close_all(SMIQ_FIFO *p)
{
    int i;

    for (i = 0; i < MAX_CLIENTS; i++)
        if (p->sq_clients[i].cl_fd != -1) {
            (void)close(p->sq_clients[i].cl_fd);
            p->sq_clients[i].cl_fd = -1;
        }
}
```

We'll see these calls in the `smi_open_fifo` and `smi_close_fifo` functions.

When the server gets a message, it uses `clients_find` to see if it already has a file descriptor open to the FIFO, and to find an available slot if not:

```
static int clients_find(SMIQ_FIFO *p, pid_t pid)
{
    int i, avail = -1;

    for (i = 0; i < MAX_CLIENTS; i++) {
        if (p->sq_clients[i].cl_pid == pid)
            return i;
        if (p->sq_clients[i].cl_fd == -1 && avail == -1)
            avail = i;
    }
    if (avail != -1)
        p->sq_clients[avail].cl_pid = pid;
    return avail;
}
```

This function returns –1 if no more clients can be handled.

The final internal functions are for making the FIFO names. The server's is fixed, while the clients' contain their process IDs, just as in the earlier example in Section 7.2.2.

```
static void make_fifo_name_server(const SMIQ_FIFO *p, char *fifoname,
  size_t fifoname_max)
{
    snprintf(fifoname, fifoname_max, "/tmp/smififo-%s", p->sq_name);
}

static void make_fifo_name_client(pid_t pid, char *fifoname,
  size_t fifoname_max)
{
    snprintf(fifoname, fifoname_max, "/tmp/smififo%ld", (long)pid);
}
```

Now we're ready to look at `smi_open_fifo`:

```
SMIQ *smi_open_fifo(const char *name, SMIENTITY entity, size_t msgsize)
{
    SMIQ_FIFO *p = NULL;
    char fifoname[SERVER_NAME_MAX + 50];

    ec_null( p = calloc(1, sizeof(SMIQ_FIFO)) )
    p->sq_msgsize = msgsize + offsetof(struct smi_msg, smi_data);
    ec_null( p->sq_msg = calloc(1, p->sq_msgsize) )
    p->sq_entity = entity;
```

```
    if (strlen(name) >= SERVER_NAME_MAX) {
        errno = ENAMETOOLONG;
        EC_FAIL
    }
    strcpy(p->sq_name, name);
    make_fifo_name_server(p, fifoname, sizeof(fifoname));
    if (p->sq_entity == SMI_SERVER) {
        clients_bgn(p);
        if (mkfifo(fifoname, PERM_FILE) == -1 && errno != EEXIST)
            EC_FAIL
        ec_neg1( p->sq_fd_server = open(fifoname, O_RDONLY) )
        ec_neg1( p->sq_fd_server_w = open(fifoname, O_WRONLY) )
    }
    else {
        ec_neg1( p->sq_fd_server = open(fifoname, O_WRONLY) )
        make_fifo_name_client(getpid(), fifoname, sizeof(fifoname));
        (void)unlink(fifoname);
        ec_neg1( mkfifo(fifoname, PERM_FILE) )
        ec_neg1( p->sq_clients[0].cl_fd =
          open(fifoname, O_RDONLY | O_NONBLOCK) )
        ec_false( setblock(p->sq_clients[0].cl_fd, true) )
        ec_neg1( p->sq_clients[1].cl_fd = open(fifoname, O_WRONLY) )
    }
    return (SMIQ *)p;

EC_CLEANUP_BGN
    if (p != NULL) {
        free(p->sq_msg);
        free(p);
    }
    return NULL;
EC_CLEANUP_END
}
```

This code should be understandable, especially if you've understood the example in Section 7.2.2. But the parts that actually open the FIFOs are worth reviewing. For the server, the sequence for opening its FIFO is:

```
ec_neg1( p->sq_fd_server = open(fifoname, O_RDONLY) )
ec_neg1( p->sq_fd_server_w = open(fifoname, O_WRONLY) )
```

The first call will block if there is no writer, which is OK because without a writer the server doesn't have anything to do anyway. However, the sequence for a client opening its FIFO is more complicated:

```
ec_neg1( p->sq_clients[0].cl_fd =
  open(fifoname, O_RDONLY | O_NONBLOCK) )
ec_false( setblock(p->sq_clients[0].cl_fd, true) )
ec_neg1( p->sq_clients[1].cl_fd = open(fifoname, O_WRONLY) )
```

Without the `O_NONBLOCK` flag, a client would block in its `open` waiting for the server to open the FIFO for writing. But, the server won't do this until it gets a message, for only then does it even know that this particular client exists. Deadlock! The solution, as shown in the code, is to open the FIFO nonblocking, which means it returns with a valid file descriptor without waiting for a writer. But then we have to clear the `O_NONBLOCK` flag (i.e., set blocking on) so that subsequent `read`s will block, which is the behavior we want. (`setblock`, which uses the `fcntl` system call, came from Section 4.2.2.) In the example in Section 7.2.2, we didn't have to do this because the client didn't open its own FIFO until it had already sent the message to the server. That doesn't work here because the SMI functions are separate, with specific jobs to do. (Sometimes modularization requires extra work.)

Here's the companion `smi_close_fifo` function:

```
bool smi_close_fifo(SMIQ *sqp)
{
    SMIQ_FIFO *p = (SMIQ_FIFO *)sqp;

    clients_end(p);
    (void)close(p->sq_fd_server);
    if (p->sq_entity == SMI_CLIENT) {
        char fifoname[SERVER_NAME_MAX + 50];

        make_fifo_name_client(getpid(), fifoname, sizeof(fifoname));
        (void)unlink(fifoname);
    }
    else
        (void)close(p->sq_fd_server_w);
    free(p->sq_msg);
    free(p);
    return true;
}
```

Note that clients unlink their FIFOs, but the server leaves its around so that in the future a client can be started without the server running.

Here's `smi_send_getaddr_fifo`, which doesn't do much other than save the client ID and return the buffer address:

```
bool smi_send_getaddr_fifo(SMIQ *sqp, struct client_id *client,
  void **addr)
{
    SMIQ_FIFO *p = (SMIQ_FIFO *)sqp;
```

```
    if (p->sq_entity == SMI_SERVER)
        p->sq_client = *client;
    *addr = p->sq_msg;
    return true;
}
```

The real work is in `smi_send_release_fifo`:

```
bool smi_send_release_fifo(SMIQ *sqp)
{
    SMIQ_FIFO *p = (SMIQ_FIFO *)sqp;
    ssize_t nwrite;

    if (p->sq_entity == SMI_SERVER) {
        int nclient = clients_find(p, p->sq_client.c_id1);
        if (nclient == -1 || p->sq_clients[nclient].cl_fd == -1) {
            errno = EADDRNOTAVAIL;
            EC_FAIL
        }
        ec_neg1( nwrite = write(p->sq_clients[nclient].cl_fd, p->sq_msg,
          p->sq_msgsize) )
    }
    else {
        p->sq_msg->smi_client.c_id1 = (long)getpid();
        ec_neg1( nwrite = write(p->sq_fd_server, p->sq_msg,
          p->sq_msgsize) )
    }
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

There's nothing unusual here that didn't appear in the earlier example in Section 7.2.2.

`smi_receive_getaddr_fifo` does all the work of receiving the message:

```
bool smi_receive_getaddr_fifo(SMIQ *sqp, void **addr)
{
    SMIQ_FIFO *p = (SMIQ_FIFO *)sqp;
    ssize_t nread;

    if (p->sq_entity == SMI_SERVER) {
        int nclient;
        char fifoname[SERVER_NAME_MAX + 50];

        while (true) {
            ec_neg1( nread = read(p->sq_fd_server, p->sq_msg,
              p->sq_msgsize) )
```

```
            if (nread == 0) {
                errno = ENETDOWN;
                EC_FAIL
            }
            if (nread < offsetof(struct smi_msg, smi_data)) {
                errno = E2BIG;
                EC_FAIL
            }
            if ((nclient = clients_find(p,
              (pid_t)p->sq_msg->smi_client.c_id1)) == -1) {
                continue; /* client not notified */
            }
            if (p->sq_clients[nclient].cl_fd == -1) {
                make_fifo_name_client((pid_t)p->sq_msg->smi_client.c_id1,
                  fifoname, sizeof(fifoname));
                ec_neg1( p->sq_clients[nclient].cl_fd =
                  open(fifoname, O_WRONLY) )
            }
            break;
        }
    }
    else
        ec_neg1( nread = read(p->sq_clients[0].cl_fd, p->sq_msg,
          p->sq_msgsize) )
    *addr = p->sq_msg;
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

The one sticky part here is:

```
if ((nclient = clients_find(p, (pid_t)mp->smi_client)) == -1) {
    continue; /* client not notified */
}
```

If `clients_find` returns −1, it means that the server got a message from a client but can't get a slot in the `sq_clients` array so it can't open the response FIFO. Since it can't respond, it can't even send an error message. The example code just ignores the error, leaving the client blocked. Some alternatives would be:

- Send a signal to the client, and modify `smi_open_fifo` to arrange to catch it.
- Use an integer variable to hold an emergency file descriptor to open the FIFO so an error message can be sent.
- At least change the client so it times out rather than staying blocked forever.

There's one function left, but it has nothing at all to do:

```
bool smi_receive_release_fifo(SMIQ *sqp)
{
    return true;
}
```

That's the whole implementation. Even with the limitations I've discussed, it's still a highly portable, reasonably reliable implementation of the SMI functions that you can start using right away in your own application, at least for prototyping. Later, you can use one of the other implementations since the interface is identical.

7.4 System V IPC (Interprocess Communication)

As explained in Section 1.1.7, there are two sets of system calls for messages, semaphores, and shared memory. The older set is usually called System V IPC, and the newer set is called POSIX IPC. I'll discuss both in this chapter, first System V and POSIX messages, then both types of semaphores, and finally shared memory. This section explains some general concepts that apply to all of the System V IPC mechanisms; later, in Section 7.6, general POSIX IPC concepts will be explained.

7.4.1 System V IPC Objects

There are three kinds of System V IPC Objects: message queues, semaphore sets, and shared memory segments. They are not files—not even special files—but objects with their own naming scheme, own lifetime rules, and own permission system. Here are the principal characteristics of System V IPC objects:

- They exist only within a single machine. They can't be used for communication across a network.
- Their lifetime is the same as the kernel—they are destroyed on a reboot.
- You access an object with an integer *identifier* that's fixed for the lifetime of the object. Any process that knows the identifier can use it directly to access the object—there's no need to open the object first. This is different from a file descriptor, which is a property of the process and goes away when the process does. (System V IPC also has *keys,* which I'll explain in the next section.)

- There are no i-nodes or pathnames, so none of the traditional file and directory manipulation system calls, such as `unlink`, `stat`, `read`, or `write`, can be used.

All three kinds of objects have system calls of the form *X*`get` and *X*`ctl`, where *X* is `msg`, `sem`, or `shm`. Thus, the six calls are `msgget`, `semget`, `shmget`, `msgctl`, `semctl`, and `shmctl`. I'll continue to use the terms *X*`get` and *X*`ctl` to refer to "get" and "ctl" calls when what I'm discussing applies to all three objects.

There are only five more calls: `msgsnd` and `msgrcv` for sending and receiving messages, `semop` for manipulating semaphore sets, and `shmat` and `smdt` for attaching and detaching shared memory segments.

7.4.2 Identifiers, Keys, and the ftok System Call

An identifier is assigned by the kernel when a System V IPC object is created, so in general it has a different value from reboot to reboot. To make it easier for different processes to get the identifier from an object they need to share, there are also permanent *keys* that never change in value. A process can specify a key when it creates an object with *X*`get` or just use *X*`get` to get the identifier from the key if the object already exists. However, the key doesn't really identify the specific object, in the sense that a pathname identifies a file, because the object lasts only until the next reboot. Next time, the same key is used to generate a new object with a possibly different identifier.

Keys are of type `key_t`, which is not required to even be an arithmetic type, although it is used that way by various system calls, so it's a safe assumption. If you wanted to, you could simply define key values for programs to use, like this:

```
#define MSGQ_KEY   1234
#define SEM_KEY    1235
#define SHM_KEY1   1236
#define SHM_KEY2   1237
```

Then any process that calls `msgget` with the key `MSGQ_KEY` as an argument will get an identifier to the same message queue.

The problem with keys is that some world-wide administration scheme is needed to assign them so there are conflicts between applications. This is clearly unworkable, so another layer of abstraction allows a key to be generated from a pathname with the `ftok` system call.

ftok—generate System V IPC key

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

key_t ftok(
    const char *path,          /* pathname of existing file */
    int id                     /* desired key number */
);
/* Returns key on success or -1 on error (sets errno) */
```

Actually, `ftok` can generate lots of keys from the same path; the desired key is specified by the `id` argument, which could be a character, as only its lowest 8 bits are used. It can't be zero, either. For example, if you need four keys for your application (one message queue, one semaphore set, and two shared memory segments), you can use a single path name (e.g., /tmp/myappkeys) and call `ftok` four times with the second argument taking on the values `q`, `s`, `m`, and `n` (or whatever four values you like). The pathname must already exist—`ftok` will not create it.

Recall that I said that System V IPC objects are not files and do not have i-nodes. The file whose name is passed to `ftok` exists only to provide a global name for key generation. The contents of the file don't matter. It's not guaranteed that if the file is unlinked and re-created the same keys will be generated even though the name stays the same, so you should create the file when the application is installed or run for the first time and leave it around until the application is uninstalled.

Two different paths are guaranteed to generate two different keys only if the paths are in the same file system (Section 3.2.4). If you're worried about different applications accidentally using the same key, you can use the special key `IPC_PRIVATE` in an *X*`get` call (explained further in Section 7.5.1), which guarantees you a unique IPC object without using `ftok` or specific keys at all. You have to somehow publish the resulting identifier so all the processes that need it can get it; one way to do that is to write it to a file whose name is known to those processes. But, for most purposes, including all the examples in this book, we're going to assume that `ftok` conflicts won't occur.

To summarize:

- You access a System V IPC object with an identifier. How you get the identifier doesn't matter—it can be an argument to an `exec`, passed with a message, read from a file, returned from a system call like `msgget`, or whatever.

- If you want (and you usually do), you can refer to an object with a key which, unlike an identifier, remains constant even as the object is destroyed and re-created.
- Because it's hard to manage keys, they can be generated from pathnames with `ftok`, and in practice, this is usually what you'll do. Nonunique keys are a remote possibility; creating `IPC_PRIVATE` objects is a potential solution.

You may be asking yourself, "Why have all this identifier/key/`ftok` stuff when UNIX has always had a perfectly good mechanism involving path name and file descriptors already?" The answer will be clear a bit later when we look at some example code that uses System V IPC messages. We'll see that a server can take an identifier for a client's message queue and immediately use it to send a message back, with no overhead to find and open an i-node, which is what `open` does.

Still, one wonders if a cleaner way could have been designed, one that's efficient but still more tightly integrated into the rest of the file system. We'll see such a way when we look at POSIX IPC messages, although it has its own problems with the way it uses pathnames.

Aside from the complexity and irregularity, another disadvantage of the System V IPC approach is that since identifiers aren't file descriptors, you can't use `select` or `poll` (Section 4.2) to block, for example, until a message is ready. I've already shown one solution to this problem in Section 5.18.

In any case, the System V IPC system calls are now standardized and there's no chance that any improvements will ever be made.

7.4.3 System V IPC Ownership and Permissions

If only System V IPC objects were files, they could use the permission system that files use, and I wouldn't have to explain anything here. But, as they are not files, they have their own system. Fortunately, it's a lot like the one that files use.

Permissions are specified using the usual 9 bits: read, write, and execute for owner, group, and others. However, "execute" doesn't mean anything, so those bits aren't used. The permissions for an object are set when it's created with one of the *X*`get` system calls. Later, you can change the permissions with an *X*`ctl` system call.

Files have an owner user-ID and owner group-ID. System V IPC objects have those two and, in addition, store the user-ID and group-ID of the creator. You can use *X*`ctl` to change the owner IDs but not the creator IDs.

When you manipulate an object, the algorithm for checking the permissions uses the effective user-ID and effective group-ID just as for files, and the "other" permission bits are used only when there's no match for user- or group-IDs. However, the effective user- or group-IDs can match either the creator or owner IDs.

Having separate creator and owner IDs allows an administrative user, say, "dbmsadm," to be the creator of a message queue and also the effective user ID of a database server process that has access to database files. A lesser user, "dbmsuser," say, can access the message queue but not the files.

When you're getting or setting permissions with one of the *X*`ctl` system calls, you use this structure:

struct ipc_perm—structure for System V IPC permissions

```
struct ipc_perm {
     uid_t uid;             /* owner user-ID */
     gid_t gid;             /* owner group-ID */
     uid_t cuid;            /* creator user-ID */
     gid_t cgid;            /* creator group-ID */
     mode_t mode;           /* permission bits */
};
```

7.4.4 System V IPC Utilities

Since System V IPC objects are not files and don't have i-nodes, you can't use system calls like `unlink` or `stat` on them, and, therefore, you can't remove them with `rm` and you won't see them listed with the `ls` command. Instead, there are two commands specifically for manipulating System V IPC objects: `ipcrm`, which removes an object, and `ipcs`, which reports its status. Check [SUS2002] or your system's documentation to see all the various options for these commands; I'll just sketch the basics here.

You call `ipcrm` with one or more argument pairs of the form

-X Y

where *X* is a lower or upper case `Q` for a message queue, `S` for a semaphore set, or `M` for a shared memory segment. If lower case, *Y* is an identifier; if upper case, *Y* is a key.[3] For example, the command

```
ipcrm -q 50
```

removes the message queue whose identifier is 50. When you remove a file, all you're really doing is removing an entry from a directory; the i-node stays around until the last file descriptor open to it is closed, so running processes aren't unduly affected. No such behavior is specified for System V IPC objects, however—the object may go away immediately, causing havoc with running processes that are using it. So, generally, `ipcrm` is used only as a part of off-hours system administration or when you know that the object isn't in use.

The System V IPC equivalent to `ls` is `ipcs`. Used without arguments it displays something like this:

```
IPC status from <running system> as of Wed Mar 12 15:04:06 MST 2003
T         ID      KEY        MODE        OWNER         GROUP
Message Queues:
q         50   0x1007b8c  --rw-rw-rw-     marc sysadmin
Shared Memory:
m          0   0x500004b7 --rw-r--r--     root        root
m        102   0          --rw-rw-rw-     marc sysadmin
m        103   0          --rw-rw-rw-     marc sysadmin
m        104   0          --rw-rw-rw-     marc sysadmin
m        105   0          --rw-rw-rw-     marc sysadmin
Semaphores:
s     131072   0x1007b8d  --ra-ra-ra-     marc sysadmin
s     131073   0          --ra-ra-ra-     marc sysadmin
```

Note that some of the keys are zero. These are private objects, created by *X*`get` with a key argument of `IPC_PRIVATE`, rather than some key. This is done when the identifier will be passed to other processes directly (e.g., as data in a message) and there's no need for the other process to execute its own *X*`get`. We'll see this in the System V message queue implementation of the SMI functions (Section 7.5.3). (`IPC_PRIVATE` can also be used when you want to ensure that the object is unique, as I explained in Section 7.4.2.)

3. That's the standard, anyway. Linux (or maybe it's GNU) does it differently; see your system documentation for details.

7.5 System V Message Queues

Now we're ready for the details of the System V message-queue system calls.

7.5.1 System V Message-Queue System Calls

We already sketched what `msgget` does; here are the particulars:

msgget—get message-queue identifier

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

int msgget(
    key_t key,              /* key */
    int flags               /* creation flags */
);
/* Returns identifier or -1 on error (sets errno) */
```

As I said in Section 7.4, where we discussed System V IPC objects in general, `msgget` gets the identifier for the existing message queue whose key is given by its first argument. If the `flags` argument has the `IPC_CREAT` flag set, it creates the queue if it doesn't already exist. In this case the permissions are taken from the low-order 9 bits of the `flags` argument. The creator and owner user- and group-IDs are taken from the effective IDs of the process that issues the `msgget`.

For the permission bits, you can use the same `S_` flags that you use with `open` (Section 2.3). But make sure you use `IPC_CREAT`; don't use `O_CREAT` by mistake![4] There's also an `IPC_EXCL` flag you can use to make `msgget` fail if the queue already exists.

A `key` argument of `IPC_PRIVATE` allows you to create a message queue that's not associated with a specific key. Each time you call `msgget` with `IPC_PRIVATE` (the `IPC_CREAT` flag isn't needed) you get a different queue. This is ideal for a client process that wants its own message queue for replies from a server; it passes the identifier to the server, which can then use it to reply. There's no need for client and server to share a key, which would be awkward, since usually a server doesn't know in advance what clients it may have.

4. Why didn't the people who designed the System V IPC calls use the same symbols as `open` instead of making up their own symbols? Because back then (mid-1970s), `open` didn't use symbols. So the question should be, why didn't `open` use the same symbols as System V IPC? Because those symbols all started with the prefix `IPC_`, and, besides, the mainstream UNIX community within Bell Labs didn't like the IPC calls.

You can control an existing queue with `msgctl`:

msgctl—control message queue

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

int msgctl(
     int msqid,             /* identifier */
     int cmd,               /* command */
     struct msqid_ds *data  /* data for command */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

struct msqid_ds—structure for msgctl

```
struct msqid_ds {
  struct ipc_perm msg_perm; /* permission structure */
  msgqnum_t msg_qnum;       /* number of messages currently on queue */
  msglen_t msg_qbytes;      /* maximum number of bytes allowed on queue */
  pid_t msg_lspid;          /* process ID of last msgsnd */
  pid_t msg_lrpid;          /* process ID of last msgrcv */
  time_t msg_stime;         /* time of last msgsnd */
  time_t msg_rtime;         /* time of last msgrcv */
  time_t msg_ctime;         /* time of last msgctl change */
};
```

There are three values for the `cmd` argument:

`IPC_RMID` Remove the queue associated with `msqid`. The effective user-ID must be superuser or equal to the creator or owner user-IDs of the queue. The `data` argument isn't used and can be `NULL`.

`IPC_STAT` Fill the structure pointed to by `data` with information about the queue.

`IPC_SET` Set four properties of the queue from these members of the structure pointed to by data:

```
msg_perm.uid
msg_perm.gid
msg_perm.mode
msg_qbytes
```

The same permissions as for `IPC_RMID` are required, except that only the superuser can raise the value of `msg_qbytes`.

You put messages on and take messages from a queue with `msgsnd` and `msgrcv`.

msgsnd—send message

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

int msgsnd(
     int msqid,              /* identifier */
     const void *msgp,       /* message */
     size_t msgsize,         /* size of message */
     int flags               /* flags */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

msgrcv—receive message

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

ssize_t msgrcv(
     int msqid,              /* identifier */
     void *msgp,             /* message */
     size_t mtextsize,       /* size of mtext buffer */
     long msgtype,           /* message type requested */
     int flags               /* flags */
);
/* Returns number of bytes placed in mtext or -1 on error (sets errno) */
```

struct msg—typical structure for msgsnd and msgrcv

```
struct msg {
     long mtype;             /* message type */
     char mtext[MTEXTSIZE]; /* message text */
};
```

The structure you use for the message can be anything you like, as long as its first member is a `long` that's used for the type of message. This is so you can group messages that you send into types and then specify what types of messages you want to receive when you call `msgrcv`:

- If the `msgtype` argument to `msgrcv` is 0, you get the first message in the queue, regardless of type.
- If the argument is >0, you get the first message of that type.
- If the argument is <0, you get the first message of the lowest type that's less than or equal to the absolute value of `msgtype`. That is, if there are messages of types 5, 6, and 17 on the queue, and you specify –6, you will get the first message of type 5.

If you don't care about types, use 1 when you send (the type must be nonzero) and 0 for the `msgtype` argument to `msgrcv`. When types are used, it's most often to establish a priority system. You could also use a single queue for multiple serv-

ers and clients where each has its own type number, but that's an awkward way to organize things.

The `msgsize` argument to `msgsnd` is the size of just the message that's in the `mtext` member of the structure, not the whole structure, which includes the `mtype` member. Similarly, `msgrcv` returns the number of bytes of message in the `mtext` member, not the size of the whole structure. It's an error if the received message exceeds the `mtextsize` argument, unless you specify `MSG_NOERROR` in flags argument, in which case an oversize message is truncated, with no notice at all. This is usually a bad idea.

`msgsnd` ordinarily blocks if a limit on the number of messages in a queue or the total size of the messages on a queue would be exceeded. Or, you can specify `IPC_NOWAIT` (the System V IPC version of `O_NONBLOCK`) in the flags argument to instead make it return –1 with `errno` set to `EAGAIN`. (More on limits in the next section.)

`msgrcv` ordinarily blocks if a message of the requested type isn't present. Or, as with `msgsnd`, you can set the `IPC_NOWAIT` flag to make it nonblocking. Because message queues don't use file descriptors, you can't use `select` or `poll`, so try to avoid designs that require you to wait on more than one queue. You may be able to use a single queue instead with different message types to keep the contents straight. If you have to wait on multiple queues, consider using the techniques in Section 5.18.

7.5.2 System V Message-Queue Limits

There are a bunch of limits that serious applications are likely to bump up against, including:

- Limit on the total size of all messages in a queue. You can access this limit with the `msgctl` system call, through the `msg_qbytes` member of the `msqid_ds` structure.
- Limit on the number of messages in the queue.
- Limit on the size of a message.
- Limit on the number of queues.

Reaching the first two limits isn't necessarily a serious error. You can arrange for `msgsnd` to block or to return with an indication that a limit has been reached, as discussed in the previous section.

The other limits do cause an error, and there's nothing you can do about it other than reconfigure the kernel. You can't easily even determine what the limits are (`sysconf` won't get them), although it's fairly easy to run experiments to get an idea. For example, I hit a limit of 40 messages on Solaris, but that was the maximum number of messages for all queues. On FreeBSD, it was 20, and that seemed to be the limit per queue. There was no limit on Linux. For the maximum size of a message, I came up with 2048 bytes on Solaris and FreeBSD, and 8192 on Linux.[5]

Reconfiguring the kernel may be OK on your development system, but it's impractical if you're shipping your application for a customer to run on its own computers. Worse, the customer may have more than one application using System V IPC message queues, and the configuration instructions might conflict.

So here's some practical advice:

- Keep messages short—say, 1024 bytes or under. Use shared memory if you have to communicate more data. Shared memory also avoids two expensive copies (from one process to the kernel, and back to the other process).
- Don't push the other limits. Keep the number of queues small and don't assume that you can place any particular number of messages on a queue.
- During your application's installation, run some tests to ensure that the limits are adequate. Be prepared to advise the customer how to reconfigure the system if necessary.

7.5.3 System V Message-Queue Implementation of SMI

The message-queue system calls are actually very easy to use, as I'll demonstrate by implementing the SMI interface. You may want to review Section 7.3.1, where the interface was introduced, before proceeding.

The internal message queue is pretty simple compared to the one for FIFOs in Section 7.3.3:

```
typedef struct {
    SMIENTITY sq_entity;     /* entity */
    int sq_qid_server;       /* server identifier */
    int sq_qid_client;       /* client identifier (not used by server) */
```

5. The version of Darwin I used, 6.6, doesn't have System V messages.

```
    char sq_name[SERVER_NAME_MAX];  /* server name */
    struct client_id sq_client;     /* client ID */
    size_t sq_msgsize;              /* msg size */
    struct smi_msg *sq_msg;         /* msg buffer */
} SMIQ_MSG;
```

Note that that server doesn't have to retain information on each client to reduce the overhead when a message is to be returned because the identifier that will be passed with each message can be used directly in a call to `msgsnd`. In fact, the server doesn't even use the `sq_qid_client` member.

Opening an SMI message queue is also simple:

```
SMIQ *smi_open_msg(const char *name, SMIENTITY entity, size_t msgsize)
{
    SMIQ_MSG *p = NULL;
    char msgname[SERVER_NAME_MAX + 100];
    key_t key;

    ec_null( p = calloc(1, sizeof(SMIQ_MSG)) )
    p->sq_msgsize = msgsize + offsetof(struct smi_msg, smi_data);
    ec_null( p->sq_msg = calloc(1, p->sq_msgsize) )
    p->sq_qid_server = p->sq_qid_client = -1;
    p->sq_entity = entity;
    if (strlen(name) >= SERVER_NAME_MAX) {
        errno = ENAMETOOLONG;
        EC_FAIL
    }
    strcpy(p->sq_name, name);
    mkmsg_name_server(p, msgname, sizeof(msgname));
    (void)close(open(msgname, O_WRONLY | O_CREAT, 0));
    ec_neg1( key = ftok(msgname, 1) )
    if (p->sq_entity == SMI_SERVER) {
        if ((p->sq_qid_server = msgget(key, PERM_FILE)) != -1)
            (void)msgctl(p->sq_qid_server, IPC_RMID, NULL);
        ec_neg1( p->sq_qid_server = msgget(key, IPC_CREAT | PERM_FILE) )
    }
    else {
        ec_neg1( p->sq_qid_server = msgget(key, 0) )
        ec_neg1( p->sq_qid_client = msgget(IPC_PRIVATE, PERM_FILE) );
    }
    return (SMIQ *)p;

EC_CLEANUP_BGN
    if (p != NULL)
        (void)smi_close_msg((SMIQ *)p);
    return NULL;
EC_CLEANUP_END
}
```

```
static void mkmsg_name_server(const SMIQ_MSG *p, char *msgname,
  size_t msgname_max)
{
    snprintf(msgname, msgname_max, "/tmp/smimsg-%s", p->sq_name);
}
```

We use a file in the /tmp directory for each unique server name; the function `mkmsg_name_server` constructs the pathname. If that file doesn't already exist, the line just before the call to `ftok` creates it. We want the server to start with a fresh queue so it removes the message queue if it already exists before creating it. (In your own application you probably can't be so cavalier—you'll be concerned with what to do about running clients and whether you really want a fresh queue or whether you want to do more with existing messages than simply tossing them out.) The client gets access to the server queue, assuming the server started already, and creates a private queue for itself.

Here's `smi_close_msg`:

```
bool smi_close_msg(SMIQ *sqp)
{
    SMIQ_MSG *p = (SMIQ_MSG *)sqp;

    if (p->sq_entity == SMI_SERVER) {
        char msgname[FILENAME_MAX];

        (void)msgctl(p->sq_qid_server, IPC_RMID, NULL);
        mkmsg_name_server(p, msgname, sizeof(msgname));
        (void)unlink(msgname);
    }
    else
        (void)msgctl(p->sq_qid_client, IPC_RMID, NULL);
    free(p->sq_msg);
    free(p);
    return true;
}
```

Next come `smi_send_getaddr_msg` and `smi_send_release_msg`:

```
bool smi_send_getaddr_msg(SMIQ *sqp, struct client_id *client,
  void **addr)
{
    SMIQ_MSG *p = (SMIQ_MSG *)sqp;

    if (p->sq_entity == SMI_SERVER)
        p->sq_client = *client;
    *addr = p->sq_msg;
    return true;
}
```

```
bool smi_send_release_msg(SMIQ *sqp)
{
    SMIQ_MSG *p = (SMIQ_MSG *)sqp;
    int qid_receiver;

    p->sq_msg->smi_mtype = 1;
    if (p->sq_entity == SMI_SERVER)
        qid_receiver = p->sq_client.c_id1;
    else {
        qid_receiver = p->sq_qid_server;
        p->sq_msg->smi_client.c_id1 = p->sq_qid_client;
    }
    ec_neg1( msgsnd(qid_receiver, p->sq_msg,
      p->sq_msgsize - sizeof(p->sq_msg->smi_mtype), 0) )
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

In `smi_send_release_msg`, if the client is sending, it already has the server's identifier, and it puts the identifier to its private queue in the message. If the server is sending, it takes the identifier to send to from the `SMIQ_MSG` structure where it was saved by the previous call to `smi_send_getaddr_msg`. You can also see now why we put a message type at the start of the `smi_msg` structure: It allows us to use that structure directly in calls to `msgsnd` and `msgrcv`. Note also that the size passed to `msgsnd` is just the data part, not the whole structure.

Finally, here are `smi_receive_getaddr_msg` and `smi_receive_release_msg`, which are even simpler than the sending pair:

```
bool smi_receive_getaddr_msg(SMIQ *sqp, void **addr)
{
    SMIQ_MSG *p = (SMIQ_MSG *)sqp;
    int qid_receiver;
    ssize_t nrcv;

    if (p->sq_entity == SMI_SERVER)
        qid_receiver = p->sq_qid_server;
    else
        qid_receiver = p->sq_qid_client;
    ec_neg1( nrcv = msgrcv(qid_receiver, p->sq_msg,
      p->sq_msgsize - sizeof(p->sq_msg->smi_mtype), 0, 0) )
    *addr = p->sq_msg;
    return true;
```

```
EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}

bool smi_receive_release_msg(SMIQ *sqp)
{
    return true;
}
```

Server and client both have the identifiers of their queues at hand in the `SMIQ_MSG` structure.

7.5.4 System V Message Queues Critiqued

This SMI implementation illustrates the best parts of System V message queues: A process can send a message to a queue given only the identifier, without the overhead of additional system calls to get access to it, and there's no worry about keeping messages atomic. We don't see it here, but even multiple receivers from the same queue are no problem (with FIFOs they're not safe).

One negative is that the various limits are inadequately specified (minimums would be nice) and awkward to query at run time. The biggest negative, however, true of all IPC mechanisms other than sockets, is that messages can be sent only within a single machine. For today's applications, that's often too restrictive.

The first edition of this book included this advice:

> [System V messages queues are] complex, incompletely documented, and nonportable. *Avoid them if at all possible!*

Reconsidering from a 2004 perspective, they no longer seem very complex, now that we have sockets, threads, and lots of other features that are even more complex. They are still incompletely documented, but, again, that doesn't make them so unusual. They are definitely portable—they're in the SUS and seem to be present on all the principal systems (Darwin soon, we hope). Thus, there's no reason to avoid them, as long as you know their limitations.

7.6 POSIX IPC

This section explains some general things about POSIX IPC that apply to messages, semaphores, and shared memory, which are covered in Sections 7.7, 7.10, and 7.14.

7.6.1 POSIX IPC History

POSIX IPC was introduced with POSIX 1003.1b-1993 (POSIX1993 for short) as part of the real-time extensions. The POSIX message-queue system calls described here constitute the message-passing option of that standard and are still optional as of POSIX2001. By contrast, System V IPC was never part of POSIX but is now mandatory on Open Group UNIX-certified systems.

Historically System V IPC was nonstandard yet almost universally implemented, whereas POSIX IPC has been a standard (but optional) for 10 years yet is still not available on many UNIX systems, notably Linux, FreeBSD, and Darwin. In theory POSIX IPC was more portable than System V IPC; in practice it was the reverse.

What's more, even when it is implemented, POSIX IPC isn't necessarily efficient enough for critical real-time applications. The standard requires no particular level of performance, and some OS vendors haven't committed the resources to any objectives beyond being conformant.

Therefore, as we'll see, the POSIX IPC system calls are in many ways more functional and more cleanly designed than the ones of System V IPC, but you may not be able to use them in your application.

7.6.2 POSIX IPC Names

Recall from Section 7.4.2 that System V IPC uses keys to specify its objects, with the added-on convenience of a scheme to generate keys from pathnames with the `ftok` system call. POSIX IPC uses a simpler scheme, with names (strings) instead of keys.

Names have to follow the same rules as for pathnames, and the standard requires that if the name begins with a slash then all references to that name refer to the same object. There's no such requirement if the name does not begin with a slash, but there are problems with this approach:

- There's no requirement that something like a file with the specified name actually appear in the file system. POSIX IPC objects, like System V IPC objects, are *not* files. This is for efficiency—file systems represent a lot of heavy machinery that a fast implementation of POSIX IPC is free to bypass,

using, for example, an in-memory hash table to find objects instead of using the file system's directory tree.
- On most UNIX systems, ordinary users can't write in the root directory, which creates a problem if the implementation actually creates a file corresponding to the name. A potential solution to this problem is for an administrator to first set up at least one subdirectory for general use (e.g., /ipc), but see the next point.
- The interpretation of slashes other than the first is implementation dependent. Some implementations might treat them as ordinary characters, some might treat them as directory delimiters, and some (such as Solaris) might prohibit them altogether. So, for portability, you should have no slashes other than the leading one. This rules out the solution in the previous point.

So, what to do? Probably write a small initialization program that just creates all the objects your application will need (using names with a leading slash only) and set it up so its effective user ID is the superuser or whatever user can write into the root directory. For development, either use the initialization program or run on a system that doesn't create actual files. The examples in this book were developed on Solaris, which doesn't create files, so the pathnames that appear to be in the root directory aren't really there and everything works just fine.

An alternative approach is to `#ifdef` your code for various systems, using whatever naming scheme that system supports. But generally this is a bad idea because whatever small set of systems you decide to include will undoubtedly be insufficient someday, and then somebody will have to modify the code to include the new system. Then the modified code has to be worked back into the base code, with all the support and maintenance headaches that invariably result.

Yet another idea is to put the pattern for the POSIX IPC names in a configuration file that's read at run-time. Then choosing the appropriate scheme becomes an installation option, rather than a compile-time option. This might be OK, but it still adds to the many ways that a complex application can be installed incorrectly, which is something you don't need. Maybe the System V IPC scheme using keys and `ftok` wasn't so silly after all.

7.6.3 POSIX IPC Feature-Test Macros

As explained at length in Section 1.5.4, the SUS and earlier POSIX standards provided macros that you can check at compile time to test whether optional features

are present, absent, or whether you have to code a run-time check. Unfortunately, the systems that implement the macros exactly as they're supposed to usually have the optional features (which means you could have skipped the test), and the systems that don't have the optional features don't implement the macros correctly either. Therefore, in practice, testing the macros doesn't actually lead to portability.

But, if you want to give it a try, read carefully what I said in Chapter 1 about how they work in general and then consult [SUS2002] or the relevant POSIX standard to see what specific macros to test.

I generally won't use macro tests in my examples. They compile and run on the systems that support the optional features, and they don't compile on the others.

7.6.4 POSIX IPC Utilities

For System V, we had the handy `ipcs` and `ipcrm` utilities (Section 7.4.4). There aren't any for POSIX IPC, so this section is short. Too bad if you want to check what message queues, semaphores, or shared-memory segments might be left around by errant applications.

7.7 POSIX Message Queues

This section explains POSIX message queues and shows how they can be used to implement the SMI.

7.7.1 POSIX Message-Queue System Calls

It turns out that the name issue is the only real sticky part of using POSIX message queues. The rest is easy enough. First, `mq_open` opens a message queue.

`mq_open` behaves much like `open`, although the object it creates can't portably be treated as a file, and it returns a message-queue descriptor, not a file descriptor. Regrettably, you can't use the message-queue descriptor with `select` or `poll`, which is the same disadvantage that we saw with System V message queues. However, you can be notified with a signal when a message is available; see `mq_notify`, below.

mq_open—open message-queue

```
#include <mqueue.h>

mqd_t mq_open(
     const char *name,      /* POSIX IPC name */
     int flags              /* flags (excluding O_CREAT) */
);
/* Returns message-queue descriptor or -1 on error (sets errno) */

mqd_t mq_open(
     const char *name,      /* POSIX IPC name */
     int flags,             /* flags (including O_CREAT) */
     mode_t perms,          /* permissions */
     struct mq_attr *attr   /* attributes (or NULL) */
);
/* Returns message-queue descriptor or -1 on error (sets errno) */
```

struct mq_attr—structure for mq_open, mq_getattr, and mq_setattr

```
struct mq_attr {
     long mq_flags;         /* flags */
     long mq_maxmsg;        /* max number of messages */
     long mq_msgsize;       /* max message size */
     long mq_curmsgs;       /* number of messages currently queued */
};
```

The macros for the `flags` argument are the same as for `open` (e.g., `O_CREAT`), not special macros as used with `msgget` (e.g., `IPC_CREAT`):

- You must specify one of `O_RDONLY`, `O_WRONLY`, or `O_RDWR`, depending on whether you want to receive messages only, send messages only, or both.
- You specify `O_CREAT` if you want to create the queue if it doesn't already exist. In this case you supply four arguments, as shown in the second form in the synopsis box.
- You specify `O_EXCL` if you want the function to fail when the queue already exists.
- You specify `O_NONBLOCK` if you want the message-queue descriptor to be nonblocking, which means that `mq_send` and `mq_receive` return –1 with `errno` set to `EAGAIN` if the queue is full or empty, respectively. The default behavior is blocking.

When `O_CREAT` is set and the queue is actually created, the interpretation of the `perms` argument is similar to the permissions on files and System V message queues, with the usual `S_` flags (Section 2.3). Read and write permission mean the ability to receive and send messages, respectively, and execute doesn't mean anything. As with files, the queue has a user- and group-ID, and they're set from the effective IDs of the process that created it. Subsequently, the interaction of the

owner, group, and other bits with the effective user- and group-IDs is that same as with files.

Also, if `O_CREAT` is specified and a queue is created and the `attr` argument is non-`NULL`, two attributes, `mq_maxmsg` and `mq_msgsize` are set from the supplied `mq_attr` structure. These are the maximum number of messages that the queue can hold and the maximum size for an individual message. The standard doesn't specify a limit for either attribute, but if there's insufficient space `mq_open` will return –1 with `errno` set to `ENOSPC`. Most implementations will implement the queue with a linked list, so `mq_open` won't run out of space regardless of the numbers because a new queue is empty. If the `attr` argument is `NULL`, the two attributes are set to implementation-defined values.

If your eyes glazed over reading the previous few paragraphs and you go ahead assuming `mq_open` is just like `open`, you'll probably be just fine as long as you don't assume it returns a file descriptor.

You close an open message queue with `mq_close`:

mq_close—close message queue

```
#include <mqueue.h>

int mq_close(
    mqd_t mqd              /* message-queue descriptor */
);
/* Returns zero on success or -1 on error (sets errno) */
```

As with System V message queues, POSIX message queues persist until they're removed or the kernel is rebooted. You remove a POSIX message queue with a call that's like `unlink`:

mq_unlink—remove message queue

```
#include <mqueue.h>

int mq_unlink(
    const char *name       /* POSIX IPC name */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

Like `unlink`, `mq_unlink` makes the name disappear right away, but actual destruction of the queue is postponed until all open message-queue descriptors are closed. (Recall that with System V message queues, removing a queue with `msgctl` was immediate and potentially disruptive.) So, if you want to, once all

processes that need the queue have opened it, you can `mq_unlink` it, just as you sometimes do with temporary files.

OK, ready to send a message? You call `mq_send`, as you might have guessed:

mq_send—send message

```
#include <mqueue.h>

int mq_send(
     mqd_t mqd,             /* message-queue descriptor */
     const char *msg,       /* message */
     size_t msgsize,        /* size of message */
     unsigned priority      /* priority */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

`mq_send` places the message of size `msgsize` onto the queue, positioning it in front of all messages with a lower `priority`, but after messages with the same `priority` (in other words, just as you would expect). Priorities go from zero to 31, at least; you can retrieve the actual maximum value with `sysconf` (Section 1.5.5).

As mentioned above, `mq_send` blocks if the queue is full, unless the `O_NONBLOCK` flags is set.

The call to receive a message is, of course, `mq_receive`:

mq_receive—receive message

```
#include <mqueue.h>

ssize_t mq_receive(
     mqd_t mqd,             /* message-queue descriptor */
     char *msg,             /* message buffer */
     size_t msgsize,        /* size of message buffer */
     unsigned *priorityp    /* returned priority or NULL */
);
/* Returns size of message or -1 on error (sets errno) */
```

`mq_receive` gets the oldest of the highest priority messages, which, because of the way `mq_send` is defined, is the same as saying the first message on the queue.

The `msgsize` argument is a little tricky: It's the size of the buffer pointed to by the `msg` argument, as you would expect, but it must be at least as great as the `mq_msgsize` attribute (see `mq_open`, above), or `mq_receive` will fail, *even if the front-most message is small enough to fit*. This is actually an excellent design choice: Catch the problem of a too-small buffer right away, rather than requiring

just the right test data to expose the bug. The actual size of the received message is the return value.

If the `priorityp` argument is non-`NULL`, it gets the priority of the received message. The received message is removed from the queue; there's no way to just peek at it. If the queue is empty, `mq_receive` blocks, unless `O_NONBLOCK` is set.

There are variants of `mq_send` and `mq_receive` that allow a blocked call to timeout:

mq_timedsend—send message with timeout

```
#include <mqueue.h>
#include <time.h>

int mq_timedsend(
     mqd_t mqd,                     /* message-queue descriptor */
     const char *msg,               /* message */
     size_t msgsize,                /* size of message */
     unsigned priority              /* priority */
     const struct timespec *tmout   /* timeout */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

mq_timedreceive—receive message with timeout

```
#include <mqueue.h>
#include <time.h>

ssize_t mq_timedreceive(
     mqd_t mqd,                     /* message-queue descriptor */
     char *msg,                     /* message buffer */
     size_t msgsize,                /* size of message buffer */
     unsigned *priorityp            /* returned priority or NULL */
     const struct timespec *tmout   /* timeout */
);
/* Returns size of message or -1 on error (sets errno) */
```

The timeout applies only if the call would block and `O_NONBLOCK` is clear. When that amount of time elapses, the function returns with –1 and `errno` set to `ETIMEDOUT`. These two functions are part of the Timeouts option (`_POSIX_TIMEOUTS`) and are new with SUS3.

As I said, it's awkward to handle more than one message queue or a message queue in combination with something else that can block, such as terminal or a network connection since you can't test a message-queue descriptor with `select` or `poll`. However, you can arrange to be notified with a signal when a message arrives, and then you can call `mq_receive` without worrying about blocking. (See Section 5.18 for another solution.) You set this up with `mq_notify`.

mq_notify—register or unregister for message notification

```
#include <mqueue.h>

int mq_notify(
    mqd_t mqd,                      /* message-queue descriptor */
    const struct sigevent *ep       /* notification */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

When a process or thread is registered, it receives a signal when a message arrives on an empty queue designated by `mqd`, unless one or more processes or threads are already blocking in an `mq_receive`, in which case one of the `mq_receive`s returns instead.

Only one process or thread may be registered; attempting to register a second process or thread results in an error. A process or thread is unregistered when a signal is sent or when `mq_notify` is called with a `NULL` second argument.

There's no notification of a message arrival on a nonempty queue, so `mq_notify` all by itself doesn't ensure that a process will get a signal every time a message arrives. For example, it may get a signal when a message arrives on an empty queue, but then a second message may arrive before the first message is received, which would not cause notification, since the queue was not empty. So, an application needs to do something like this:

- After calling `mq_notify`, repeatedly call `mq_receive` with `O_NONBLOCK` set until the queue is empty.
- When a signal arrives, immediately call `mq_notify` again and then do the previous step.

What signal is sent and other properties of the notification are determined by the contents of the `sigevent` structure passed to `mq_notify`, as explained in Section 9.5.6.

Finally, you can get and set message-queue attributes:

mq_getattr—get message-queue attributes

```
#include <mqueue.h>

int mq_getattr(
    mqd_t mqd,              /* message-queue descriptor */
    struct mq_attr *attr    /* attributes */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

mq_setattr—set message queue attributes

```
#include <mqueue.h>

int mq_setattr(
    mqd_t mqd,                      /* message-queue descriptor */
    const struct mq_attr *attr,     /* new attributes */
    struct mq_attr *oldattr         /* old attributes if not NULL */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

`mq_getattr` fills the structure (shown along with `mq_open`) according to the current state of the queue. The most useful attribute is probably member `mq_curmsgs`, which contains the current number of messages on the queue.

`mq_setattr` sets or clears the `O_NONBLOCK` flag in the `mq_flags` member; by the standard, it has no effect on anything else. (Implementations might have other nonportable flags that can be set.) Thus, for the `O_NONBLOCK` flag, `mq_setattr` is used on message-queue descriptors as `fcntl` is used on file descriptors. If `oldattr` is non-`NULL`, it's used to get the old attributes returned.

7.7.2 POSIX Message-Queue Implementation of SMI

Now for our third implementation of the SMI functions. First, here is the internal message queue, which is almost the same as the one for FIFOs, because it has similar issues to deal with:

```
#define MAX_CLIENTS 20

typedef struct {
    SMIENTITY sq_entity;            /* entity */
    mqd_t sq_mqd_server;            /* server message-queue descriptor */
    char sq_name[SERVER_NAME_MAX];  /* server name */
    struct client_info {            /* Client uses only sq_clients[0] */
        mqd_t cl_mqd;               /* client message-queue descriptor */
        pid_t cl_pid;               /* client process ID */
    } sq_clients[MAX_CLIENTS];
    struct client_id sq_client;     /* client ID */
    size_t sq_msgsize;              /* msg size */
    struct smi_msg *sq_msg;         /* msg buffer */
} SMIQ_MQ;
```

As with FIFOs, to avoid the overhead of opening a client's message queue for each message, the server is going to keep track of each client it knows about and just look up the stored message-queue descriptor for an already-open queue. In many real servers, this isn't really such a burden because it would almost always

want to keep track of its clients anyway. The client will only use the first element of the array to store its own message-queue descriptor, but we don't care about wasted space in this example.

There are two functions to translate SMI server names and client process IDs to POSIX IPC names:

```
static void make_mq_name_server(const SMIQ_MQ *p, char *mqname,
  size_t mqname_max)
{
    snprintf(mqname, mqname_max, "/smimq-s%s", p->sq_name);
}

static void make_mq_name_client(pid_t pid, char *mqname,
  size_t mqname_max)
{
    snprintf(mqname, mqname_max, "/smimq-c%d", pid);
}
```

These names look like they're in the root directory, but on Solaris they're not, as explained in Section 7.6.2.

Next comes a function used by the server to get a client's message-queue descriptor from its process ID, which is what a client sends in a message to the server:

```
static mqd_t get_client_mqd(SMIQ_MQ *p, pid_t pid)
{
    int i, avail = -1;
    char mqname[SERVER_NAME_MAX + 100];

    for (i = 0; i < MAX_CLIENTS; i++) {
        if (p->sq_clients[i].cl_pid == pid)
            return p->sq_clients[i].cl_mqd;
        if (avail == -1 && p->sq_clients[i].cl_pid == 0)
            avail = i;
    }
    errno = ECONNREFUSED;
    ec_neg1( avail )
    p->sq_clients[avail].cl_pid = pid;
    make_mq_name_client(pid, mqname, sizeof(mqname));
    ec_neg1( p->sq_clients[avail].cl_mqd = mq_open(mqname, O_WRONLY) )
    return p->sq_clients[avail].cl_mqd;

EC_CLEANUP_BGN
    return (mqd_t)-1;
EC_CLEANUP_END
}
```

The function first looks in the array to see if it's already seen the client, in which case it already has the message-queue descriptor. If not, it opens the queue for writing and saves the descriptor for next time.

Now we're ready for the first SMI function, `smi_open_mq`:

```
static pid_t my_pid;

SMIQ *smi_open_mq(const char *name, SMIENTITY entity, size_t msgsize)
{
    SMIQ_MQ *p = NULL;
    char mqname[SERVER_NAME_MAX + 100];
    struct mq_attr attr = {0};

    my_pid = getpid();
    ec_null( p = calloc(1, sizeof(SMIQ_MQ)) )
    p->sq_msgsize = msgsize + offsetof(struct smi_msg, smi_data);
    ec_null( p->sq_msg = calloc(1, p->sq_msgsize) )
    p->sq_entity = entity;
    p->sq_mqd_server = p->sq_clients[0].cl_mqd = (mqd_t)-1;
    if (strlen(name) >= SERVER_NAME_MAX) {
        errno = ENAMETOOLONG;
        EC_FAIL
    }
    strcpy(p->sq_name, name);
    make_mq_name_server(p, mqname, sizeof(mqname));
    attr.mq_maxmsg = 100;
    attr.mq_msgsize = p->sq_msgsize;
    if (p->sq_entity == SMI_SERVER) {
        (void)mq_unlink(mqname);
        ec_cmp( errno, ENOSYS )
        ec_neg1( p->sq_mqd_server = mq_open(mqname, O_RDONLY | O_CREAT,
          PERM_FILE, &attr) )
    }
    else {
        ec_neg1( p->sq_mqd_server = mq_open(mqname, O_WRONLY) )
        make_mq_name_client(my_pid, mqname, sizeof(mqname));
        ec_neg1( p->sq_clients[0].cl_mqd = mq_open(mqname,
          O_RDONLY | O_CREAT, PERM_FILE, &attr) )
    }
    return (SMIQ *)p;

EC_CLEANUP_BGN
    if (p != NULL)
        (void)smi_close_mq((SMIQ *)p);
    return NULL;
EC_CLEANUP_END
}
```

After initializing the internal SMI queue, we initialize an `mq_attr` structure with limits of 100 messages on the queue and a message size that was passed as the third argument. Then, if we're the server, we discard any existing server queue and create a fresh one. We don't bother doing this for a client, although we probably should, because the client's process ID is embedded in the queue name, and it's unlikely to be reused.

The close function looks like this, doing what you'd expect:

```
bool smi_close_mq(SMIQ *sqp)
{
    SMIQ_MQ *p = (SMIQ_MQ *)sqp;
    char msgname[SERVER_NAME_MAX + 100];

    if (p->sq_entity == SMI_SERVER) {
        make_mq_name_server(p, msgname, sizeof(msgname));
        (void)mq_close(p->sq_mqd_server);
        (void)mq_unlink(msgname);
    }
    else {
        make_mq_name_client(my_pid, msgname, sizeof(msgname));
        (void)mq_close(p->sq_mqd_server);
        (void)mq_unlink(msgname);
    }
    free(p->sq_msg);
    free(p);
    return true;
}
```

Next comes the `smi_send_getaddr_mq` and `smi_send_release_mq`:

```
bool smi_send_getaddr_mq(SMIQ *sqp, struct client_id *client,
  void **addr)
{
    SMIQ_MQ *p = (SMIQ_MQ *)sqp;

    if (p->sq_entity == SMI_SERVER)
        p->sq_client = *client;
    *addr = p->sq_msg;
    return true;
}

bool smi_send_release_mq(SMIQ *sqp)
{
    SMIQ_MQ *p = (SMIQ_MQ *)sqp;
    mqd_t mqd_receiver;
```

```
    if (p->sq_entity == SMI_SERVER)
        ec_neg1( mqd_receiver = get_client_mqd(p, p->sq_client.c_id1) )
    else {
        mqd_receiver = p->sq_mqd_server;
        p->sq_msg->smi_client.c_id1 = my_pid;
    }
    ec_neg1( mq_send(mqd_receiver, (const char *)p->sq_msg,
      p->sq_msgsize, 0) )
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

As before with FIFOs and System V message queues, `smi_send_getaddr_mq` just saves the client ID (if it's called from the server) and returns the buffer address. The real work is in `smi_send_release_mq`. The server calls the look-up function `get_client_mqd` to get the message-queue descriptor. Its argument was saved in the `SMIQ_MQ` structure by `smi_send_getaddr_mq`.

The client already has the server's descriptor, and it puts its process ID into the message. (The message-queue descriptor can't be passed between processes, like a System V message-queue identifier can.)

The receive pair are simpler since a receiver already has the descriptor to its message queue:

```
bool smi_receive_getaddr_mq(SMIQ *sqp, void **addr)
{
    SMIQ_MQ *p = (SMIQ_MQ *)sqp;
    mqd_t mqd_receiver;
    ssize_t nrcv;

    if (p->sq_entity == SMI_SERVER)
        mqd_receiver = p->sq_mqd_server;
    else
        mqd_receiver = p->sq_clients[0].cl_mqd;
    ec_neg1( nrcv = mq_receive(mqd_receiver, (char *)p->sq_msg,
      p->sq_msgsize, NULL) )
    *addr = p->sq_msg;
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

```
bool smi_receive_release_mq(SMIQ *sqp)
{
    return true;
}
```

7.7.3 POSIX Message Queues Critiqued

POSIX message-queue system calls have a somewhat cleaner interface than the System V calls and are more integrated into the rest of UNIX, but they still use their own descriptor rather than a file descriptor so `select` and `poll` can't be used. To compensate, there is a notification feature, but it's very tricky to use, as we'll see in Chapter 9, because it's based on signals, which are always tricky to use.

The main problem with POSIX message queues (all POSIX IPC, actually) is that it's not widely available, but, of course, this isn't a criticism of the design.

Table 7.1 compares POSIX and System V message queues in more detail:

Table 7.1 POSIX vs. System V Message Queues

Criterion	POSIX	System V
Standardized?	yes	yes
Mandatory in UNIX-certified systems?	no	yes
Available in all major UNIX systems?	no	yes
Message limits?	very few; settable	many; only one settable; hard to manage
Thread safe (according to SUS3)?	yes	yes
Message priorities?	yes	yes
Receive other than highest-priority message?	no	yes
Notification?	yes	no

Table 7.1 POSIX vs. System V Message Queues (cont.)

Criterion	POSIX	System V
Uses file descriptors?	no	no
Efficiency?	depends on implementation	depends on implementation
Two user-kernel copies of each message?*	yes	yes
Portability issues with queue names?	yes	no
* This means that data is copied from user space to kernel space, or the other way around.		

With this mixed bag of pros and cons, why did the POSIX real-time group invent a new message system when System V messages had been around for over 10 years? And, since they did, why aren't the results better?

Here are my answers:

- While the POSIX group certainly had a lot of quibbles with the existing System V approach, the main reason they needed a new standard was that many major systems already had a System V implementation that was unsuitable for real-time applications, and it would be hard to have two simultaneous implementations behind the same interface. They needed a way to give implementors a fresh start.
- But, there was no practical way within the POSIX process to require any particular level of performance, so on many, if not most, systems, the efficiency of the two (when there even are two) is about the same.
- The nonportability of POSIX IPC names is caused by underspecification, which was required in order to allow a broad set of implementations, including very fast in-memory queues with no file-system lookup of any kind.
- The POSIX group is not responsible for the lack of implementations. What's probably most responsible is that by the mid-1990s many, if not most, application developers were focusing on network applications, for which they used sockets (Chapter 8), and there wasn't much interest in an alternative way to handle non-networked messages. So well-funded outfits like Sun implemented the POSIX option packages, and shoestring outfits like the BSD and Linux communities found more pressing things to do.

7.8 About Semaphores

This section explains general properties of semaphores and shows how they can be implemented with files and messages. The System V and POSIX semaphore system calls are covered in Sections 7.9 and 7.10.

7.8.1 Basic Semaphore Usage

We already encountered mutexes in Section 5.17.3. More generally, a semaphore is a counter that prevents two or more processes or threads from accessing a shared resource simultaneously. It's only advisory, however: if a process or thread doesn't check a semaphore before accessing a shared resource, chaos might result.

In UNIX, the term "semaphore" usually applies to an object manipulated by the semaphore system calls to synchronize processes. The term "mutex" usually means a much lighter-weight object that's used to synchronize threads, which is what we discussed in Section 5.17.3. I'll talk only about semaphores in this section.

A *binary* semaphore has only two states: locked and unlocked. A *general* semaphore has an infinite (or, at least, very large) number of states. It's a counter that decreases by one when it is acquired ("locked") and increases by one when it is released ("unlocked"). If it's zero, a process trying to acquire it must wait for another process to increase its value—it can't ever get negative. (If a semaphore can take on only two values, 0 and 1, it is a binary semaphore.) General semaphores are usually accessed with two operations that we can refer to abstractly as *semwait* and *sempost* (sometimes called P and V[6]). We can try to write these in C:

```
void semwait(int *sem)
{
    while (*sem <= 0)
        ; /* do nothing */
     (*sem)--;
}

void sempost(int *sem)
{
     (*sem)++;
}
```

6. P and V are abbreviations used by E. W. Dijkstra for the Dutch phrase *proberen te verlagen* ("try to decrease") and the word *verhogen* ("increase"). Some books say that P stands for *prolagen*, but that isn't a word, not even in Dutch. It's a contraction of the full phrase.

The semaphore must be initialized by calling *sempost;* otherwise it starts out with nothing to acquire. For example, if the semaphore counts the number of free buffers, and there are initially five buffers, we would start out by calling *sempost* five times. Alternatively, we can just set the semaphore variable to five.

Having gone to the trouble of programming *semwait* and *sempost,* I now have to say that they won't work, for three reasons:

- The semaphore variable pointed to by `sem` isn't normally shared among processes, which have distinct data segments. (Although it could be in shared memory.)
- The functions do not execute atomically—the kernel can interrupt a process at any time. The following scenario could occur: Process 1 completes the `while` loop in *semwait* and is interrupted before it can decrement the semaphore; process 2 enters *semwait,* finds the semaphore equal to 1, completes its `while` loop, and decrements the semaphore to 0; process 1 resumes and decrements the semaphore to –1 (an illegal value).
- *semwait* does what's called a *busy-wait* if `sem` is zero. This is a dumb way to use a CPU.

Therefore, *semwait* and *sempost* can't just be programmed in user space. Semaphores have to be supplied by the kernel, which can share data between processes, can execute atomic operations, and can give the CPU to a ready process when a process blocks.

7.8.2 Implementing Semaphores with Files and Messages

Back in Section 2.4.3 I showed how to use `open` to make a crude binary semaphore. That method is OK for a process that accesses a shared resource only a few times, such as a mail program that's writing into a mailbox file, but the overhead is far too great for heavy-duty use. We want a semaphore that takes less time to check and set.

A message queue can also be used as a semaphore: A send operation adds a message to the queue and is equivalent to *sempost;* a receive removes a message from the queue and is equivalent to *semwait*. A receive blocks when the queue is empty, which is equivalent to the semaphore being zero.

However, any UNIX system that supports messages (System V or POSIX varieties) also supports semaphores in their own right, and much more efficiently.

7.9 System V Semaphores

Neither System V nor POSIX semaphores use anything as simple as *semwait* and *sempost*. Their system calls are more complicated, although POSIX semaphores are not much more complicated. I'll critique the two together in Section 7.10.3.

I'll start with System V. These system calls are too complex for me to describe completely. I'll make sure, however, that I explain enough to allow us to implement *semwait* and *sempost;* for the rest, see your system's documentation or [SUS2002].

The information in Section 7.4 applies to these system calls so we don't need to say anything specific about keys, identifiers, ownership, etc.

What's interesting about the System V semaphore facility is that the system calls operate not just on a single semaphore, but on a whole array at once. With one atomic operation, you can do a *semwait* on several and a *sempost* on several others. It's questionable whether you will ever want to do this in practice, but it's there if you need it. In most of our examples we'll operate on only one semaphore at a time, although we will later on use a set of two, just for convenience.

One comment before we go on: System V semaphores are too complicated! In any program using semaphores it's essential to be able to demonstrate that access to shared resources is exclusive, that deadlock does not occur, and that starvation (never getting access) does not occur. Testing alone can't usually suffice because things are so timing-dependent; analysis must be used. This is difficult enough with plain *semwait* and *sempost*. With the System V system calls, used in their full glory, analysis is probably impossible.

7.9.1 System V Semaphore System Calls

As with System V message queues, you start with the *X*`get` call:

semget—get semaphore-set identifier

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

int semget(
    key_t key,                  /* key */
    int nsems,                  /* size of set */
    int flags                   /* flags */
);
/* Returns identifier or -1 on error (sets errno) */
```

As you would expect, `semget` translates a key to an ID representing a set of semaphores. If the `IPC_CREAT` bit of `flags` is on, the set is created if it doesn't already exist. There are `nsems` semaphores in the set, numbered starting with zero.

The semaphores aren't ready to use right away—they have to be initialized with a call to `semctl`. Why `semget` doesn't initialize them to zero is a mystery, but the SUS doesn't require it to do so,[7] and many implementations don't. Here's `semctl`:

semctl—control semaphore set

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

int semctl(
     int semid,                 /* identifier */
     int semnum,                /* semaphore number */
     int cmd,                   /* command */
     union semun arg            /* argument for command */
);
/* Returns value or 0 on success; -1 on error (sets errno) */
```

union semun—union for semctl

```
union semun {
    int val;                    /* integer */
    struct semid_ds *buf;       /* pointer to structure */
    unsigned short *array;      /* array */
};
```

struct semid_ds—structure for semctl

```
struct semid_ds {
     struct ipc_perm sem_perm; /* permission structure */
     unsigned short sem_nsems; /* size of set */
     time_t sem_otime;         /* time of last semop */
     time_t sem_ctime;         /* time of last semctl */
};
```

Oddly, `union semun` isn't defined in the header sem.h—you have to define it yourself.

7. Actually, the SUS says, "The data structure associated with each semaphore in the set shall not be initialized." The use of the phrase "shall not" means that initializing it is not even an implementation option. Is this intentional or just sloppy writing?

There are seven commands unique to semaphores, in addition to the common System V IPC commands IPC_RMID, IPC_STAT, and IPC_SET:

GETNCNT Get the number of processes blocked waiting for semaphore semnum to increase.

GETZCNT Get the number of processes blocked waiting for semaphore semnum to become zero.

GETPID Get the ID of the process to last perform a semop.

GETVAL Get the value of semaphore semnum.

SETVAL Set the value of semaphore semnum. Uses arg.val.

GETALL Get the values of all the semaphores in the set. Uses arg.array.

SETALL Set the values of all the semaphores in the set. Uses arg.array.

Amazingly, if you have a set of 100 semaphores and you want to set them all to zero with a single semctl, you have to build an array of 100 zeros to use as the fourth argument! Doesn't bother us, because we always initialize them one at a time, with SETVAL.

The IPC_RMID command acts just like it does with msgctl (Section 7.5.1).

IPC_STAT fills the semid_ds structure passed via the fourth argument, using the buf member of the union. IPC_SET can be used only to set the sem_perm.uid, sem_perm.gid, and sem_perm.mode members.

Since, as I said, semget doesn't initialize the semaphores, you use semctl to do it, as in this sequence:

```
ec_neg1(semid = semget(key, 1, PERM_FILE | IPC_CREAT) )
arg.val = 0;
ec_neg1( semctl(semid, 0, SETVAL, arg) )
```

Unfortunately, as it takes two system calls to get the semaphore created and initialized, a second process or thread that does a semget on the same semaphore may start processing with an uninitialized semaphore before the semctl is executed. The solution is to take advantage of the fact that, while the value of a new semaphore isn't initialized, its "last semop time," which is stored in the

sem_otime member of the semid_ds structure, is initialized to zero by semget. This suggests[8] the following scheme:

- The process or thread that creates the semaphore also calls semctl to initialize it and then does a semop on it so sem_otime will take on a nonzero value.
- Any other process or thread that gets the semaphore ID waits for the time to become nonzero.

This scheme works only for brand-new semaphores. If you restart an application with a semaphore left around from a previous run, that semaphore will already have a value and a nonzero value for sem_otime, which will probably mess things up. So, make sure you clean up between runs of the application by removing the semaphore, either with a call to semctl or with the ipcrm command (Section 7.4.4).

Unfortunately, as of this writing, waiting for sem_otime doesn't work on FreeBSD or Darwin because the time is never updated.[9] Therefore, System V semaphores can't be used reliably at all unless you're sure because of the nature of the application that the initialization will occur before the second attempt to semget the semaphore. (Even more unfortunately, those systems don't completely support POSIX semaphores either.)

The system call to operate on a semaphore set, semop, is in the next section.

7.9.2 Simple Semaphore Interface

We can hide the complexities of semaphores with a simple semaphore-open call that we'll implement for System V now, and for POSIX semaphores a bit later on. Here's the synopsis for SimpleSemOpen and its companion, SimpleSemClose:

SimpleSemOpen—open simple semaphore

```
#include "SimpleSem.h"

struct SimpleSem *SimpleSemOpen(
    const char *name       /* name (follows System V or POSIX rules) */
);
/* Returns pointer to structure or NULL on error (sets errno) */
```

8. I learned of this technique from [Ste1999], p. 284. I had it wrong in the first edition of *Advanced UNIX Programming*.
9. I've been told it's fixed in FreeBSD 5.1.

SimpleSemClose—close simple semaphore

```
#include "SimpleSem.h"

bool SimpleSemClose(
    struct SimpleSem *sem  /* semaphore */
);
/* Returns true on success or false on error (sets errno) */
```

struct SimpleSem—structure for simple-semaphore functions

```
struct SimpleSem {
    union {
        int sm_semid;       /* System V semaphore-set ID */
        void *sm_sem;       /* POSIX sem_t pointer (needs a cast) */
    } sm;
};
```

`SimpleSemOpen` has no options at all: It opens a single semaphore by name (getting the key itself, in the case of a System V implementation), creating it if necessary with permissions `PERM_FILE` (from Section 2.3). It returns a pointer to a `SimpleSem` structure, which contains whatever the other functions need to identify the semaphore set (of one), which for System V is an integer semaphore-set ID. (We'll get to what POSIX semaphores require later.) But the user of the simple-semaphore package doesn't need to worry about what's inside a `SimpleSem` structure, as a pointer to it is just passed to the other functions, like `SimpleSemClose`, which closes the semaphore and frees any memory used by the SimpleSem structure.

Here's a System V implementation of `SimpleSemOpen`, where you can see the scheme outlined above for ensuring that the semaphore is properly initialized:

```
struct SimpleSem *SimpleSemOpen(const char *name)
{
    struct SimpleSem *sem = NULL;
    key_t key;
    union semun {
        int val;
        struct semid_ds *buf;
        unsigned short *array;
    } arg;
    struct sembuf sop;

    (void)close(open(name, O_WRONLY | O_CREAT, 0));
    ec_neg1( key = ftok(name, 1) )
    ec_null( sem = malloc(sizeof(struct SimpleSem)) )
```

```
    if ((sem->sm.sm_semid = semget(key, 1,
      PERM_FILE | IPC_CREAT | IPC_EXCL)) != -1) {
        arg.val = 0;
        ec_neg1( semctl(sem->sm.sm_semid, 0, SETVAL, arg) )
        sop.sem_num = 0;
        sop.sem_op = 0;
        sop.sem_flg = 0;
        ec_neg1( semop(sem->sm.sm_semid, &sop, 1) )
    }
    else {
        if (errno == EEXIST) {
            while (true)
                if ((sem->sm.sm_semid = semget(key, 1, PERM_FILE)) == -1) {
                    if (errno == ENOENT) {
                        sleep(1);
                        continue;
                    }
                    else
                        EC_FAIL
                }
                else
                    break;
            while (true) {
                struct semid_ds buf;

                arg.buf = &buf;
                ec_neg1( semctl(sem->sm.sm_semid, 0, IPC_STAT, arg) )
                if (buf.sem_otime == 0) {
                    sleep(1);
                    continue;
                }
                else
                    break;
            }
        }
        else
            EC_FAIL
    }
    return sem;

EC_CLEANUP_BGN
    free(sem);
    return NULL;
EC_CLEANUP_END
}
```

The sequence

```
    (void)close(open(name, O_WRONLY | O_CREAT, 0));
    ec_neg1( key = ftok(name, 1) )
```

is the same as what we did in Section 7.5.3. It creates the name if it doesn't already exist, and any problems with the name are reported as problems with `ftok`, not with `open` or `close`. (If you don't like that, you can check the `open` for an error other than `ENOENT` separately, instead of embedding it in an immediate call to `close`.)

Then we do a `semget` with the `IPC_CREAT` and `IPC_EXCL` flags so that it will fail if the semaphore already exists. All processes and threads for which it failed for that reason need to wait for it to be initialized. The process or thread that got a successful return from `semget` uses `semctl` to initialize it, and then it does a `semop` (which I'll explain shortly) that doesn't do anything to the value, but as a side-effect sets the `sem_otime`.

Processes and threads that need to wait will loop until a `semget` succeeds, sleeping each time, and then looping again until the time becomes nonzero, also sleeping each time.

For System V semaphores, `SimpleSemClose` doesn't do much:

```
bool SimpleSemClose(struct SimpleSem *sem)
{
    free(sem);
    return true;
}
```

To remove a simple semaphore, you call `SimpleSemRemove`:

SimpleSemRemove—remove simple semaphore

```
#include "SimpleSem.h"

bool SimpleSemRemove(
    struct SimpleSem *sem      /* semaphore */
);
/* Returns true on success or false on error (sets errno) */
```

It's OK to call `SimpleSemRemove` on a semaphore that doesn't exist—that's not considered an error. In fact, that's exactly what you should do when an application starts up and you want to start with a fresh semaphore.

Here's the code for `SimpleSemRemove`:

```
bool SimpleSemRemove(const char *name)
{
    key_t key;
    int semid;
```

```
    if ((key = ftok(name, 1)) == -1) {
        if (errno != ENOENT)
            EC_FAIL
    }
    else {
        if ((semid = semget(key, 1, PERM_FILE)) == -1) {
            if (errno != ENOENT)
                EC_FAIL
        }
        else
            ec_neg1( semctl(semid, 0, IPC_RMID) )
    }
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

Now, back to System V. We're ready to introduce the scary `semop` system call, which does both *semwait* and *sempost* operations:

semop—operate on semaphore set

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

int semop(
     int semid,                 /* identifier */
     struct sembuf *sops,       /* operations */
     size_t nsops               /* number of operations */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

struct sembuf—structure for semop

```
struct sembuf {
    unsigned short sem_num;    /* semaphore number */
    short sem_op;              /* Semaphore operation */
    short sem_flg;             /* Operation flags */
};
```

As I said, `semop` doesn't just operate on one semaphore but on as many as you like, even the whole set. You need to build an array of operations (`struct sembufs`) before you make the call, although if there's only one to be operated on, a single `struct sembuf` will do—you just pass its address and use 1 for `nsops`.

Each `sem_op` can be positive, negative, or zero:

> 0 The value of `sem_op` is added to the semaphore value (*sempost*).

< 0 The absolute value of `sem_op` is subtracted from the value, unless that would make the value go negative, in which case the call blocks until the entire value can be subtracted (*semwait*).

0 The call blocks until the value becomes zero.

All of the operations passed to `semop` are performed atomically, and the function doesn't return until everything is done. (Unless it's interrupted by thread cancellation, a signal, or the removal of the semaphore set.)

You can prevent blocking by setting the `IPC_NOWAIT` flag in the `sem_flg` member for an operation. If any operation in the array would block and has the flag set, `semop` returns immediately. Because `semop` always acts atomically, no other operations are done, even if they appeared earlier in the array and even if they wouldn't have blocked.

There's one more feature: For every semaphore that a process increments or decrements, an adjustment is kept, along with the actual value. When the `IPC_UNDO` flag is set for an incrementing operation, the adjustment is decremented, and vice versa for a decrementing operation. When a process exits, its adjustment is added to the semaphore's value, thus undoing whatever the process did to the semaphore. For example, suppose a process decrements a semaphore (i.e., *semwait*) to lock a buffer, and increments it (*sempost*) when it's done. With the `IPC_UNDO` flag, it can ensure that the buffer is unlocked if it should exit abnormally.

For our simple semaphores, we only want to operate on one at a time, we only increment or decrement by 1, and we don't use `IPC_NOWAIT` or `IPC_UNDO`. So `SimpleSemWait` and `SimpleSemPost` are, well, simple:

SimpleSemWait—decrement simple semaphore

```
#include "SimpleSem.h"

bool SimpleSemWait(
    struct SimpleSem *sem      /* semaphore */
);
/* Returns true on success or false on error (sets errno) */
```

SimpleSemPost—increment simple semaphore

```
#include "SimpleSem.h"

bool SimpleSemPost(
    struct SimpleSem *sem      /* semaphore */
);
/* Returns true on success or false on error (sets errno) */
```

Here's the code:

```
bool SimpleSemWait(struct SimpleSem *sem)
{
    struct sembuf sop;

    sop.sem_num = 0;
    sop.sem_op = -1;
    sop.sem_flg = 0;
    ec_neg1( semop(sem->sm.sm_semid, &sop, 1) )
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}

bool SimpleSemPost(struct SimpleSem *sem)
{
    struct sembuf sop;

    sop.sem_num = 0;
    sop.sem_op = 1;
    sop.sem_flg = 0;
    ec_neg1( semop(sem->sm.sm_semid, &sop, 1) )
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

I should mention one additional use of System V semaphores before we move on: to pass an integer between processes. Suppose, for example, a client wants to pass its process ID to a server. It gets access to a semaphore and sets its value to its process ID. Then the server can look at the value to get the number. Since a semaphore set can have an array of semaphores, you can pass an array of integers this way.

7.10 POSIX Semaphores

These semaphore system calls are part of the POSIX standard, though optional, and you have to check the `_POSIX_SEMAPHORES` feature-test macro to tell whether they're present (see Section 1.5.4 and Section 7.6.3). As of now, they're not in FreeBSD, Darwin, or Linux.

7.10.1 Named POSIX Semaphores

POSIX semaphores are much simpler and much easier to use than System V semaphores. In fact, five of the system calls line right up with the SimpleSem interface from the previous section:

sem_open—open named semaphore

```
#include <semaphore.h>

sem_t *sem_open(
    const char *name,      /* POSIX IPC name */
    int flags              /* flags (excluding O_CREAT) */
);
/* Returns pointer to semaphore or SEM_FAILED on error (sets errno) */

sem_t *sem_open(
    const char *name,      /* POSIX IPC name */
    int flags,             /* flags (including O_CREAT) */
    mode_t perms,          /* permissions */
    unsigned value         /* initial value */
);
/* Returns pointer to semaphore or SEM_FAILED on error (sets errno) */
```

sem_close—close named semaphore

```
#include <semaphore.h>

int sem_close(
    sem_t *sem             /* semaphore */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

sem_unlink—remove named semaphore

```
#include <semaphore.h>

int sem_unlink(
    const char *name       /* POSIX IPC name */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

sem_wait—decrement semaphore

```
#include <semaphore.h>

int sem_wait(
    sem_t *sem             /* semaphore */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

sem_post—increment semaphore

```
#include <semaphore.h>

int sem_post(
     sem_t *sem/* semaphore */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

POSIX semaphores are counting semaphores, like the System V variants, but in steps of one only. Each `sem_t` object represents only one semaphore, not a set of them, as with System V.

The calls for opening, closing, and unlinking follow the pattern for POSIX IPC calls that we saw in Section 7.7.1 for POSIX message-queue system calls. In particular, the name passed to `sem_open` has to follow the portability rules that were discussed in Section 7.6.2. As you would expect, `sem_post` adds one to the semaphore's value, and `sem_wait` subtracts one, blocking if the value is already zero.

You don't use `O_RDONLY`, `O_WRONLY`, or `O_RDWR` with `sem_open`, as they're meaningless—the semaphore is useless unless `sem_post` and `sem_wait` can both be used.

All the initialization rigmarole[10] that we needed for System V is gone. We just pass the value we want (often zero) to `sem_open`.

Be careful about the return value from `sem_open` when it fails: It's `SEM_FAILED`, not `NULL`, which is the usual failure return from functions that otherwise return a pointer.

The implementation of the SimpleSem calls for POSIX semaphores is trivial:

```
struct SimpleSem *SimpleSemOpen(const char *name)
{
    struct SimpleSem *sem = NULL;

    ec_null( sem = malloc(sizeof(struct SimpleSem)) )
    if ((sem->sm.sm_sem = sem_open(name, O_CREAT, PERM_FILE, 0)) ==
      SEM_FAILED)
        EC_FAIL
    return sem;
```

10. If you're not familiar with this term, one dictionary I consulted defined it as "a complicated, petty set of procedures," which is just perfect.

```
EC_CLEANUP_BGN
    free(sem);
    return NULL;
EC_CLEANUP_END
}

bool SimpleSemClose(struct SimpleSem *sem)
{
    ec_neg1( sem_close(sem->sm.sm_sem) )
    free(sem);
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}

bool SimpleSemRemove(const char *name)
{
    if (Sem_unlink(name) == -1 && errno != ENOENT)
        EC_FAIL
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}

bool SimpleSemPost(struct SimpleSem *sem)
{
    ec_neg1( sem_post(sem->sm.sm_sem) )
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}

bool SimpleSemWait(struct SimpleSem *sem)
{
    ec_neg1( sem_wait(sem->sm.sm_sem) )
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

POSIX semaphores have some additional features that we didn't need for the SimpleSem interface. First of all, as with System V, you can query the value of a semaphore without modifying it or waiting:

sem_getvalue—get value of semaphore

```
#include <semaphore.h>

int sem_getvalue(
    sem_t *restrict sem,       /* semaphore */
    int *valuep                /* returned value */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

If the value of the semaphore is greater than zero during the call to `sem_getvalue`, that value is returned. It could be something different, however, by the time `sem_getvalue` returns, so the actual number isn't that useful. If the value is zero, the value returned is the negative of the number of processes waiting on the semaphore. If there aren't any, zero is returned.

There are two variations on `sem_wait`: One, `sem_trywait`, is nonblocking:

sem_trywait—decrement semaphore if possible

```
#include <semaphore.h>

int sem_trywait(
    sem_t *sem                 /* semaphore */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

`sem_trywait` returns –1 with `errno` set to `EAGAIN` if the semaphore is already zero. (Other system calls use a flag like `O_NONBLOCK` or `IPC_NOWAIT` for nonblocking; here it's a separate system call.)

The other `sem_timedwait` variant times out after an interval if the semaphore doesn't become positive:

sem_timedwait—decrement semaphore

```
#include <semaphore.h>
#include <time.h>

int sem_timedwait(
    sem_t *restrict sem,           /* semaphore */
    const struct timespec *time    /* absolute time */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

The time passed to `sem_timedwait` is an absolute time (e.g., 1:23:17 PM), not a time interval (e.g., 27 secs.). What clock it's compared to and with what resolution depends on whether the POSIX Timers option is supported. If so, it's the real-time clock. If not, the ordinary clock (as used by the `time` system call) is used. (See Section 1.7 for a discussion of `struct timespec` and other aspects of UNIX time.)

`sem_timedwait` is part of the Timeouts option (`_POSIX_TIMEOUTS`) and is new with SUS3.

7.10.2 Unnamed POSIX Semaphores

Quick review: `sem_open`, `sem_close`, and `sem_unlink` are used with named semaphores that exist external to a process somewhere and are accessed by a POSIX IPC name. `sem_open` gets you a pointer to a `sem_t` object, which you never deal with directly. Whatever memory it uses is freed when you call `sem_close`. These semaphores inherently work between threads and processes.

To make semaphores faster, you can also declare a `sem_t` object directly:

```
sem_t sem;
```

or even allocate one dynamically, like this:

```
sem_t *semp = malloc(sizeof(sem_t));
```

But if you allocate the `sem_t` object yourself, you have to call `sem_init` to initialize it:

sem_init—initialize unnamed semaphore

```
#include <semaphore.h>

int sem_init(
    sem_t *sem,            /* semaphore */
    int pshared,           /* shared between processes? */
    unsigned value         /* initial value */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

and then you call `sem_destroy` to destroy it:

sem_destroy—destroy unnamed semaphore

```
#include <semaphore.h>

int sem_destroy(
    sem_t *sem         /* semaphore */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

`sem_destroy` doesn't deallocate the `sem_t` object, as it knows nothing about how the memory got there (static declaration, `malloc`, etc.). You have to free the memory (if that's appropriate) yourself. If the semaphore was allocated globally or on the stack, of course, you don't have to do anything; its life will end at the appropriate time.

`sem_init` and `sem_destroy` are substitutes for `sem_open` and `sem_close`; you use one pair or the other, depending on whether the semaphore is named or not. The other calls (e.g., `sem_post`, `sem_timedwait`) don't care how you got the `sem_t` pointer.

OK, unnamed semaphores may be fast, but if they're in the memory of a single process, what good are they? They're real good:

- They work fine for synchronizing between threads when you want a counting semaphore, not just a mutex that has only two states (binary semaphore).
- They work fine between processes if they're in shared memory, which we'll get to shortly. In this case the `pshared` argument to `sem_init` has to be nonzero.

The first use is important for threads, and some UNIX implementations, notably Linux and FreeBSD (but not Darwin), support it even though some versions support POSIX semaphores completely. That is, `sem_init` (`pshared` zero only) and `sem_destroy` are supported but not `sem_open` and `sem_close`. In these limited implementations `sem_post`, `sem_wait`, `sem_trywait`, and `sem_getvalue` are also supported but not `sem_timedwait`.

One more rule about unnamed (in-memory) semaphores: Only the actual memory passed to `sem_init` can be used, not a copy of it. So the following is wrong:

```
void fcn(sem_t s)
{
    sem_post(&s);
    ...
}

void fcn2(void)
{
    sem_t sem;

    sem_init(&sem);
    fcn(sem);
    ...
}
```

Calling `fcn` copies the semaphore, which violates the rule. Rather, all parts of the program that manipulate the semaphore need to work with the address of the originally initialized storage. I'm going to provide an example of an unnamed semaphore located in shared memory in Section 7.14.2.

7.10.3 System V and POSIX Semaphores Critiqued

For interprocess communication, the System V semaphore system calls are ridiculously difficult to use and overwrought with features, but they're universally available and not so bad once the hard stuff (initialization, mainly) is hidden behind a reasonable interface like SimpleSem. POSIX semaphores are much easier to use but not universally available. If portability is important to you, you need to use the System V system calls, and there's nothing to be gained by using the POSIX calls anywhere if you can't use them everywhere. If portability isn't important, and the POSIX calls are supported on the UNIX systems of interest, then clearly those are the ones to use.[11]

For intraprocess communication—that is, between threads—the System V semaphores are much too cumbersome and slow, and unnamed, nonprocess-shared POSIX semaphores are usually available wherever POSIX Threads are, so those are the ones to use if you need something beyond mutexes.

7.10.4 Process-Shared Mutexes and Read-Write Locks

I introduced mutexes in Section 5.17.3 for use in synchronizing threads, but they were in-memory and within a process—essentially, in-memory, nonprocess-shared POSIX semaphores used in a binary fashion. You can also create an in-memory mutex and initialize it with `pthread_mutex_init` with the `PTHREAD_PROCESS_SHARED` attribute set, in which case it's shared between threads that may be in different processes. However, this feature isn't always implemented even if POSIX Threads are (it's a separate suboption), so it may not be available.

11. Be careful about assuming that portability isn't important. Almost every application nowadays is a candidate for someday being ported to Linux. And potential portability, even if it never happens, may be helpful in negotiations with vendors.

There are also POSIX Thread read-write locks, which I didn't describe in Chapter 5 at all. These are like mutexes, but they distinguish between reading and writing (share and exclusive), to allow more throughput. As with mutexes, there's a `PTHREAD_PROCESS_SHARED` attribute you can set. (As we'll see in the next section, read-write locks are to mutexes as `fcntl` file-locking is to `lockf` file-locking [Sections. 7.11.3 and 7.11.4.)]

7.11 File Locking

This section explains how file locking works in UNIX and how it sometimes doesn't work the way you need it to.

7.11.1 A Bad Example

We like to do things wrong at first—it provides some motivation for getting it right.

To motivate our discussion of file locking, let's build an example application that has one process building a file while another is reading it. The file is a linked list of records, each of which holds an integer of data and the offset of the next record. The linked list is sorted, but the physical position of the records is in their order of creation. The structure for each record is:

```
struct rec {
    int r_data;
    off_t r_next;
};
```

If we insert records with data of 1000, 999, and 998, in that order, the file looks like Figure 7.2. The first record is just a header that tells where the list starts.

The example's `main` program starts a child process (`process1`) to build the list while the parent (`process2`) traverses it repeatedly to check if it's properly formed:

```
int main(void)
{
    pid_t pid;
```

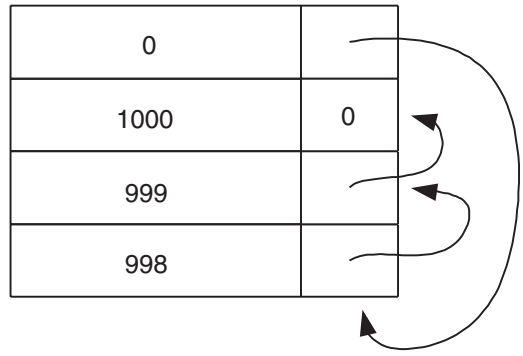


Figure 7.2 Linked list of records.

```
    ec_neg1( pid = fork() )
    if (pid == 0)
        process1();
    else {
        process2();
        ec_neg1( waitpid(pid, NULL, 0) )
    }
    exit(EXIT_SUCCESS);

EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
EC_CLEANUP_END
}
```

For data, `process1` just counts backwards from 1000 after writing the header record:

```
#define DBNAME "termdb"

static void process1(void)
{
    int dbfd, data;
    struct rec r;

    ec_neg1( dbfd = open(DBNAME, O_CREAT | O_TRUNC | O_RDWR, PERM_FILE) )
    memset(&r, 0, sizeof(r));
    ec_false( writerec(dbfd, &r, 0) )
    for (data = 100; data >= 0; data--)
        ec_false( store(dbfd, data) )
    ec_neg1( close(dbfd) )
    exit(EXIT_SUCCESS);

EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
```

```
EC_CLEANUP_END
}
```

To make the I/O easy, there are two functions: `writerec` writes a record at a specified offset, and `readrec` reads a record from an offset. They both consider partial writes or reads or an end-of-file to be errors:

```
bool readrec(int dbfd, struct rec *r, off_t off)
{
    ssize_t nread;

    if ((nread = pread(dbfd, r, sizeof(struct rec), off)) ==
      sizeof(struct rec))
        return true;
    if (nread != -1)
        errno = EIO;
    EC_FAIL
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}

bool writerec(int dbfd, struct rec *r, off_t off)
{
    ssize_t nwrote;

    if ((nwrote = pwrite(dbfd, r, sizeof(struct rec), off)) ==
      sizeof(struct rec))
        return true;
    if (nwrote != -1)
        errno = EIO;
    EC_FAIL
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

We could have used `pwrite` and `pread` directly almost as easily, but the error checking is a bit convoluted so it makes sense to encapsulate them.

The function `store` (called from `process1`) is responsible for keeping the linked list in order. Note that it doesn't try to minimize I/O:

```
bool store(int dbfd, int data)
{
```

```
    struct rec r, rnew;
    off_t end, prev;

    ec_neg1( end = lseek(dbfd, 0, SEEK_END) )
    prev = 0;
    ec_false( readrec(dbfd, &r, prev) )
    while (r.r_next != 0) {
        ec_false( readrec(dbfd, &r, r.r_next) )
        if (r.r_data > data)
            break;
        prev = r.r_next;
    }
    ec_false( readrec(dbfd, &r, prev) )
    rnew.r_next = r.r_next;
    r.r_next = end;
    ec_false( writerec(dbfd, &r, prev) )
    rnew.r_data = data;
    usleep(1); /* give up CPU */
    ec_false( writerec(dbfd, &rnew, end) )
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

Spend a bit of time with `store` and it should become clear. Note that it handles the cases of the new record going at the beginning or at the end properly. The call to `usleep` (sleeps for 1 microsecond) near the end is just to give up the CPU to let other processes run a bit. This happens naturally in complicated programs, but I had to force it in this simple one because I want the other process, which I'm about to show, to run concurrently.

Here's `process2` which checks the integrity of the file:

```
static void process2(void)
{
    int try, dbfd;
    struct rec r1, r2;

    for (try = 0; try < 10; try++)
        if ((dbfd = open(DBNAME, O_RDWR)) == -1) {
            if (errno == ENOENT) {
                continue;
            }
            else
                EC_FAIL
        }
    ec_neg1( dbfd )
```

```
    for (try = 0; try < 100; try++) {
        ec_false( readrec(dbfd, &r1, 0) )
        while (r1.r_next != 0) {
            ec_false( readrec(dbfd, &r2, r1.r_next) )
            if (r1.r_data > r2.r_data) {
                printf("Found sorting error (try %d)\n", try);
                break;
            }
            r1 = r2;
        }
    }
    ec_neg1( close(dbfd) )
    return;

EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
EC_CLEANUP_END
}
```

I try a few times to open the database because it may take `process1` a while to get scheduled and to complete its call to `open`. Then I repeatedly (100 times) go through the list looking for errors, such as a missing record or out-of-order data.

Sure enough, here's what I got when I ran the program:

```
ERROR:  0: process2 [/aup/c7/f1.c:107] readrec(dbfd, &r2, r1.r_next)
        1: readrec [/aup/c7/f1.c:17] 0
                *** EIO (5: "I/O error") ***
```

This particular error resulted because a new record that `store` was about to write to the end of the database hadn't gotten written when `process2` looked for it.

Now we have an application that doesn't work, which is no great accomplishment, but ours doesn't work specifically because two processes are accessing the same file and one of them is finding inconsistent data. They need some coordination.

So, are you motivated to see what this section is about?

7.11.2 Using a Semaphore as a File Lock

The obvious way to fix the example from the previous section is to use a semaphore to prevent `process2` from seeing an inconsistent file. Using the SimpleSem interface from Section 7.9.2, we can define a global like this:

```
static struct SimpleSem *sem;
```

Each process independently opens the semaphore with a line like this:

```
ec_null( sem = SimpleSemOpen("sem") )
```

In addition, `process1` initially increments the semaphore to indicate that the database is consistent:

```
ec_false( SimpleSemPost(sem) )
```

The critical lines in `store` are the ones that update a record in place and store a new record at the end:

```
ec_false( SimpleSemWait(sem) )
ec_false( readrec(dbfd, &r, prev) )
rnew.r_next = r.r_next;
r.r_next = end;
ec_false( writerec(dbfd, &r, prev) )
rnew.r_data = data;
usleep(1); /* give up CPU */
ec_false( writerec(dbfd, &rnew, end) )
ec_false( SimpleSemPost(sem) )
```

And the critical lines in `process2` are the ones that traverse the list:

```
for (try = 0; try < 100; try++) {
    ec_false( SimpleSemWait(sem) )
    ec_false( readrec(dbfd, &r1, 0) )
    while (r1.r_next != 0) {
        ec_false( readrec(dbfd, &r2, r1.r_next) )
        if (r1.r_data > r2.r_data) {
            printf("Found sorting error (try %d)\n", try);
            break;
        }
        r1 = r2;
    }
    ec_false( SimpleSemPost(sem) )
}
```

With these changes, the application works perfectly. (Except on FreeBSD and Darwin, where System V semaphores are incorrectly implemented, as explained in Section 7.9.1.)

The chief problem with using a semaphore as a file lock is that for an arbitrary file it's not clear what the semaphore name should be. This isn't a problem for an application with a few fixed file names, but dealing with files on an ad hoc basis would be much too awkward. We would have to invent some facility for mapping file names to semaphore names. To make things worse, sometimes only part of a file needs to be locked, and that would mean different semaphores for different

parts. Finally, managing semaphores—getting them open and closed, and removing them when they get left around—is a pain.

7.11.3 `lockf` System Call

The hassle of using a semaphore to lock a file isn't really a problem because UNIX has a special system call to lock a section of a file:

lockf—lock section of file

```
#include <unistd.h>

int lockf(
     int fd,            /* file descriptor */
     int op,            /* operation */
     off_t len          /* length of section */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

The file descriptor must be opened for writing (`O_WRONLY` or `O_RDWR`).

The section to be locked or unlocked is from the current file offset (set by a `read`, a `write`, or an `lseek`) for `len` bytes either forward or, if `len` is negative, backward. In the case of backward, the byte at the current offset is not part of the section. In the case of forward, bytes can be locked that don't exist because the file hasn't gotten that big yet. If `len` is zero, the section extends from the current offset to the end of the file, even as the file grows. So, to lock the whole file, just make sure the offset is zero and use a length of zero.

If a section to be locked overlaps a section already locked, the two sections are merged. If part of a locked section is unlocked, the locked section is made smaller, and is possibly split into two discontinuous sections.

All locks that a process has on a file are released when *any* file descriptor that that process has open on the file is closed, even if the file descriptor that's closed was obtained independently (i.e., different `open`) from the one that was passed to `lockf`.[12] It follows that any locks are released when a process terminates since

12. From a FreeBSD man page: "This interface follows the completely stupid semantics of System V and IEEE Std 1003.1-1988 ('POSIX.1') that require that all locks associated with a file for a given process are removed when any file descriptor for that file is closed by that process. This semantic means that applications must be aware of any files that a subroutine library may access. For example if an application for updating the password file locks the password file database while making the update, and then calls getpwnam(3) to retrieve a record, the lock will be lost because getpwnam(3) opens, reads, and closes the password database." BSD-based systems have a better call, `flock`, but it's nonstandard.

that causes all file descriptors to be closed. Locks are *not* inherited when a process calls `fork`.

Here are the operations for the `op` argument:

`F_LOCK` Lock the section; block if any part of it is already locked by another process.

`F_TLOCK` Like `F_LOCK`, but return –1 with `errno` set to `EAGAIN` (or `EACCES`) if `F_LOCK` would have blocked.

`F_TEST` Don't lock, but return an error as `F_TLOCK` would if `F_LOCK` would have blocked.

`F_ULOCK` Unlock the section.

You don't have to have a semaphore name or open anything special to use `lockf` on a file, since it takes the same file descriptor you use to access the file. So, we can easily slip it into our example in place of the semaphore calls from the previous section. I'll show just the critical part of `store`:

```
ec_neg1( lseek(dbfd, 0, SEEK_SET) )
ec_neg1( lockf(dbfd, F_LOCK, 0) )
ec_false( readrec(dbfd, &r, prev) )
rnew.r_next = r.r_next;
r.r_next = end;
ec_false( writerec(dbfd, &r, prev) )
rnew.r_data = data;
usleep(1); /* give up CPU */
ec_false( writerec(dbfd, &rnew, end) )
ec_neg1( lseek(dbfd, 0, SEEK_SET) )
ec_neg1( lockf(dbfd, F_ULOCK, 0) )
```

and the critical part of `process2`:

```
for (try = 0; try < 100; try++) {
    ec_neg1( lseek(dbfd, 0, SEEK_SET) )
    ec_neg1( lockf(dbfd, F_LOCK, 0) )
    ec_false( readrec(dbfd, &r1, 0) )
    while (r1.r_next != 0) {
        ec_false( readrec(dbfd, &r2, r1.r_next) )
        if (r1.r_data > r2.r_data) {
            printf("Found sorting error (try %d)\n", try);
            break;
        }
        r1 = r2;
    }
    ec_neg1( lseek(dbfd, 0, SEEK_SET) )
    ec_neg1( lockf(dbfd, F_ULOCK, 0) )
}
```

7.11.4 `fcntl` System Call for File Locking

You can also lock a file with the `fcntl` system call, which we first encountered in Section 3.8.3. Its locking functionality is a superset of `lockf`'s. Here's a recap of the `fcntl` synopses:

fcntl—control open file

```
#include <unistd.h>
#include <fcntl.h>

int fcntl(
    int fd,             /* file descriptor */
    int op,             /* operation */
    ...                 /* optional argument depending on op */
);
/* Returns result depending on op or -1 on error (sets errno) */
```

There are three `fcntl` operations for locking that operate on a structure, a pointer to which is passed as the third argument:

struct flock—structure for fcntl file locking

```
struct flock {
    short l_type;       /* lock type: F_RDLCK, F_WRLCK, F_UNLCK */
    short l_whence;     /* interpretation of l_start */
    off_t l_start;      /* start of section */
    off_t l_len;        /* length of section */
    pid_t l_pid;        /* process holding lock; used with F_GETLK */
};
```

Three members of the structure, `l_whence`, `l_start`, and `l_len`, establish the section to be operated on. The first two are like the arguments to `lseek` (`l_whence` can be `SEEK_SET`, `SEEK_CUR`, or `SEEK_END`), and `l_start` is absolute, relative to the current file offset, or relative to the end of the file. The length of the section is given by `l_len`.

With `lockf` there is only one type of lock, but with `fcntl` you can have both read locks and write locks, or, as they're commonly called, *share* locks and *exclu-*

sive locks. A share lock on a section prevents an exclusive lock on that section; an exclusive lock prevents a share or exclusive lock. In practice, you set a share lock when you will only be reading data, and an exclusive lock when you will be writing it.

You specify the lock type in the `l_type` member; the third choice shown in the synopsis is for unlocking a section.

Now we're ready to explain the `fcntl` operations for locking, which are pretty simple:

`F_SETLK` Perform the operation specified in the structure, if possible. If a lock can't be set immediately, return –1 with `errno` set to `EAGAIN` or `EACCES`; that is, do not block.

`F_SETLKW` Just like `F_SETLK`, but block if the lock can't be set immediately.

`F_GETLK` Return information about the first lock that would cause the lock specified in the structure to block, if any. All members of the passed-in structure are overwritten with the results, including the process ID of the process holding the lock. If the lock passed in would not block, the structure is passed back unchanged except that the first member is changed to `F_UNLCK`.

Thus, `fcntl`-locking can do everything `lockf` can. In addition, it distinguishes between share and exclusive locks, and it can retrieve information about existing locks. Usually, implementations implement `lockf` as a library function on top of `fcntl`, but that's not required. Also, [SUS2002] says that you shouldn't assume that the locks manipulated by the two functions are the same. That is, if you lock with `lockf`, don't unlock with `fcntl`. In fact, don't expect `fcntl` to even know about the lock.

As with `lockf` locks, locks set with `fcntl` are released when a file descriptor open to the file is closed or when the process terminates, and they are not inherited when a process calls `fork`.

7.11.5 Advisory and Mandatory Locks

The locks set with `lockf` and `fcntl` normally affect only those function calls, not other I/O operations. That is, if you set a lock on a file with `lockf` and then another process writes on the file without calling `lockf`, that write will proceed. This is called *advisory* locking, and obviously it works only if all the processes cooperate by calling `lockf` or `fcntl` appropriately.

Mandatory locking means that once a lock is set, it really does prohibit a conflicting I/O operation. The POSIX and SUS standards don't specify mandatory locking at all, but they don't prohibit it either.

On those systems that support mandatory locking, you don't do anything differently in the calls to `lockf` or `fcntl`. Instead you mark the file itself with a set of permissions that otherwise make no sense: set-group-ID-on-execution bit on, and group-execute bit off. Here's an example program that uses a lock that's advisory or mandatory, depending on the argument:

```
int main(int argc, char *argv[])
{
    int fd;
    mode_t perms = PERM_FILE;

    if (fork() == 0) {
        sleep(1); /* wait for parent */
        ec_neg1( fd = open("tmpfile", O_WRONLY | O_NONBLOCK) )
        ec_neg1( write(fd, "x", 1) )
        printf("child wrote OK\n");
    }
    else {
        (void)unlink("tmpfile");
        if (argc == 2)
            perms |= S_ISGID; /* mandatory locking */
        ec_neg1( fd = open("tmpfile", O_CREAT | O_RDWR, perms) )
        ec_neg1( lockf(fd, F_LOCK, 0) )
        printf("parent has lock\n");
        ec_neg1( wait(NULL) )
    }
    exit(EXIT_SUCCESS);

EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
EC_CLEANUP_END
}
```

Here's what I got when I ran this on Solaris, which has mandatory locking:

```
$ lockftest
parent has lock
child wrote OK
$ lockftest x
parent has lock
ERROR:  0: main [/aup/c7/lockftest.c:11] write(fd, "x", 1)
                *** EAGAIN (11: "Resource temporarily unavailable") ***
$
```

Without the `O_NONBLOCK` flag, the write in the child process would have blocked waiting for the lock to be released.

7.11.6 High-Performance Database Locking

File locking as provided for by the `fcntl` and `lockf` system calls, even with mandatory locking, isn't suitable for high-performance databases because there's too much overhead in manipulating the locks, and it's too hard to implement sophisticated deadlock detection and correction algorithms. That's usually OK, however, because big database systems run in their own processes anyway, so they act as a gatekeeper to the database files. Locks can easily be kept in the address space of the database process or in memory that's shared among multiple database processes. Done this way, there's no need at all to use system calls to manage the locks.

7.12 About Shared Memory

Recall from Section 5.17 that threads within a process share all static data—global data and static data that's internal to a function. Processes, on the other hand, have entirely separate memory, even if a `fork` was executed without an `exec`, in which case the child gets a copy of the parent's address space.

With shared memory, you can arrange for separate processes to have some memory in common. As with threads, they usually need to use mutexes or semaphores to coordinate their access to the memory.

There are both System V and POSIX versions of shared memory, just as with messages and semaphores. For both mechanisms, each process "opens" a shared-memory segment and gets a pointer to the memory, which it is then free to use with ordinary C or C++ operators, without using a system call. Normally, each process's pointer has a different value, meaningful only within that process, but

the underlying memory is the same. As we did before, we'll start with System V shared memory and then move on to POSIX.

7.13 System V Shared Memory

By now you should be familiar with how the System V IPC calls work. As Section 7.4 explained, you get a key, with `ftok` if you like, and use an *X*`get` call (`shmget`) to get an identifier. You control it with an *X*`ctl` call (`shmctl`). In the case of shared memory, you attach it to your process with `shmat`, which returns a pointer, and detach it with `shmdt` when you're done with it.

7.13.1 System V Shared-Memory System Calls

Here are the synopses for the System V shared-memory system calls:

shmget—get shared memory segment

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

int shmget(
     key_t key,                 /* key */
     size_t size,               /* size of segment */
     int flags                  /* creation flags */
);
/* Returns shared-memory identifier or -1 on error (sets errno) */
```

shmctl—control shared memory segment

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

int shmctl(
     int shmid,                 /* identifier */
     int cmd,                   /* command */
     struct shmid_ds *data      /* data for command */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

struct shmid_ds—structure for shmctl

```
struct msqid_ds {
     struct ipc_perm shm_perm;  /* permission structure */
     size_t shm_segsz;          /* size of segment in bytes */
     pid_t shm_lpid;            /* process ID of last shared memory op */
     pid_t shm_cpid;            /* process ID of creator */
     shmatt_t shm_nattch;       /* number of current attaches */
     time_t shm_atime;          /* time of last shmat */
     time_t shm_dtime;          /* time of last shmdt */
     time_t shm_ctime;          /* time of last change by shmctl */
};
```

shmat—attach shared memory segment

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

void *shmat(
     int shmid,             /* identifier */
     const void *shmaddr,   /* desired address or NULL */
     int flags              /* attachment flags */
);
/* Returns pointer or -1 on error (sets errno) */
```

shmdt—detach shared memory segment

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

int shmdt(
     const void *shmaddr    /* pointer to segment */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

As you'd expect, `shmget` accesses a shared memory segment, creating it if necessary and if the `IPC_CREAT` or `IPC_PRIVATE` flags are present. The `size` argument is meaningful only if the segment is created. A newly created segment is initialized to zeros.

Then, to use the segment, you call `shmat`, which gives you a pointer to it. Oddly, `shmat` returns –1 on an error, not `NULL`. This works because it will never return a pointer that can be mistaken for –1; in fact, returned pointers are even numbers on essentially all UNIX systems.

When you're done with the segment, you call `shmdt`, passing in the pointer you got from `shmat`, not the identifier. The segment stays around until it's explicitly removed (via `shmctl` or the `ipcrm` command) or until the machine is rebooted, so you're free to reattach it. You might get a different pointer, though.

Normally, you don't care what address `shmat` gives you, so you set the second argument to `NULL`. But, if you want, you can try to force it to give you the address you specify with `shmaddr`. If it can't, because that address is already in use or otherwise invalid, it fails with `errno` set to `EINVAL`. There's also a flag, `SHM_RND`, for rounding the address to a proper boundary, but I won't describe the details. Another flag, `SHM_RDONLY`, if set, attaches the segment read-only.

With `shmctl` you use the same commands as with the other *X*`ctl` system calls, `IPC_STAT`, `IPC_SET`, and `IPC_RMID`. `IPC_SET` sets `shm_perm.uid`, `shm_perm.gid`, and the low-order 9 bits of `shm_perm.mode`.

Here's a simple program that shows how a segment (a small one!) can be shared between two processes:

```
static int *getaddr(void)
{
    key_t key;
    int shmid, *p;

    (void)close(open("shmseg", O_WRONLY | O_CREAT, 0));
    ec_neg1( key = ftok("shmseg", 1) )
    ec_neg1( shmid = shmget(key, sizeof(int), IPC_CREAT | PERM_FILE) )
    ec_neg1( p = shmat(shmid, NULL, 0) )
    return p;

EC_CLEANUP_BGN
    return NULL;
EC_CLEANUP_END
}

int main(void)
{
    pid_t pid;

    if ((pid = fork()) == 0) {
        int *p, prev = 0;

        ec_null( p = getaddr() )
        while (*p != 99)
            if (prev != *p) {
                printf("child saw %d\n", *p);
                prev = *p;
            }
        printf("child is done\n");
    }
    else {
        int *p;

        ec_null( p = getaddr() )
        for (*p = 1; *p < 4; (*p)++)
            sleep(1);
        *p = 99;
    }
    exit(EXIT_SUCCESS);

EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
EC_CLEANUP_END
}
```

Each process calls `getaddr`; one of them creates the segment, and the other just accesses it. Here's the output we got:

```
$ shmex
child saw 1
child saw 2
child saw 3
$ child saw 99
child is done
```

A little strange, no? Why did the child report that it saw 99 when that's what terminates its `while` loop? Well, in the line

```
while (*p != 99)
```

`*p` was equal to 3 at one point, but by the time it got to the line

```
printf("child saw %d\n", *p);
```

`*p` was already 99. (The `$` prompt is where it is because the parent didn't wait for the child before exiting. That's not a defect but a pretty common occurrence in UNIX.)

Things get even stranger if you run `shmex` a second time:

```
$ shmex
child is done
$
```

It broke! No, actually all that's wrong is that the segment was left there—unattached—after `shmex` terminated the first time, and it still held the value 99. In the second execution, the child saw 99 before the parent even got to its `for` loop. Obviously, we could fix this problem by removing the segment when `shmex` terminates. But you get the point: Sharing memory is tricky!

7.13.2 Shared Memory and Semaphores

The example in the previous section is wrong for another, more subtle reason: As we saw with threads (Section 5.17.3), we can't assume that references to `*p` are atomic. Generally, you can't share memory between processes without some control in the form of a semaphore. Therefore, let's add that in (`getaddr` is unchanged):

```
int main(void)
{
    pid_t pid;

    ec_false( SimpleSemRemove("shmexsem") )
    if ((pid = fork()) == 0) {
        struct SimpleSem *sem;
        int *p, prev = 0, n;

        ec_null( sem = SimpleSemOpen("shmexsem") )
        ec_null( p = getaddr() )
        while (true) {
            ec_false( SimpleSemWait(sem) )
            n = *p;
            ec_false( SimpleSemPost(sem) )
            if (n == 99)
                break;
            if (prev != n) {
                printf("child saw %d\n", n);
                prev = n;
            }
        }
        printf("child is done\n");
        ec_false( SimpleSemClose(sem) )
    }
    else {
        struct SimpleSem *sem;
        int *p, i;

        ec_null( sem = SimpleSemOpen("shmexsem") )
        ec_null( p = getaddr() )
        *p = 0;
        ec_false( SimpleSemPost(sem) )
        for (i = 1; i < 4; i++) {
            ec_false( SimpleSemWait(sem) )
            *p = i;
            ec_false( SimpleSemPost(sem) )
            sleep(1);
        }
        ec_false( SimpleSemWait(sem) )
        *p = 99;
        ec_false( SimpleSemPost(sem) )
        ec_false( SimpleSemClose(sem) )
    }
    exit(EXIT_SUCCESS);

EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
EC_CLEANUP_END
}
```

Here's the output now—it's much more sensible:

```
$ shmex2
child saw 1
child saw 2
child saw 3
$ child is done
shmex2
child saw 1
child saw 2
child saw 3
$ child is done
```

Look at all the changes we needed to put in the protection:

- The child assigned `*p` to local memory with the semaphore locked and then was free to use the local memory with the semaphore unlocked.
- Similarly, the parent used a local variable in the `for` loop, locking the semaphore only to access the shared memory.
- Initially, the semaphore is locked (zero value), so the parent is free to initialize the shared memory to zero. Then it calls `SimpleSemPost` to get things moving. It's OK if the child accesses the shared memory at that point. This version then can be run repeatedly since it initializes the segment each time it's run.
- We remove the semaphore at the start of each run so that it will start with zero.

Although we've fixed the atomicity problems, the child process is still inefficient. To see why, look again at its loop:

```
while (true) {
    ec_false( SimpleSemWait(sem) )
    n = *p;
    ec_false( SimpleSemPost(sem) )
    if (n == 99)
        break;
    if (prev != n) {
        printf("child saw %d\n", n);
        prev = n;
    }
}
```

It keeps racing around and around, locking and unlocking the semaphore, but only does something (prints) when the value changes. Most of its efforts are wasted, and all the locking keeps the semaphore unavailable for no reason. Wouldn't it be better for the parent to just tell the child when the value changed? (This is reminiscent of the motivation for condition variables in Section 5.17.4.)

We can fix the program with two semaphores instead of one: A semaphore *W* that must be locked to write to the shared memory, and a semaphore *R* for reading it. The parent waits on *W* before writing and then posts *R* when it's done. The child waits on *R* before reading and then posts *W* when it's done. To get things started, *W* is initialized to 1 (posted), and *R* is left at zero. Here's the revised code (`getaddr` is still unchanged):

```
int main(void)
{
    pid_t pid;

    ec_false( SimpleSemRemove("shmexsem") )
    if ((pid = fork()) == 0) {
        struct SimpleSem *semR, *semW;
        int *p, n;

        ec_null( semR = SimpleSemOpen("shmexsemR") )
        ec_null( semW = SimpleSemOpen("shmexsemW") )
        ec_null( p = getaddr() )
        while (true) {
            ec_false( SimpleSemWait(semR) )
            n = *p;
            ec_false( SimpleSemPost(semW) )
            if (n == 99)
                break;
            printf("child saw %d\n", n);
        }
        printf("child is done\n");
        ec_false( SimpleSemClose(semR) )
        ec_false( SimpleSemClose(semW) )
    }
    else {
        struct SimpleSem *semR, *semW;
        int *p, i;

        ec_null( semR = SimpleSemOpen("shmexsemR") )
        ec_null( semW = SimpleSemOpen("shmexsemW") )
        ec_null( p = getaddr() )
        *p = 0;
        ec_false( SimpleSemPost(semW) )
        for (i = 1; i < 4; i++) {
            ec_false( SimpleSemWait(semW) )
            *p = i;
            ec_false( SimpleSemPost(semR) )
            sleep(1);
        }
        ec_false( SimpleSemWait(semW) )
        *p = 99;
```

```
            ec_false( SimpleSemPost(semR) )
            ec_false( SimpleSemClose(semR) )
            ec_false( SimpleSemClose(semW) )
        }
        exit(EXIT_SUCCESS);

EC_CLEANUP_BGN
        exit(EXIT_FAILURE);
EC_CLEANUP_END
}
```

Note that the child no longer uses `prev` to tell when the shared memory changed because now it never gets `semR` unless there's a change. This is way more efficient!

7.13.3 System V Shared-Memory Implementation of SMI

Now let's extend the ideas from the previous section and use shared memory and semaphores to implement the SMI functions that we already implemented with FIFOs (Section 7.3.3) and messages (Secs. 7.5.3 and 7.7.2).

The server and each client will use a separate shared-memory segment for incoming messages, along with two semaphores, as I explained in the previous example. So, if there are two clients, there will be three shared-memory segments and six semaphores. With this approach, there isn't a queue of messages—each segment holds one message at a time, and a new one can't be written until the recipient has finished with it and posts the writing semaphore (*W*). This is not the best design because each client can only work as fast as the server can service it, but it will suffice to show how shared memory is used, which is our objective.

Figure 7.3 shows the server and two clients. Shared-memory segment *mem-server* is shared by all three; segment *mem-1* is shared by client1 and the server; segment *mem-2* is shared by client2 and the server. Each of the three processes can access messages located in shared memory without them ever moving.

In the presentation of the SMI implementation for System V shared memory, we're going to assume that you're familiar with the FIFO and message-queue implementations from earlier sections in this chapter, so we won't explain details that have already been explained.

Recall that the SMI functions were specifically designed to allow messages to be processed in-place; the send and receive operations are divided into "getaddr" and "release" functions. The benefit of that wasn't so apparent in the earlier implementations, but it will be in this one.

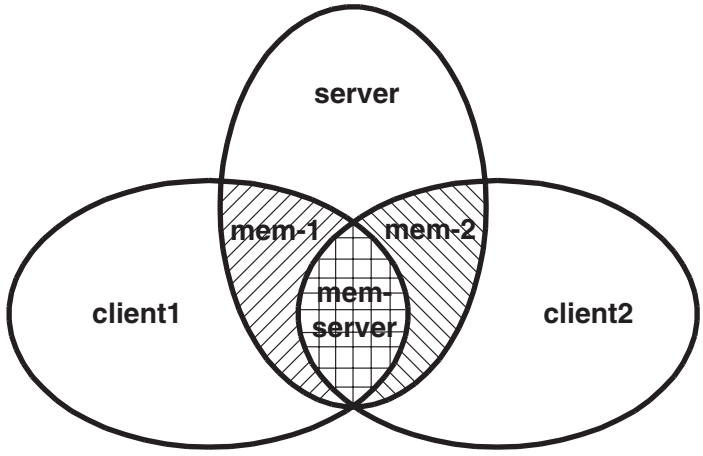


Figure 7.3 Server sharing memory with two clients.

We'll take advantage of one of the features of System V semaphores and use one semaphore set of two semaphores for each shared-memory segment. Semaphore 0 is the read semaphore, and 1 is the write semaphore. Here are some macros for those numbers and for the *semwait* and *sempost* operations on them:

```
#define SEMI_READ    0
#define SEMI_WRITE   1
#define SEMI_POST    1
#define SEMI_WAIT    -1
```

Given a semaphore, the function `op_semi` operates on it. For example, to post the write semaphore for a segment, you execute:

```
ec_neg1( op_semi(semid_receiver, SEMI_WRITE, SEMI_POST) )
```

Here's the code for `op_semi`:

```
static int op_semi(int semid, int sem_num, int sem_op)
{
    struct sembuf sbuf;
    int r;

    sbuf.sem_num = sem_num;
    sbuf.sem_op = sem_op;
    sbuf.sem_flg = 0;
    ec_neg1( r = semop(semid, &sbuf, 1) )
    return r;
```

```
EC_CLEANUP_BGN
    return -1;
EC_CLEANUP_END
}
```

There's also a handy function to initialize a semaphore:

```
static int init_semi(int semid)
{
    union semun arg;
    int r;

    arg.val = 0;
    semctl(semid, SEMI_WRITE, SETVAL, arg);
    semctl(semid, SEMI_READ, SETVAL, arg);
    /* Following call will set otime, allowing clients to proceed. */
    ec_neg1( r = op_semi(semid, SEMI_WRITE, SEMI_POST) )
    return r;

EC_CLEANUP_BGN
    return -1;
EC_CLEANUP_END
}
```

This is the internal data structure behind the `SMIQ` type that the SMI uses:

```
typedef struct {
    SMIENTITY sq_entity;          /* entity */
    int sq_semid_server;          /* server sem */
    int sq_semid_client;          /* client sem (client only) */
    int sq_shmid_server;          /* server shm ID */
    int sq_shmid_client;          /* client shm ID (client only) */
    struct smi_msg *msg_server;   /* ptr to server shm */
    struct smi_msg *msg_client;   /* ptr to client shm (client only) */
    char sq_name[SERVER_NAME_MAX];/* server name */
    struct client_id sq_client;   /* client identification (server only) */
} SMIQ_SHM;
```

A client uses almost the whole structure, but the server just uses some of the members. A client stores:

- Its entity (`SMI_CLIENT`) and the server name
- The semaphore-set IDs (2 semaphores each) for itself and the server
- The shared-memory segment IDs for itself and the server
- Pointers (from `shmat`) to its and the server's segments

The server stores:

- Its entity (`SMI_SERVER`) and name.
- Its semaphore-set ID (2 semaphores).
- Its shared-memory segment ID.
- A pointer to its segment.
- The `client_id` passed to `smi_send_getaddr`, so it can use it in the following `smi_send_release`. This is how it knows which client to send to. For this use, and in messages, the `c_id1` member of the `client_id` structure is the shared-memory identifier, and the `c_id2` member is the semaphore-set identifier. (See the code for `smi_send_getaddr_shm`, below, to see where these members are set.)

The server doesn't store any client's semaphore-set or shared-memory information because there are a lot of clients. This information can be readily passed in an incoming message from a client, similar to the way a message-queue ID was passed in the message in the System V message-queue SMI implementation (Section 7.5.3). We'll see the details soon.

Given this description of how the structure is used, the code for `smi_open_shm` should be understandable:

```
SMIQ *smi_open_shm(const char *name, SMIENTITY entity, size_t msgsize)
{
    SMIQ_SHM *p = NULL;
    char shmname[FILENAME_MAX];
    int i;
    key_t key;

    ec_null( p = calloc(1, sizeof(SMIQ_SHM)) )
    p->sq_entity = entity;
    if (strlen(name) >= SERVER_NAME_MAX) {
        errno = ENAMETOOLONG;
        EC_FAIL
    }
    strcpy(p->sq_name, name);
    mkshm_name_server(p, shmname, sizeof(shmname));
    (void)close(open(shmname, O_WRONLY | O_CREAT, 0));
    ec_neg1( key = ftok(shmname, 1) )
    if (p->sq_entity == SMI_SERVER) {
        if ((p->sq_semid_server = semget(key, 2, PERM_FILE)) != -1)
            (void)shmctl(p->sq_semid_server, IPC_RMID, NULL);
        ec_neg1( p->sq_semid_server = semget(key, 2,
          PERM_FILE | IPC_CREAT) )
        p->sq_semid_client = -1;
        if ((p->sq_shmid_server = shmget(key, 0, PERM_FILE)) != -1)
            (void)shmctl(p->sq_shmid_server, IPC_RMID, NULL);
```

```
        ec_neg1( p->sq_shmid_server = shmget(key, msgsize,
          PERM_FILE | IPC_CREAT) )
        p->sq_shmid_client = -1;
        ec_neg1( init_semi(p->sq_semid_server) )
    }
    else {
        ec_neg1( p->sq_semid_server = semget(key, 2, PERM_FILE) )
        ec_neg1( p->sq_semid_client = semget(IPC_PRIVATE, 2,
          PERM_FILE | IPC_CREAT) )
        ec_neg1( p->sq_shmid_server = shmget(key, msgsize, PERM_FILE) )
        ec_neg1( p->sq_shmid_client = shmget(IPC_PRIVATE, msgsize,
          PERM_FILE | IPC_CREAT) )
        ec_neg1( p->msg_client = shmat(p->sq_shmid_client, NULL, 0) )
        ec_neg1( init_semi(p->sq_semid_client) )
        for (i = 0; !smi_client_nowait && i < 10; i++) {
            union semun arg;
            struct semid_ds ds;

            arg.buf = &ds;
            ec_neg1( semctl(p->sq_semid_server, SEMI_WRITE, IPC_STAT,
              arg) )
            if (ds.sem_otime > 0)
                break;
            sleep(1);
        }
    }
    ec_neg1( p->msg_server = shmat(p->sq_shmid_server, NULL, 0) )
    return (SMIQ *)p;

EC_CLEANUP_BGN
    free(p);
    return NULL;
EC_CLEANUP_END
}
```

The first part, through the call to `ftok`, is almost identical to the code in the version of this function for System V message queues in Section 7.5.3. Then, for the server, it removes an old semaphore set, if there is one, and creates a new one, and the same for a shared-memory segment. The client creates a private semaphore set and shared-memory segment. Both attach the server's shared-memory segment.

Note that `init_semi` (shown earlier), which posts the writing semaphore (to allow sends to proceed), also sets the operation-time, which is how another process can tell that the semaphore has been initialized, as explained in Section 7.9.1. We showed a waiting algorithm in the implement of `SimpleSem` in that section; the algorithm here is a little more elaborate:

- We try at most 10 times because FreeBSD and Darwin systems don't set the time, and we don't want to get stuck.
- There's an under-the-covers global variable `smi_client_nowait` that can be used to prevent any waiting. It's used during timing tests when we know that the server started well in front of any clients. We used it in the program that developed the data for the comparison table at the end of this chapter.

`smi_close_shm` is pretty simple:

```
bool smi_close_shm(SMIQ *sqp)
{
    SMIQ_SHM *p = (SMIQ_SHM *)sqp;

    if (p->sq_entity == SMI_SERVER) {
        char shmname[FILENAME_MAX];

        (void)getaddr(-1);
        ec_neg1( semctl(p->sq_semid_server, 0, IPC_RMID) )
        (void)shmdt(p->msg_server);
        (void)shmctl(p->sq_shmid_server, IPC_RMID, NULL);
        mkshm_name_server(p, shmname, sizeof(shmname));
        (void)unlink(shmname);
    }
    else {
        ec_neg1( semctl(p->sq_semid_client, 0, IPC_RMID) )
        (void)shmdt(p->msg_server);
        (void)shmdt(p->msg_client);
        (void)shmctl(p->sq_shmid_client, IPC_RMID, NULL);
    }
    free(p);
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

Now we're ready for `smi_send_getaddr_shm`:

```
bool smi_send_getaddr_shm(SMIQ *sqp, struct client_id *client,
  void **addr)
{
    SMIQ_SHM *p = (SMIQ_SHM *)sqp;
    int semid_receiver;

    if (p->sq_entity == SMI_SERVER) {
        semid_receiver = client->c_id2;
        p->sq_client = *client;
    }
```

```
    else
        semid_receiver = p->sq_semid_server;
    ec_neg1( op_semi(semid_receiver, SEMI_WRITE, SEMI_WAIT) )
    if (p->sq_entity == SMI_SERVER)
        ec_null( *addr = getaddr(client->c_id1) )
    else {
        *addr = p->msg_server;
        ((struct smi_msg *)*addr)->smi_client.c_id1 = p->sq_shmid_client;
        ((struct smi_msg *)*addr)->smi_client.c_id2 = p->sq_semid_client;
    }
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

For sends from the server, the `client` argument must point to client identification; for sends from the client, it's `NULL`. Let's first track the code for the server and then for a client.

For the server, `semid_receiver` is set from the `c_id2` member, and the whole `client_id` structure is saved for later use by `smi_send_release`, as I mentioned earlier. Then we wait on the `SEMI_WRITE` semaphore. It's been posted in `init_semi`, so at the start we proceed immediately. The message itself is in the shared-memory segment given by `client->c_id1`, and we call `getaddr` to get its address, which is what's returned through the `addr` argument. We'll see `getaddr` shortly; for now, think of it as just doing a `shmat`.

For a client, it's actually simpler, as the client already has all it needs to access its and the server's shared-memory segments and semaphores. It sets `semid_receiver` directly from the `SMIQ_SHM` structure, waits on the `SEMI_WRITE` semaphore, sets the returned address, and stores identifiers in the message for its shared-memory segment and semaphore set. That's how the receiver (the server) will know who sent the message and how to respond.

Once `smi_send_getaddr_shm` returns, its caller is free to use the returned address of the message, and that memory is locked from further writing by any other client. (Somewhat inefficient, as I noted earlier—a pool of incoming server messages would be better, but much more complicated to implement.)

When the caller is done, it calls `smi_send_release`:

```
bool smi_send_release_shm(SMIQ *sqp)
{
    SMIQ_SHM *p = (SMIQ_SHM *)sqp;
    int semid_receiver;

    if (p->sq_entity == SMI_SERVER)
        semid_receiver = p->sq_client.c_id2;
    else
        semid_receiver = p->sq_semid_server;
    ec_neg1( op_semi(semid_receiver, SEMI_READ, SEMI_POST) )
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

All this function needs to do is post the `SEMI_READ` semaphore, but to do that it needs the semaphore-set identifier. For the server, it's in the `client_id` that `smi_send_getaddr_shm` stored in the `SMIQ_SHM` structure; for the client we had it in that structure from the start.

At this point you should be able to follow `smi_receive_getaddr_shm` pretty easily:

```
bool smi_receive_getaddr_shm(SMIQ *sqp, void **addr)
{
    SMIQ_SHM *p = (SMIQ_SHM *)sqp;
    int semid_receiver;

    if (p->sq_entity == SMI_SERVER)
        semid_receiver = p->sq_semid_server;
    else
        semid_receiver = p->sq_semid_client;
    ec_neg1( op_semi(semid_receiver, SEMI_READ, SEMI_WAIT) )
    if (p->sq_entity == SMI_SERVER)
        *addr = p->msg_server;
    else
        *addr = p->msg_client;
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

The corresponding function `smi_receive_release_shm` has to post the

SEMI_WRITE semaphore:

```
bool smi_receive_release_shm(SMIQ *sqp)
{
    SMIQ_SHM *p = (SMIQ_SHM *)sqp;
    int semid_receiver;

    if (p->sq_entity == SMI_SERVER)
        semid_receiver = p->sq_semid_server;
    else
        semid_receiver = p->sq_semid_client;
    ec_neg1( op_semi(semid_receiver, SEMI_WRITE, SEMI_POST) )
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

That's it. Lots of horsing around with semaphores, but very little with the shared memory. If the messages are large (100,000 bytes, say), this is a huge win. Message queues (either System V or POSIX) probably couldn't even handle messages of that size, and, even if they could, the copying from user space to the kernel and back on each message would really slow things down. The speed of the shared-memory SMI implementation, by contrast, is independent of the message size.

7.13.4 System V Shared Memory Critiqued

Not much to dislike about System V shared memory. The system calls are reasonably straightforward, efficient, and usually very well implemented. The hard part is synchronization, which is also the hard part about using threads, and for exactly the same reason: Once things are shared, the program runs fast, but possibly incorrectly, and the work to make it right is exceedingly difficult to test. You have to prove it's correct and then implement it flawlessly.[13]

7.14 POSIX Shared Memory

This section explains the POSIX shared-memory system calls and shows an SMI implementation using POSIX shared memory and POSIX semaphores.

13. About 30 years ago, a colleague of mine said that if a program didn't have to be correct, he could make it arbitrarily fast—he would just change it to print zero and throw away all the other code. So, if you can't guarantee that your use of shared memory or threads is correct, don't use those features—something slower but correct would be more efficient.

7.14.1 POSIX Shared-Memory System Calls

POSIX shared memory involves opening, and perhaps creating, a shared-memory segment with a POSIX IPC name, with all of the entanglements described in Section 7.6.2. The call, `shm_open`, returns a file descriptor, as though a file had been opened. Indeed, POSIX shared-memory segments act like in-memory files. Once opened, you set the size with `ftruncate`, which we saw way back in Section 2.17, as though you're setting the size of a file, which you are. Then you map the segment to your address space with `mmap`, which can map files in general, not just those that are shared-memory segments. In other words, only `shm_open` and `shm_unlink` are specialized for POSIX shared memory. The other calls work on any regular files.

Here are the shared-memory-specific calls:

shm_open—open shared-memory object

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

int shm_open(
    const char *name,      /* POSIX IPC name */
    int flags,             /* flags */
    mode_t perms           /* permissions */
);
/* Returns file descriptor or -1 on error (sets errno) */
```

shm_unlink—remove shared-memory object

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

int shm_unlink(
    const char *name       /* POSIX IPC name */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

The flags for `shm_open` are the ones we've already seen for other file-oriented system calls: `O_CREAT`, `O_EXCL`, `O_TRUNC`, `O_RDONLY`, and `O_RDWR`. You can't use `O_WRONLY`. If the object is created, the third argument establishes its permissions.

As with other POSIX and System V IPC objects, the contents of a POSIX shared-memory object persist at least until the system is rebooted.

You close the file descriptor when you're done with it using `close`, just as for any other file descriptor.

[SUS2002] doesn't say that you can actually do I/O on the shared-memory object, using, say, `read` and `write`, but it's possible for an implementation to allow that. This would be a way of using an in-memory file like any other file. The whole point of having an in-memory file, however, is so you can use ordinary C or C++ operations on it, not go through the expense of an I/O system call, so, even if this feature were supported, it wouldn't be very useful.

Usually, after creating a shared-memory object you set its size with `ftruncate`, as its initial size is zero bytes. Here's a recap of `ftruncate` from Section 2.17:

ftruncate—truncate or stretch a file by file descriptor

```
#include <unistd.h>

int ftruncate(
    int fd,             /* file descriptor */
    off_t length        /* new length */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

Next, you map the object to your address space (analogously to using `shmat` on System V shared-memory objects):

mmap—map pages of memory

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

void *mmap(
    void *addr,         /* desired address or NULL */
    size_t len,         /* length of segment */
    int prot,           /* protection (see below) */
    int flags,          /* flags */
    int fd,             /* file descriptor */
    off_t off           /* offset in file or shared-memory object */
);
/* Returns pointer to segment or MAP_FAILED on error (sets errno) */
```

The first argument, `addr`, is an address near where the segment should be mapped, or exactly where it should be mapped if `MAP_FIXED` is set in the `flags` argument. Otherwise, it's `NULL`. This means you'll take any address, which is the most common case. We'll do it that way in our examples.

The part of the object to be mapped starts at `off` within the object and extends for `len` bytes. It's not necessary to map the whole object (whose size was set with

`ftruncate`) at once, although that's what we'll do in our examples, so `off` will be zero and `len` will be the same as the argument to `ftruncate`.

The `prot` argument is either `PROT_NONE`, meaning that the memory can't be accessed at all, or one or more of the following flags ORed together:

`PROT_READ` Data can be read.

`PROT_WRITE` Data can be written.

`PROT_EXEC` Data can be executed (may be unsupported).

For our purposes, we're going to use `PROT_READ | PROT_WRITE`. Also, the only flag for the flags argument we'll use is `MAP_SHARED`, which means that all changes to the segment are immediately visible. The alternative is `MAP_PRIVATE`, which means changes are private to the process that did the `mmap`. `MAP_PRIVATE` with shared memory doesn't make much sense.

`mmap` is another one of those pointer-returning functions that has a special symbol, `MAP_FAILED`, in this case, for the error return. Make sure you test against it and not `NULL`.

So, all of the above means that to map all of a shared-memory object for reading and writing we'll do this:

```
ec_neg1( ftruncate(fd, len) )
mem = mmap(NULL, len, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
ec_cmp(mem, MAP_FAILED)
```

When you're finished with a mapped segment, you unmap it:

munmap—unmap pages of memory

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

int munmap(
    void *addr,        /* pointer to segment */
    size_t len         /* length of segment */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

You don't have to `unmap` all of what was mapped, but we will in our examples. Thus, `addr` will be what `mmap` returned and `len` is the same as what was passed into `mmap`.

7.14.2 POSIX Shared-Memory Implementation of SMI

The POSIX shared-memory implementation of the SMI functions is very similar to the System V shared-memory implementation (from Section 7.13.3, which you should review before proceeding), except that instead of distinct semaphore objects, we're going to take advantage of a POSIX semaphore feature and use in-memory semaphores. Since they have to be shared between server and client, we'll put them in the shared-memory segment.

To be more precise, each shared memory segment (one for the server and one for each client) has this layout:

```
struct shared_mem {
    sem_t sm_sem_w;
    sem_t sm_sem_r;
    struct smi_msg sm_msg; /* variable size -- must be last */
};
```

The data part of the `smi_msg` extends to the end of the segment, the size of which is calculated from what's passed to `smi_open_pshm` using this macro

```
#define MEM_SIZE(s)\
  (sizeof(struct shared_mem) - sizeof(struct smi_msg) + (s))
```

where `s` is the third argument to `smi_open_pshm`.

```
#define SEMI_READ       0
```

Given a pointer to a `struct shared_mem`, in shared memory, here's a handy function that performs a *semwait*, *sempost*, or destroy operation:

```
#define SEMI_WRITE      1
#define SEMI_DESTROY    2
#define SEMI_POST       1
#define SEMI_WAIT       -1

static int op_semi(struct shared_mem *m, int sem_num, int sem_op)
{
    sem_t *sem_p = NULL;

    if (sem_num == SEMI_WRITE)
        sem_p = &m->sm_sem_w;
    else
        sem_p = &m->sm_sem_r;
    switch (sem_op) {
    case SEMI_WAIT:
        ec_neg1( sem_wait(sem_p) )
        break;
```

```
    case SEMI_POST:
        ec_neg1( sem_post(sem_p) )
        break;
    case SEMI_DESTROY:
        ec_neg1( sem_destroy(sem_p) )
    }
    return 0;

EC_CLEANUP_BGN
    return -1;
EC_CLEANUP_END
}
```

So, if `m` is such a pointer, we can make calls like these:

```
ec_neg1( op_semi(m, SEMI_READ, SEMI_POST) )
ec_neg1( op_semi(m, SEMI_WRITE, SEMI_WAIT) )
```

The POSIX calls don't allow a client to just pass an identifier for a shared-memory segment along with a message to the server. Instead, we have to use an approach similar to what we used with FIFOs back in Section 7.3.3: A client passes its process ID, and the server uses that to form the POSIX name for the object and then opens it and maps it. This is expensive, so the server does this only once per client, looking up an already-mapped segment in a table. All of that and more is stored in the `SMIQ_PSHM` structure, which takes on what by now should be a familiar shape:

```
#define MAX_CLIENTS 50

typedef struct {
    SMIENTITY sq_entity;            /* entity */
    char sq_name[SERVER_NAME_MAX];  /* server name */
    int sq_srv_fd;                  /* server shm file descriptor */
    struct shared_mem *sq_srv_mem;  /* server mapped shm segment */
    struct client {
        pid_t cl_pid;               /* client process ID */
        int cl_fd;                  /* client shm file descriptor */
        struct shared_mem *cl_mem;  /* client mapped shm segment */
    } sq_clients[MAX_CLIENTS];      /* client uses only [0] */
    struct client_id sq_client;     /* client id (server only) */
    size_t sq_msgsize;              /* message size */
} SMIQ_PSHM;
```

The server can keep track of 50 clients. Missing entirely from our implementation is a way for a client to tell the server that it's finished so the server can re-use its slot.

Now we're ready to see how the `SMIQ_PSHM` structure is populated by `smi_open_pshm`.

```
SMIQ *smi_open_pshm(const char *name, SMIENTITY entity, size_t msgsize)
{
    SMIQ_PSHM *p = NULL;
    char shmname[SERVER_NAME_MAX + 50];

    ec_null( p = calloc(1, sizeof(SMIQ_PSHM)) )
    p->sq_entity = entity;
    p->sq_msgsize = msgsize;
    if (strlen(name) >= SERVER_NAME_MAX) {
        errno = ENAMETOOLONG;
        EC_FAIL
    }
    strcpy(p->sq_name, name);
    mkshm_name_server(p, shmname, sizeof(shmname));
    if (p->sq_entity == SMI_SERVER) {
        if ((p->sq_srv_fd = shm_open(shmname, O_RDWR, PERM_FILE)) != -1) {
            (void)shm_unlink(shmname);
            (void)close(p->sq_srv_fd);
        }
        ec_neg1( p->sq_srv_fd = shm_open(shmname, O_RDWR | O_CREAT,
          PERM_FILE) )
        ec_neg1( ftruncate(p->sq_srv_fd, MEM_SIZE(msgsize)) )
        p->sq_srv_mem = mmap(NULL, MEM_SIZE(msgsize),
          PROT_READ | PROT_WRITE, MAP_SHARED, p->sq_srv_fd, 0);
        ec_cmp(p->sq_srv_mem, MAP_FAILED)
        ec_neg1( sem_init(&p->sq_srv_mem->sm_sem_w, true, 1) )
        ec_neg1( sem_init(&p->sq_srv_mem->sm_sem_r, true, 0) )
    }
    else {
        ec_neg1( p->sq_srv_fd = shm_open(shmname, O_RDWR, PERM_FILE) )
        p->sq_srv_mem = mmap(NULL, MEM_SIZE(msgsize),
          PROT_READ | PROT_WRITE, MAP_SHARED, p->sq_srv_fd, 0);
        ec_cmp(p->sq_srv_mem, MAP_FAILED)
        mkshm_name_client(getpid(), shmname, sizeof(shmname));
        if ((p->sq_clients[0].cl_fd = shm_open(shmname, O_RDWR, PERM_FILE))
          != -1) {
            (void)shm_unlink(shmname);
            (void)close(p->sq_clients[0].cl_fd);
        }
        ec_neg1( p->sq_clients[0].cl_fd = shm_open(shmname,
          O_RDWR | O_CREAT, PERM_FILE) )
        ec_neg1( ftruncate(p->sq_clients[0].cl_fd, MEM_SIZE(msgsize)) )
        p->sq_clients[0].cl_mem = mmap(NULL, MEM_SIZE(msgsize),
          PROT_READ | PROT_WRITE, MAP_SHARED, p->sq_clients[0].cl_fd, 0);
        ec_cmp(p->sq_clients[0].cl_mem, MAP_FAILED)
        ec_neg1( sem_init(&p->sq_clients[0].cl_mem->sm_sem_w, true,
          1) )
        ec_neg1( sem_init(&p->sq_clients[0].cl_mem->sm_sem_r, true,
          0) )
    }
    return (SMIQ *)p;
```

```
EC_CLEANUP_BGN
    if (p != NULL)
        (void)smi_close_pshm((SMIQ *)p);
    return NULL;
EC_CLEANUP_END
}

static void mkshm_name_server(const SMIQ_PSHM *p, char *shmname,
  size_t shmname_max)
{
    snprintf(shmname, shmname_max, "/smipshm-%s", p->sq_name);
}

static void mkshm_name_client(pid_t pid, char *shmname,
  size_t shmname_max)
{
    snprintf(shmname, shmname_max, "/smipshm-%d", pid);
}
```

The first few lines, up through the call to `mkshm_name_server`, are just like what we've seen before in previous implementations. Then the server creates a fresh shared-memory object (removing the old one), sets its size, and maps it. Once that's done, the semaphores are in memory, and they're initialized.

A client also maps in the server's segment, but it doesn't set its size nor does it initialize the server's semaphores because the server already did that. A client does, however, create, size, and map its own segment, which it uses the first element of the array for, and it initializes its own semaphores.

As I said, the server won't map in a client's segment until it gets a message from that client, since it doesn't know in advance who the clients may be. Here's a function for the server to use (we'll see where just below) to get a mapped-in address to a client's segment, given only the client's process ID, which, as we've seen before, is sent in each message from client to server:

```
static struct client *get_client(SMIQ_PSHM *p, pid_t pid)
{
    int i, avail = -1;
    char shmname[SERVER_NAME_MAX + 50];

    for (i = 0; i < MAX_CLIENTS; i++) {
        if (p->sq_clients[i].cl_pid == pid)
            return &p->sq_clients[i];
        if (p->sq_clients[i].cl_pid == 0 && avail == -1)
            avail = i;
    }
```

```
    if (avail == -1) {
        errno = EADDRNOTAVAIL;
        EC_FAIL
    }
    p->sq_clients[avail].cl_pid = pid;
    mkshm_name_client(pid, shmname, sizeof(shmname));
    ec_neg1( p->sq_clients[avail].cl_fd = shm_open(shmname, O_RDWR,
      PERM_FILE) )
    p->sq_clients[avail].cl_mem = mmap(NULL, MEM_SIZE(p->sq_msgsize),
      PROT_READ | PROT_WRITE, MAP_SHARED, p->sq_clients[avail].cl_fd,
      0);
    ec_cmp(p->sq_clients[avail].cl_mem, MAP_FAILED)
    return &p->sq_clients[avail];

EC_CLEANUP_BGN
    return NULL;
EC_CLEANUP_END
}
```

To see `get_client` in use, let's look next at `smi_send_getaddr_pshm`:

```
bool smi_send_getaddr_pshm(SMIQ *sqp, struct client_id *client,
  void **addr)
{
    SMIQ_PSHM *p = (SMIQ_PSHM *)sqp;
    struct client *cp;
    struct shared_mem *sm;

    if (p->sq_entity == SMI_SERVER) {
        p->sq_client = *client;
        ec_null( cp = get_client(p, client->c_id1) )
        sm = cp->cl_mem;
    }
    else
        sm = p->sq_srv_mem;
    ec_neg1( op_semi(sm, SEMI_WRITE, SEMI_WAIT) )
    if (p->sq_entity == SMI_CLIENT)
        sm->sm_msg.smi_client.c_id1 = getpid();
    *addr = &sm->sm_msg;
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

Recall that the argument `client` is only used when the server sends a message; it got that structure from the message that it previously received. The `c_id1` mem-

ber is the process ID, and that's what's passed to `get_client`. For a client, the address of the server's segment is right in the `SMIQ_PSHM` structure. With the segment address, we wait on the writing semaphore, and, when it's available, the segment is ours to use. A client then stores its process ID in the message, and the address is returned.

The counterpart is `smi_send_release_pshm`:

```
bool smi_send_release_pshm(SMIQ *sqp)
{
    SMIQ_PSHM *p = (SMIQ_PSHM *)sqp;
    struct client *cp;
    struct shared_mem *sm;

    if (p->sq_entity == SMI_SERVER) {
        ec_null( cp = get_client(p, p->sq_client.c_id1) )
        sm = cp->cl_mem;
    }
    else
        sm = p->sq_srv_mem;
    ec_neg1( op_semi(sm, SEMI_READ, SEMI_POST) )
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

This function when called by the server calls `get_client` to get the address of the client's segment using the process ID (`c_id1` member) that `smi_send_getaddr_pshm` saved. The client's segment was already mapped by `smi_send_getaddr_pshm`—we just need to look it up. For a client, as before, the server's segment is right in the `SMIQ_PSHM` structure. Once we have the segment, the only work we need to do is post the reading semaphore, which is in the segment. That will allow a `smi_receive_getaddr_pshm` on the segment to proceed, as we're about to see:

```
bool smi_receive_getaddr_pshm(SMIQ *sqp, void **addr)
{
    SMIQ_PSHM *p = (SMIQ_PSHM *)sqp;
    struct shared_mem *sm;

    if (p->sq_entity == SMI_SERVER)
        sm = p->sq_srv_mem;
```

```
    else
        sm = p->sq_clients[0].cl_mem;
    ec_neg1( op_semi(sm, SEMI_READ, SEMI_WAIT) )
    *addr = &sm->sm_msg;
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

Since the server and a client each receive from their own segments, the address is available and can be passed right to `op_semi`. There's no need for a call to `get_client`. Once the segment is available for reading, its address is returned.

Finally, `smi_receive_release_pshm` posts the writing semaphore for the received segment once the receiver has finished reading it:

```
bool smi_receive_release_pshm(SMIQ *sqp)
{
    SMIQ_PSHM *p = (SMIQ_PSHM *)sqp;
    struct shared_mem *sm;

    if (p->sq_entity == SMI_SERVER)
        sm = p->sq_srv_mem;
    else
        sm = p->sq_clients[0].cl_mem;
    ec_neg1( op_semi(sm, SEMI_WRITE, SEMI_POST) )
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

If you haven't been comparing this implementation with the one for System V shared memory from Section 7.13.3, you should do so now. You'll see that they're very similar but with these differences:

- With the System V calls, a client can pass in each message shared-memory-segment and semaphore-set identifiers that the server can use to access the objects quickly. Therefore, the server doesn't need a table to keep track of clients.

- With the POSIX calls, we used in-memory semaphores that were inside the shared memory, rather than separate semaphore objects. In theory, these should be faster, although whether they are or not depends on the implementation.

7.14.3 POSIX Shared Memory Critiqued

For shared memory, the POSIX and System V interfaces are about equally convenient. Whether one is more efficient than the other depends on the implementation, but it's likely that any implementation that has both would use the same internal mechanisms in the kernel for both.

The key advantage of POSIX shared memory is that the mapping call, `mmap`, works for any regular file, not just an in-memory file opened with `shm_open`. You have a lot of control over which part of the segment is mapped at any one time.

The key disadvantage of POSIX shared memory is that it's not always available. It's not in some versions of Linux, FreeBSD, or Darwin, for example.

7.15 Performance Comparisons

With an implementation of SMI for six different IPC methods (including sockets in Chapter 8), it's easy to construct a test program that compares the times for sending messages of different sizes. For all my tests, I sent 5000 messages between four clients and a server. Then I normalized the numbers for four systems (Solaris, FreeBSD, Darwin, and Linux) by dividing by the FIFO time for 100-byte messages on that system. Otherwise, the results would be misleading since the computer hardware for the four systems varies widely in performance. Table 7.2 shows the results. The two large-sized messages could be sent using only shared memory and POSIX message queues. Also, as of this writing, FreeBSD, Darwin, and Linux don't support POSIX message queues or POSIX shared memory.

Table 7.2 Performance of Different Message-Passing Methods

Method	100-byte msg	2000-byte msg	20,000-byte msg	100,000-byte msg
FIFO	S: 1.00 B: 1.00 D: 1.00 L: 1.00	S: 1.22 B: 1.46 D: 1.29 L: 1.51	too big	too big
System V message queue	S: 0.90 B: 0.62 L: 0.31	S: 1.82 B: 3.76 L: 0.64	too big	too big
POSIX message queue	S: 2.02	S: 2.40	S: 7.03	S: 33.39
System V shared memory	S: 1.47 B: 0.94 D: 1.04 L: 0.55	S: 1.41 B: 0.91 D: 1.07 L: 0.53	S: 1.55 B: 0.90 D: 1.02 L: 0.47	S: 1.24 B: 0.90 D: 1.06 L: 0.51
POSIX shared memory	S: 1.27	S: 1.25	S: 1.41	S: 1.37
Sockets	S: 1.84 B: 0.81 D: 1.04 L: 0.75	S: 2.15 B: 1.00 D: 1.27 L: 0.95	S: 10.15 B: 7.99 D: 5.52 L: 6.06	S: 44.13 B: 35.83 D: 25.86 L: 31.91

S: Solaris; B: FreeBSD; D: Darwin; L: Linux. All times normalized for each system so FIFO time for 100-byte messages is 1.00. Sockets used the `AF_UNIX` domain (see Chapter 8).

Some comments on the results:

- They're not definitive because this was only one possible test, SMI is only one possible interface, and our implementations weren't optimal—they were mainly designed to serve as textbook examples.
- Even with the normalization, a system with a faster time for a given method doesn't necessarily implement that method better. It might implement FIFOs (the normalization divisor) slower.

- For small messages (100 bytes or so), System V message queues performed really well on all the systems, except for Darwin 6.6, which doesn't support them.
- Sockets perform almost as well as the two message-queue methods (even better in a few cases), and handle messages of any size. In addition, they're the only method that goes between machines, including over the Internet, although the timing tests used them only within a single machine.
- On Solaris, POSIX message queues performed less well than System V message queues but had the advantage of handling very large messages.
- As we would expect, the performance of the two shared-memory methods is independent of the message size.

Therefore, if we had to make a sweeping generalization, we'd say that, for maximum performance, use System V message queues for small messages, and use shared memory for big messages. One idea we didn't try combines the two: Use a System V message to tell the server when it has work to do, but use a shared-memory segment to pass the data.

If optimal performance isn't essential and you'd like your life to be simple, use sockets for everything. They're fairly efficient, they handle messages of any size, they're universally supported, and they go between machines.

These recommendations are only for message-oriented IPC. For other uses, one IPC method may be better than another for your purposes.

Exercises

The first six Exercises ask you to implement System V IPC functions with POSIX IPC functions or the other way around. In most cases you won't be able to implement every feature and behavior, so part of the Exercise is to carefully document exactly where your implementation falls short.

7.1. Implement `msgget`, `msgctl`, `msgsnd`, and `msgrcv` with POSIX IPC functions.

7.2. Implememt `mq_open`, `mq_close`, `mq_unlink`, `mq_send`, and `mq_receive` with System V IPC functions.

7.3. Implement `semget`, `semctl`, and `semop` with POSIX IPC functions, allowing only one semaphore per set. Can you handle increments and decrements of more than one?

7.4. Same as the previous Exercise, but with more than one semaphore per set.

7.5. Implement `sem_open`, `sem_close`, `sem_unlink`, `sem_post`, and `sem_wait` with System V IPC functions.

7.6. Implement `shmget`, `shmctl`, `shmat`, and `shmdt` with POSIX IPC functions.

7.7. There's no Exercise asking you to implement `shm_open`, `shm_unlink`, `ftruncate`, `mmap`, and `munmap` with System V IPC functions. Why not? Can you propose a suitable Exercise? Can you complete it?

7.8. Design and conduct an experiment to compare the efficiency of conventional I/O on a regular file using `read` and `write` with access via a memory-mapped file. Include both sequential and random I/O, and perhaps several variations of each.

7.9. Design and conduct an experiment to compare the efficiency of write locks only (using `lockf`) with a combination of read and write locks (using `fcntl`). Try to create a pattern of reads and writes that shows the greatest difference in efficiency.

7.10. Extend the program you wrote in Exercise 5.14 to include process attributes from Appendix A that are explained in this chapter.

8

Networking and Sockets

The IPC mechanisms in Chapters 6 and 7 have their uses, of course, but many modern applications need to send data between machines, not just between processes on the same machine. By "between machines," I don't just mean machines across the room or even in the same building. I mean anywhere in the world, using the Internet.

The basic mechanism for going between machines—networking—is called *sockets*. There are eight basic socket system calls, five of which are unique to sockets: `socket`, `bind`, `listen`, `accept`, `connect`, `read`, `write`, and `close`. However, because sockets, and especially the underlying communication protocols, can get complicated, there are 60 or so other system calls involved, all of which are described in this chapter.

While there's enough here to get you started, along with lots of examples, including a Web browser and server, you'll probably need more advanced references as you get deeper into UNIX networking. The most complete book by far is [Ste2003]. You'll also need to be familiar with your system's documentation and with details of the protocol you're using, especially if it's more exotic than TCP/IP.

Here's how we'll proceed: First, I'll explain the basic socket-related system calls and give some simple client/server examples. Then I'll talk about socket addresses and socket options. I'll introduce a simple interface that hides a lot of the complexity and use it to implement a socket-based version of the Simple Messaging Interface (SMI) that I discussed extensively in Chapter 7. Next come more advanced topics: connectionless sockets, out-of-band data, network database functions, and some other miscellaneous functions. Finally, I'll say a bit about the challenge of building servers that can handle thousands of clients at once.

8.1 Socket Basics

This section introduces sockets and describes the basic socket system calls.

8.1.1 How Sockets Work

Sockets are complicated, so let's start with something we already know. Recall from Section 7.2 that a process can open a FIFO like this:

```
fd = open("MyFifo", O_RDONLY);
```

Now consider what this system call does underneath the covers:

1. It creates an endpoint for I/O and allocates a file descriptor for it.
2. It binds the file descriptor to the external name "MyFifo."
3. It waits until there's a writer.
4. It returns with the file descriptor, which can be used in a `read` system call.

Sockets work similarly, except that each step is broken down into a separate system call:

`socket` Creates an endpoint and allocates a file descriptor.

`bind` Associates the socket with an external name so other processes can refer to it.

`listen` Marks the socket as being able to accept connections from other sockets.

`accept` Blocks waiting for a connection.

`connect` Connects to a socket that is blocked in an `accept`.

FIFOs are symmetrical between client and server in the sense that both execute exactly the same system call, `open`, but with different flags (e.g., `O_RDONLY` vs. `O_WRONLY`). But connected sockets are usually asymmetrical because client and server use a different sequence of system calls, as shown in Figure 8.1.

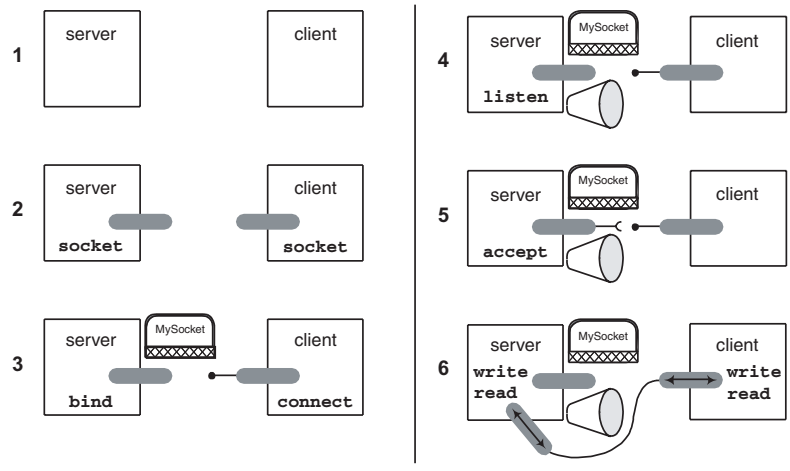


Figure 8.1 Setting Up Connected Sockets.

The server:

- Calls `socket` to create the endpoint and create a file descriptor (Step 2).
- Calls `bind` to bind the socket to a name (Step 3).
- Calls `listen` to mark it as accepting connections (Step 4).
- Calls `accept` to block until a connection is made. Then `accept` creates a second socket, with a new file descriptor (Step 5).
- Uses the new file descriptor to read and write data on the second socket (Step 6).

The client:

- Calls `socket` to create the endpoint and create a file descriptor (Step 2).
- Calls `connect`, with the server's bound name as an argument, to block until the connection is accepted by the server (Step 3, although it need not be synchronized with the server's Step 3).
- Uses the socket's file descriptor to read and write data (Step 6).

Before I formally introduce any socket system calls, let's look at an example program that forks to create a child process that acts as a client, while the parent acts as a server:

```
#define SOCKETNAME "MySocket"

int main(void)
{
    struct sockaddr_un sa;

    (void)unlink(SOCKETNAME);
    strcpy(sa.sun_path, SOCKETNAME);
    sa.sun_family = AF_UNIX;
    if (fork() == 0) { /* child -- client */
        int fd_skt;
        char buf[100];

        ec_neg1( fd_skt = socket(AF_UNIX, SOCK_STREAM, 0) )
        while (connect(fd_skt, (struct sockaddr *)&sa, sizeof(sa)) == -1)
            if (errno == ENOENT) {
                sleep(1);
                continue;
            }
            else
                EC_FAIL
        ec_neg1( write(fd_skt, "Hello!", 7 ) )
        ec_neg1( read(fd_skt, buf, sizeof(buf)) )
        printf("Client got \"%s\"\n", buf);
        ec_neg1( close(fd_skt) )
        exit(EXIT_SUCCESS);
    }
    else { /* parent -- server */
        int fd_skt, fd_client;
        char buf[100];

        ec_neg1( fd_skt = socket(AF_UNIX, SOCK_STREAM, 0) )
        ec_neg1( bind(fd_skt, (struct sockaddr *)&sa, sizeof(sa)) )
        ec_neg1( listen(fd_skt, SOMAXCONN) )
        ec_neg1( fd_client = accept(fd_skt, NULL, 0) )
        ec_neg1( read(fd_client, buf, sizeof(buf)) )
        printf("Server got \"%s\"\n", buf);
        ec_neg1( write(fd_client, "Goodbye!", 9 ) )
        ec_neg1( close(fd_skt) )
        ec_neg1( close(fd_client) )
        exit(EXIT_SUCCESS);
    }

EC_CLEANUP_BGN
        exit(EXIT_FAILURE);
EC_CLEANUP_END
}
```

The first thing we see is that the name passed to `bind` and `connect` isn't just a string, but a `sockaddr` structure that contains the string in the `sun_path` mem-

ber and an address family in the `sun_family` member. Socket addresses are a huge subject in themselves, which I'll get to, but for now just assume that `AF_UNIX` means "local communication within a single machine," which is all this example needs.

The socket file name is unlinked because, for `AF_UNIX`, `bind` won't reuse an existing name, and we want to ensure that it starts with a fresh one.

The server (parent process) follows the six-step sequence in Figure 8.1. (Ignore the mysterious arguments to `listen` and `accept` for now.) The client (child process) also follows the steps in Figure 8.1 but with a twist: If it gets to its `connect` before the server gets to its `bind`, `connect` will fail because the socket file isn't there yet. In that case we sleep and try again.

When the server returns from `accept`, it proceeds to read and write the file descriptor it got from `accept` (*not* the original socket's file descriptor). The client writes and reads its socket file descriptor (the only one it has) when it gets a return from `connect`. Remember that `accept` and `connect` are the two calls that block—the others just do their work and return right away.

The connection stays up as long as the server and client want it to, acting like a bidirectional pipe (Section 6.6).

The socket file really is a file of that type, as shown by the ls command:

```
$ ls -l MySocket
srwxr-xr-x   1 marc     sysadmin       0 Apr  4 10:03 MySocket
```

Here's the output I got when I ran the example program:

```
Server got "Hello!"
Client got "Goodbye!"
```

So, sockets are really pretty simple, except for some details about:

- Handling multiple clients from the same server
- Address families, including the interesting `AF_INET` family, for network communication
- Nonstream-oriented communication, involving datagrams instead of just streams of data
- Connectionless communication, in which processes send datagrams to named receivers, instead of setting up a semipermanent connection, as in my example

- Lots of options that fine-tune the behavior of the socket, especially for `AF_INET` and `AF_INET6`
- Passing binary data between machines with different ways of storing numbers
- Accessing databases of names (e.g., *www.basepath.com/aup*)

Well, actually, these aren't mere details. They're important topics that you need to know about to be able to build networking applications, and they fill the rest of this chapter.

8.1.2 Basic System Calls For Connected Sockets

As I indicated, there are connected sockets, in which a channel is set up by `accept` and `connect`, and connectionless sockets, in which datagrams are sent to named receivers (`listen` and `accept` aren't used, and `connect` is used in a different way). I'll confine my discussion in this section to connected sockets only, and defer connectionless sockets to Section 8.6.

This section covers just the basic five socket-specific system calls which I already informally introduced, starting with `socket`:

socket—create endpoint for communication

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

int socket(
    int domain,            /* domain (AF_UNIX, AF_INET, etc.) */
    int type,              /* SOCK_STREAM, SOCK_DGRAM, etc. */
    int protocol           /* specific to type; usually zero */
);
/* Returns file descriptor or -1 on error (sets errno) */
```

The file descriptor returned from `socket` can be used for two different purposes:

- By the server, as an endpoint for accepting connections with `accept` (The actual I/O is done on the file descriptor returned from `accept`.)
- By a client, for direct I/O once `connect` has successfully connected the socket

The `domain`[1] and `type` are the same as for the socket address that will be used in a following call to `bind` or `connect`. The `protocol` argument is for making a

1. The macros starting with `AF_` are sometimes shown as starting with `PF_`, but the standard uses the `AF_` notation, which is what you should use.

selection if the type offers a choice, but in most cases the default will do, so it's zero.

Next, the server has to name the socket with `bind`:

bind—bind name to socket

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

int bind(
     int socket_fd,             /* socket file descriptor */
     const struct sockaddr *sa, /* socket address */
     socklen_t sa_len           /* address length */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

Socket addresses get a whole section to themselves (Section 8.2). We've already seen an example for `AF_UNIX`, which is just for communication within a single machine, as were the IPC mechanisms in Chapter 7. Later we'll see how to set up addresses for other domains, including `AF_INET`.

Remember that on most systems, for `AF_UNIX`, `bind` will make a fresh socket file; it can't reuse an existing file.

The server also has to call `listen` to set the socket up for accepting connections:

listen—mark socket for accepting and set queue limit

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

int listen(
     int socket_fd,             /* socket file descriptor */
     int backlog                /* maximum connection queue length */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

Our example in the last section didn't show it, but a server can accept connections from lots of clients. Also, as it can only issue `accept`s at a certain rate, requests for connections can queue up. The second argument to `listen` limits the length of that queue. Usually an attempt by a client to `connect` when the queue is full causes its `connect` to return –1 with `errno` set to `ECONNREFUSED`. In my examples I use the constant `SOMAXCONN`, which is a system-defined maximum.

Next, the server accepts a connection:

accept—accept new connection on socket and create new socket

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

int accept(
     int socket_fd,                 /* socket file descriptor */
     struct sockaddr *sa,           /* socket address or NULL */
     socklen_t *sa_len              /* address length */
);
/* Returns file descriptor or -1 on error (sets errno) */
```

Normally, `accept` blocks until a connection request arrives (from some other process's call to `connect`), and then it creates a new socket for the I/O on that connection and returns a new file descriptor. That file descriptor can be used with ordinary `read` and `write` system calls, although for more control over the I/O you can also use socket-specific calls—`send`, `sendto`, `sendmsg`, `recv`, `recvfrom`, and `recvmsg`—which I'll discuss in Sections 8.6.2, 8.6.3, and 8.9.1.

If the `O_NONBLOCK` flag is set for the socket file descriptor (with `fcntl`; see Section 3.8.3), `accept` doesn't wait for a connection but returns –1 immediately if no request is on the queue, with `errno` set to `EAGAIN` or `EWOULDBLOCK`.

The socket file descriptor can be used with `select` or `poll`, and it usually is, as we'll see in Section 8.1.3. Typically, the server sets up the `fd_set` for `select` (or the equivalent for `poll`) with the socket file descriptor and each file descriptor returned by `accept`. If `select` or `poll` says the socket file descriptor is ready, that means the server should do an `accept` (which won't block). If one of the others is ready, that means that data has arrived from a client and should be read.

If the `sa` argument is non-`NULL`, it's used to return the socket address that connected. You have to set `sa_len` on input to the size of the storage that `sa` points to, and on return it's set to the size of the actual address.

So much for the server. The client, after creating its socket, just calls `connect`, with the same socket address that the server used for `bind`:

connect—connect socket

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

int connect(
     int socket_fd,                 /* socket file descriptor */
     const struct sockaddr *sa,     /* socket address */
     socklen_t sa_len               /* address length */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

Like `accept`, `connect` usually blocks until the connection request is accepted, but it doesn't return a file descriptor—the client does I/O on the socket file descriptor.

If the `O_NONBLOCK` flag is set, `connect` won't block waiting for a connection but will return –1 with `errno` set to `EINPROGRESS`. The connection request isn't abandoned but remains on the queue to be eventually satisfied. Further calls to `connect` during that period also return –1 but with `errno` now set to `EALREADY`. When the connection is made, the socket file descriptor can be used. If you want, you can use `select` or `poll` to wait for it to be ready for writing (not reading). A typical use for this would be an application that has some initialization to do. It issues a nonblocking `connect`, does the initialization, and then issues a `select` or `poll` to block.

8.1.3 Handling Multiple Clients

Now let's extend the example from Section 8.1.1 so the server can handle multiple clients. Here's a replacement for the server-side code in that example:

```
static bool run_server(struct sockaddr_un *sap)
{
    int fd_skt, fd_client, fd_hwm = 0, fd;
    char buf[100];
    fd_set set, read_set;
    ssize_t nread;

    ec_neg1( fd_skt = socket(AF_UNIX, SOCK_STREAM, 0) )
    ec_neg1( bind(fd_skt, (struct sockaddr *)sap, sizeof(*sap)) )
    ec_neg1( listen(fd_skt, SOMAXCONN) )
    if (fd_skt > fd_hwm)
        fd_hwm = fd_skt;
    FD_ZERO(&set);
    FD_SET(fd_skt, &set);
    while (true) {
        read_set = set;
        ec_neg1( select(fd_hwm + 1, &read_set, NULL, NULL, NULL) )
        for (fd = 0; fd <= fd_hwm; fd++)
            if (FD_ISSET(fd, &read_set)) {
                if (fd == fd_skt) {
                    ec_neg1( fd_client = accept(fd_skt, NULL, 0) )
                    FD_SET(fd_client, &set);
                    if (fd_client > fd_hwm)
                        fd_hwm = fd_client;
                }
```

```
                else {
                    ec_neg1( nread = read(fd, buf, sizeof(buf)) )
                    if (nread == 0) {
                        FD_CLR(fd, &set);
                        if (fd == fd_hwm)
                            fd_hwm--;
                        ec_neg1( close(fd) )
                    }
                    else {
                        printf("Server got \"%s\"\n", buf);
                        ec_neg1( write(fd, "Goodbye!", 9 ) )
                    }
                }
            }
    }
    ec_neg1( close(fd_skt) )
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

Most of the new code has to do with setting up for and using `select`. I discussed it in Section 4.2.3, but here's a recap:

select—wait for I/O to be ready

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

int select(
    int nfds,                   /* highest fd + 1 */
    fd_set *readset,            /* read set or NULL */
    fd_set *writeset,           /* write set or NULL */
    fd_set *errorset,           /* error set or NULL */
    struct timeval *timeout     /* time-out (microseconds) or NULL */
);
/* Returns number of bits set or -1 on error (sets errno) */
```

There are two `fd_sets` in the function `run_server`: One, `set`, holds all the file descriptors of interest: the socket file descriptor, and one per client as returned by `accept`. We need the number of file descriptors in the set that `select` should look at, so the variable `fd_hwm` ("high-water-mark") is kept up-to-date as we get new file descriptors from `accept`. When we close the highest file descriptor, we remove it from `set` and decrement the high-water-mark. Removing it from `set` is extremely important; otherwise, `select` would keep reporting it as ready, meaning not that data is there to be read, but only that `read` will not block.

The second `fd_set`, `read_set`, is copied from `set` each time we call `select` and then is modified by `select` to indicate which file descriptors are ready.[2]

The function then loops forever, or until we kill it with a signal. Each time it gets a return from `select` it cycles through the returned set (the modified `read_set` variable) looking for a ready file descriptor. If `fd_skt` is ready, it does an `accept`; if anything else is ready, it means that a client has sent some data. We read it, display it, and send a little message back.

The code for a client is the same as in the earlier example, except that the message to the server includes the process ID:

```
static bool run_client(struct sockaddr_un *sap)
{
    if (fork() == 0) {
        int fd_skt;
        char buf[100];

        ec_neg1( fd_skt = socket(AF_UNIX, SOCK_STREAM, 0) )
        while (connect(fd_skt, (struct sockaddr *)sap, sizeof(*sap)) == -1)
            if (errno == ENOENT) {
                sleep(1);
                continue;
            }
            else
                EC_FAIL
        snprintf(buf, sizeof(buf), "Hello from %ld!",
          (long)getpid());
        ec_neg1( write(fd_skt, buf, strlen(buf) + 1 ) )
        ec_neg1( read(fd_skt, buf, sizeof(buf)) )
        printf("Client got \"%s\"\n", buf);
        ec_neg1( close(fd_skt) )
        exit(EXIT_SUCCESS);
    }
    return true;

EC_CLEANUP_BGN
    /* only child gets here */
    exit(EXIT_FAILURE);
EC_CLEANUP_END
}
```

Finally, there's a small `main` function that sets up the socket address, creates four client children, and runs the server in the parent:

2. A common mistake is to keep just one `fd_set` that gets passed to `select`. Since it clears the bits for unready file descriptors, they never get waited for.

```
#define SOCKETNAME "MySocket"

int main(void)
{
    struct sockaddr_un sa;
    int nclient;

    (void)unlink(SOCKETNAME);
    strcpy(sa.sun_path, SOCKETNAME);
    sa.sun_family = AF_UNIX;
    for (nclient = 1; nclient <= 4; nclient++)
        ec_false( run_client(&sa) )
    ec_false( run_server(&sa) )
    exit(EXIT_SUCCESS);

EC_CLEANUP_BGN
        exit(EXIT_FAILURE);
EC_CLEANUP_END
}
```

Here's the output we got:

```
Server got "Hello from 31786!"
Client got "Goodbye!"
Server got "Hello from 31785!"
Client got "Goodbye!"
Server got "Hello from 31784!"
Client got "Goodbye!"
Server got "Hello from 31787!"
Client got "Goodbye!"
```

We already have almost enough to implement a fairly sophisticated client-server system. What we need to know more about is how to set up socket addresses (`AF_UNIX` isn't that interesting) and how to set socket options. That's coming right up, after we deal with the byte-order issue.

8.1.4 Byte Order

There's never any problem passing strings like "Hello" and "Goodbye" between machines over a network because all communication preserves the order of bytes. However, there can be problems passing binary numbers because how the bytes in a number are arranged does indeed vary between machines.

To see this, look at Figure 8.2, which shows two ways of storing a two-byte integer whose hex representation is 0xD04C at location 106204 in memory.

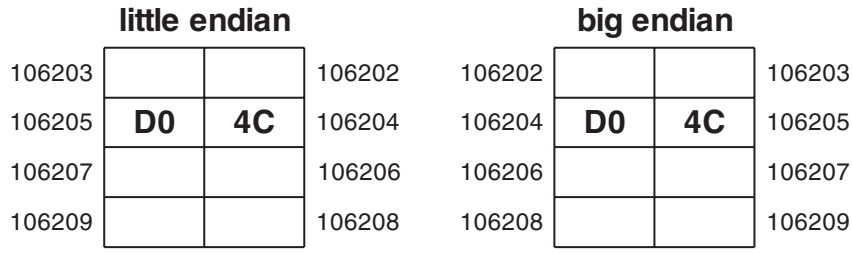


Figure 8.2 Little vs. Big Endian.

In a so-called "little-endian" machine, the address of a number is the address of its lowest-order byte; in a "big-endian" machine, it's the address of its highest-order byte.[3] Both are perfectly valid ways to address numbers, but the problem is that if one machine sends binary numbers to another machine that uses a different byte order, the numbers will be scrambled. That is, if the little-endian machine sends the number as bytes (the only way data is sent), they will go in the sequence 4C followed by D0, and if the receiving machine is big-endian, it will interpret the number as 4CD0 since it takes the first byte received as the high-order byte.

The solution is to send binary numbers in an agreed-upon network byte order. Thus, all senders must convert from host (local) byte order to network, and all receivers must convert from network byte order to host. But this is done only if the data isn't already structured in a standard way. For example, if you had a photograph encoded as a JPEG, you would send it as it stands—you would not attempt to convert any part of the JPEG to network byte order.

For most purposes you don't have to know either your host byte order or network byte order because there's a standard set of translation functions that do the conversion for 16-bit and 32-bit integers:

htons—convert 16-bit value from host to network byte order

```
#include <arpa/inet.h>

uint16_t htons(
    uint16_t hostnum          /* 16-bit number in host byte order */
);
/* Returns number in network byte order (no error return) */
```

3. The terminology is self-explanatory, but the term "big-endian" actually comes from Jonathan Swift's *Gulliver's Travels*, which describes the controversy over whether to break eggs at the bigger or smaller end. In fact, "eleven thousand [Big-Endians] have, at several times, suffered Death, rather than submit to break their Eggs at the smaller End."

htonl—convert 32-bit value from host to network byte order

```
#include <arpa/inet.h>

uint32_t htonl(
     uint32_t hostnum       /* 32-bit number in host byte order */
);
/* Returns number in network byte order (no error return) */
```

ntohs—convert 16-bit value from network to host byte order

```
#include <arpa/inet.h>

uint16_t ntohs(
     uint16_t netnum        /* 16-bit number in network byte order */
);
/* Returns number in host byte order (no error return) */
```

ntohl—convert 32-bit value from network to host byte order

```
#include <arpa/inet.h>

uint32_t ntohl(
     uint32_t netnum        /* 32-bit number in network byte order */
);
/* Returns number in host byte order (no error return) */
```

Unfortunately, the 16-bit functions use the suffix "s" for `short`, even though `short`s may have more than 16 bits, and, similarly, the 32-bit functions use the suffix "l" for `long`. In fact, these functions don't even operate on `short`s and `long`s; they operate on the types `uint16_t`, which is a 16-bit unsigned integer, and `uint32_t`, which is a 32-bit unsigned integer.

Just to show two of these functions in action, here's a program that converts 0xD04C to network byte order, displays the bytes, and converts it back:

```
int main(void)
{
    uint16_t nhost = 0xD04C, nnetwork;
    unsigned char *p;

    p = (unsigned char *)&nhost;
    printf("%x %x\n", *p, *(p + 1));
    nnetwork = htons(nhost);
    p = (unsigned char *)&nnetwork;
    printf("%x %x\n", *p, *(p + 1));
    exit(EXIT_SUCCESS);
}
```

The output shows that the Intel Pentium CPU on which it was run uses a byte order that's different from network:

```
4c d0
d0 4c
```

We can further conclude that the Pentium is little-endian, and network byte order is big-endian.[4]

Generally, you'll use the standard conversion functions when the protocol you're using requires you to send binary numbers, as we'll see in the next section. For your own purposes, try to design your applications to send characters instead of binary data, unless you're sending an object with a standardized format (JPEG, MP3, etc.). That's exactly the way HTTP (the Web protocol) does it (Section 8.4.2).

8.2 Socket Addresses

This whole section is about nothing but how to come up with a socket address (`struct sockaddr`) that you can use in calls to `bind` and `connect`. We already saw how to do this for domain `AF_UNIX`, where the address is mostly just some path name:

```
struct sockaddr_un sa;

strcpy(sa.sun_path, SOCKETNAME);
sa.sun_family = AF_UNIX;
```

But that's the only simple case. The other domains are more complicated.

8.2.1 Socket-Address Structures

Each address family has its own structure and header file. There's `sockaddr_un` for `AF_UNIX`, `sockaddr_in` for `AF_INET`, `sockaddr_x25` for `AF_X25`, and so on, although only the first two domains are standardized in [SUS2002]. One approach, which we've been using for `AF_UNIX`, is to declare a variable for the specific structure you need, fill its members somehow, and cast its address to a pointer-to-`struct sockaddr` (the generic type) when you call `bind` or `connect`.

4. It's big-endian because for years BSD systems, where the socket system calls originated, ran on big-endian VAX 11/780 computers, and network byte order was, in essence, "our byte order."

Speaking portably, the structure `sockaddr` only defines an abstract type; you can't use it to declare a structure to use because it may not have enough space. Instead, there's a standard structure named `sockaddr_storage` that is guaranteed to have enough space, but whose members aren't customized for any particular domain.

Sounds messy, but it's really not so bad:

- If you know exactly what domain you're using, declare a structure of the appropriate type (e.g., `sockaddr_un`) or allocate enough space for that structure dynamically (e.g., with `malloc`).
- If you need space enough for any domain's structure, use `sockaddr_storage` but cast a pointer to the appropriate type before using it.
- Use a cast to `struct sockaddr` in calls to `bind` and `connect`—according to their prototypes, you have no other choice.

Here's an example of the last two rules used together:

```
struct sockaddr_storage sas;
struct sockaddr_un *sa = (struct sockaddr_un *)&sas;
sa->sun_family = AF_UNIX;
...
ec_neg1( bind(fd, (struct sockaddr *)sa, sizeof(*sa)) )
```

Don't worry too much about the last rule. If you get it wrong, the compiler will tell you. Don't worry about using a `sockaddr_storage` directly either—it doesn't have any of the members you'll be wanting to use. But, since you're casting, there's no check at all at compile time that you're using the correct domain-specific structure for your domain. That you have to get right the first time.

8.2.2 `AF_UNIX` Socket Addresses

struct sockaddr_un—structure for AF_UNIX socket addresses

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

struct sockaddr_un {
    sa_family_t sun_family;    /* AF_UNIX */
    char sun_path[X];          /* socket pathname */
};
```

The actual space for the pathname, designated by *X* in the synopsis, varies from system to system. Don't assume it's more than 90 or so bytes. I already explained how to use this structure.

8.2.3 AF_INET Socket Addresses

As I mentioned earlier, the `AF_INET` domain is for communicating with sockets over the Internet. Here's the `sockaddr_in` structure:

struct sockaddr_in—structure for AF_INET socket addresses

```
#include <netinet/in.h>

struct sockaddr_in {
    sa_family_t sin_family;    /* AF_INET */
    in_port_t sin_port;        /* port number (uint16_t) */
    struct in_addr sin_addr;   /* IPv4 address */
};

struct in_addr {
    in_addr_t s_addr;          /* IPv4 address (uint32_t) */
};
```

An IPv4[5] address (usually called just an "IP address") is a 32-bit binary number assigned to a network interface on a machine. Instead of referring to these numbers the usual way (e.g., 1,182,625,240), we use dotted notation, in which each byte is expressed as a separate decimal number, as in 216.109.125.70. Even that isn't very convenient, so there's a system using descriptive names instead that allows us to say *www.yahoo.com* instead of 216.109.125.70 (Section 8.2.5), but I'll use dotted notation for now.

Each IP can be associated with lots of different services, each of which is assigned a 16-bit port number. Many of the port numbers are, by convention, assigned to well-known services. For example, HTTP (Web) servers are on port 80, FTP servers are on port 21, Telnet servers are on port 23, and so on. You can see how the ports are assigned on your machine by looking at the /etc/services file. For example, here are selected parts of the file (over 2000 lines in all) on my FreeBSD system:

```
...
ftp             21/tcp    #File Transfer [Control]
ftp             21/udp    #File Transfer [Control]
```

5. It's Version 4 of the Internet Protocol.

```
ssh              22/tcp    #Secure Shell Login
ssh              22/udp    #Secure Shell Login
telnet           23/tcp
telnet           23/udp
...
finger           79/tcp
finger           79/udp
http             80/tcp    www www-http         #World Wide Web HTTP
http             80/udp    www www-http         #World Wide Web HTTP
...
```

The types for the port and IP number, `in_port_t` and `in_addr_t`, are defined to be `uint16_t` and `uint32_t`, respectively, and must be in network byte order, which means you can use the `htons` and `htonl` functions, like this:

```
struct sockaddr_in sa;

sa.sin_family = AF_INET;
sa.sin_port = htons(80);
sa.sin_addr.s_addr =  htonl((216UL << 24) + (109UL << 16) +
  (125UL << 8) + 70UL); /* 216.109.125.70 */
```

But for dotted IP numbers, there's an even better function that goes directly from a string to a 32-bit IP number in network byte order:

inet_addr—convert dotted IPv4 address string to integer

```
#include <arpa/inet.h>

in_addr_t inet_addr(
     const char *cp             /* dotted IP address */
);
/* Returns IP address or (in_addr_t)-1 on error (errno not defined) */
```

The return value, –1 cast to an unsigned 32-bit number, is the same as what we would get if we converted 255.255.255.255. But this is OK because that's an illegal IP address.

The reverse function is also sometimes handy:

inet_ntoa—convert integer IPv4 address to dotted string

```
#include <arpa/inet.h>

char *inet_ntoa(
     struct in_addr in      /* integer address */
);
/* Returns string (no error return) */
```

Instead of `inet_addr` and `inet_ntoa`, you can use the more general functions `inet_ntop` and `inet_pton` (Section 8.9.5), which also work for IPv6 addresses.

Going back to our example, let's use `inet_addr` instead of `htonl` for the IP address and add some code that connects to the HTTP server on port 80, sends a request for a Web page, and displays part of what comes back:

```
#define REQUEST "GET / HTTP/1.0\r\n\r\n"

int main(void)
{
    struct sockaddr_in sa;
    int fd_skt;
    char buf[1000];
    ssize_t nread;

    sa.sin_family = AF_INET;
    sa.sin_port = htons(80);
    sa.sin_addr.s_addr =  inet_addr("216.109.125.70");
    ec_neg1( fd_skt = socket(AF_INET, SOCK_STREAM, 0) )
    ec_neg1( connect(fd_skt, (struct sockaddr *)&sa, sizeof(sa)) )
    ec_neg1( write(fd_skt, REQUEST, strlen(REQUEST) ) )
    ec_neg1( nread = read(fd_skt, buf, sizeof(buf)) )
    (void)write(STDOUT_FILENO, buf, nread);
    ec_neg1( close(fd_skt) )
    exit(EXIT_SUCCESS);

EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
EC_CLEANUP_END
}
```

The request is a GET command, which is defined by the HTTP protocol that defines Web communication (see Section 8.4.2). Here's part of the output that came back, which you may recognize as the HTML source for Yahoo's home page. It begins with a status line, followed immediately by the HTML (shown greatly abbreviated):

```
HTTP/1.1 200 OK
Date: Sat, 19 Jul 2003 18:51:56 GMT
P3P: policyref="http://p3p.yahoo.com/w3c/p3p.xml", CP="CAO DSP COR CUR ADM ...
Cache-Control: private
Connection: close
Content-Type: text/html

<html><head>
```

```
<title>Yahoo!</title>
...
```

8.2.4 AF_INET6 Socket Addresses

If all 32 bits of an IPv4 address could be used, there would be addresses for 4.3 billion network interfaces, which is probably enough for a while. Because of the way IPv4 addresses are assigned, however, there can be only 534 million so-called Class B addresses (those that begin with 192 through 223), which are used by large organizations, and around 1992 it looked like these would run out by mid-1995. Another problem was that routing tables used by Internet backbone sites were growing too large to fit in the memory of routers available at that time.

The proposed solution was Version 6 of the Internet Protocol, called IPv6, which, among other improvements, expands the address from 4 bytes to 16, which allows 2^{128} addresses.[6]

The preferred way of writing an IPv6 address is as 8 groups of 2 bytes each, such as FEDC:BA98:7654:3210:FEDC:BA98:7654:3210.

You use a `sockaddr_in6` structure to set up an `AF_INET6` socket address; there's more to it than for `AF_INET`, as you would expect:

struct sockaddr_in6—structure for AF_INET6 socket addresses

```
#include <netinet/in.h>

struct sockaddr_in6 {
    sa_family_t sin6_family;    /* AF_INET6 */
    in_port_t sin6_port;        /* port number (uint16_t) */
    uint32_t sin6_flowinfo;     /* traffic class and flow information */
    struct in6_addr sin6_addr;  /* IPv4 address */
    uint32_t sin6_scope_id;     /* set of interfaces for a scope */
};

struct in6_addr {
    uint8_t s6_addr[16];        /* IPv6 address */
};
```

The two new members here that have no equivalent in a `sockaddr_in` structure are `sin6_flowinfo`, which is intended to hold a "flow label" and priority but whose use is currently unspecified, and `sin6_scope_id`, which identifies a set

6. This is more than the number of particles in the universe and over 665 billion trillion addresses *per square meter* of the earth's surface!

of interfaces that are used with certain addresses and is implementation dependent. If you're setting up your own `sockaddr_in6` structure, set these to zero. In fact, you're supposed to initialize the whole structure to zeros before setting the members you're interested in because some implementations have members in addition to those listed in the synopses above.

In practice, you won't be initializing a `sockaddr_in6` structure directly; instead, you'll get one from `getaddrinfo`, which is described in Section 8.2.6.

You can't use `inet_addr` and `inet_ntoa` with IPv6 addresses. Instead you use `inet_ntop` and `inet_pton`, which are in Section 8.9.5.

8.2.5 The Domain Name System (DNS)

Of course, you seldom type a numeric IP address into your Internet browser or FTP client—you type a mnemonic name like *www.yahoo.com* or *www.basepath.com*. These host names are translated to IP addresses by a world-wide distributed database called DNS. There are several standardized functions for accessing the DNS from a UNIX application. The newest and most powerful is called `getaddrinfo`, which is the one we'll use in our examples. Section 8.8.1 briefly describes some older functions (such as `gethostbyname`).

In a URL like *http://www.basepath.com/aup* (the URL for this book's site), the host name is just the middle part. That is, the overall syntax is:[7]

scheme`:`*//hostname/path*

where *scheme* specifies the method of access to the resource (e.g., HTTP, FTP), *hostname* is the name of the host as recorded in the DNS database, and *path* is a path within the host's file system.

In our examples, we use the socket system calls to connect to the host. Once we have a connection, we interact with the host according to the scheme. For instance, in the example in Section 8.2.3, the scheme was HTTP, so we sent the request

```
GET / HTTP/1.0
```

7. This is a simplified view, but it will suffice for our purposes. The complete syntax for Uniform Resource Identifiers is more complicated.

to the server, which interpreted it as a request for a Web page, which it sent back to us. How the path is treated depends on the scheme; for HTTP, it's the argument to the GET command (a single slash in our example). None of the socket system calls care about the path or the scheme.

We don't need to get into HTTP, HTML, FTP, or any other scheme-related subjects in this book beyond the minimum that we need to understand some simple examples. The best way to learn about the various schemes is to read the RFC (Request for Comments) documents [RFC].

8.2.6 `getaddrinfo`

Rather than construct a `sockaddr_in` or `sockaddr_in6` address from scratch, it's easier to call `getaddrinfo`, which builds a socket address for you:

`getaddrinfo`—get socket-address information

```
#include <sys/socket.h>
#include <netdb.h>

int getaddrinfo(
     const char *nodename,        /* node name */
     const char *servname,        /* service name */
     const struct addrinfo *hint, /* hint */
     struct addrinfo **infop      /* returned info as linked list */
);
/* Returns 0 on success or error number on error (errno not set) */
```

`struct addrinfo`—structure for getaddrinfo

```
struct addrinfo {
     int ai_flags;                /* input flags */
     int ai_family;               /* address family */
     int ai_socktype;             /* socket type */
     int ai_protocol;             /* protocol */
     socklen_t ai_addrlen;        /* length of socket address */
     struct sockaddr *ai_addr;    /* socket address */
     char *ai_canonname;          /* canonical name of service location */
     struct addrinfo *ai_next;    /* pointer to next structure in list */
};
```

Typically, you call `getaddrinfo` with `nodename` set to the host name you want, `servname` set to the port number (as a string), and `hint` set to the address family and socket type you want. You get back, through the `infop` argument, a linked list of `addrinfo` structures that contain socket addresses and other information that match your hint. You pick a suitable socket address and use it directly in a call to `connect` or `bind`, casting the `ai_addr` member to `struct sockaddr *`.

The "host name" can be a name to be looked up using a DNS server, a name defined in the local machine's /etc/hosts file, an IPv4 address in dotted notation (as a string), or an IPv6 address in colon notation (as a string).

The one critical flag is `AI_PASSIVE`, which means that the socket addresses returned are for use with `accept`; that is, the call to `getaddrinfo` is from a server. Otherwise, with the flag clear, the call is from a client and the socket address will be used with `connect`. For a connectionless protocol (e.g., `SOCK_DGRAM`), it can also be used with `sendto` or `sendmsg`.

`getaddrinfo` returns its own kind of error code that you can't treat as an `errno` value, which means that you can't use standard functions like `perror` or `strerror` in case of an error. Instead, there's a special function just for the error code returned by `getaddrinfo` and another function, `getnameinfo` (Section 8.8.1):

gai_strerror—get error-code description

```
#include <netdb.h>

const char *gai_strerror(
    int code              /* error code */
);
/* Returns string (no error return) */
```

In our examples, we'll use the error-checking macro `ec_ai`, which knows how to deal with the return from `getaddrinfo` and `getnameinfo`, so we won't call `gai_strerror` directly.

Here's a simple example using `getaddrinfo`:

```
int main(void)
{
    struct addrinfo *infop = NULL, hint;

    memset(&hint, 0, sizeof(hint));
    hint.ai_family = AF_INET;
    hint.ai_socktype = SOCK_STREAM;
    ec_ai( getaddrinfo("www.yahoo.com", "80", &hint, &infop) )
    for ( ; infop != NULL; infop = infop->ai_next) {
        struct sockaddr_in *sa = (struct sockaddr_in *)infop->ai_addr;

        printf("%s port: %d protocol: %d\n", inet_ntoa(sa->sin_addr),
          ntohs(sa->sin_port), infop->ai_protocol);
    }
    exit(EXIT_SUCCESS);
```

```
EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
EC_CLEANUP_END
}
```

The output:

```
66.218.70.48 port: 80 protocol: 0
66.218.70.49 port: 80 protocol: 0
66.218.71.88 port: 80 protocol: 0
66.218.71.86 port: 80 protocol: 0
66.218.70.50 port: 80 protocol: 0
66.218.71.91 port: 80 protocol: 0
66.218.71.80 port: 80 protocol: 0
66.218.71.84 port: 80 protocol: 0
66.218.71.90 port: 80 protocol: 0
66.218.71.93 port: 80 protocol: 0
66.218.71.94 port: 80 protocol: 0
66.218.71.92 port: 80 protocol: 0
66.218.71.89 port: 80 protocol: 0
```

So it seems that *www.yahoo.com* is associated with several different IP numbers. Because of the hint, we know that all of them are `AF_INET` addresses of type `SOCK_STREAM`, so any of them are suitable for access to Web pages. To see this, let's pair up the retrieval of the socket address via `getaddrinfo` with the connection and display code from the example in Section 8.2.3:

```
#define REQUEST "GET / HTTP/1.0\r\n\r\n"

int main(void)
{
    struct addrinfo *infop = NULL, hint;
    int fd_skt;
    char buf[1000];
    ssize_t nread;

    memset(&hint, 0, sizeof(hint));
    hint.ai_family = AF_INET;
    hint.ai_socktype = SOCK_STREAM;
    ec_ai( getaddrinfo("www.yahoo.com", "80", &hint, &infop) )
    ec_neg1( fd_skt = socket(infop->ai_family, infop->ai_socktype,
      infop->ai_protocol) )
    ec_neg1( connect(fd_skt, (struct sockaddr *)infop->ai_addr,
      infop->ai_addrlen) )
    ec_neg1( write(fd_skt, REQUEST, strlen(REQUEST) ) )
    ec_neg1( nread = read(fd_skt, buf, sizeof(buf)) )
    (void)write(STDOUT_FILENO, buf, nread);
    ec_neg1( close(fd_skt) )
    exit(EXIT_SUCCESS);
```

```
EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
EC_CLEANUP_END
}
```

Running this produced the same output that we showed before in Section 8.2.3.

If your system, your DNS server, and the host you're accessing support it, you can use `getaddrinfo` to access more than `SOCK_STREAM AF_INET` addresses on port 80. You can use `NULL` for either the `nodename` or `servname` arguments (but not both) and/or use an address "family" of `AF_UNSPEC` in the hint to find out what's available. Then you search through the returned addresses to find what you want to work with. For example, you might want to use an `AF_INET6` (IPv6) address if one is there. For the details on this advanced use of `getaddrinfo`, see [SUS2002], your system's documentation, or Chapter 11 of [Ste2003].

One more thing before we leave this section: You should free the list returned by `getaddrinfo` when you no longer need it with a call to `freeaddrinfo`:

freeaddrinfo—free socket-address information

```
#include <sys/socket.h>
#include <netdb.h>

void freeaddrinfo(
    struct addrinfo *infop    /* list to free */
);
```

We didn't call `freeaddrinfo` in our examples because they were `main` functions that were going to exit anyway.

8.2.7 `gethostname`

Sometimes, such as in the example Web server in Section 8.4.4, a program needs the name of its own host. Usually the generic name "localhost" will work, but if other machines need to connect to the name, what's needed is the public name assigned by the system administrator. That's what `gethostname` is for:

gethostname—get name of host

```
#include <unistd.h>

int gethostname(
    char *name,           /* returned name */
    size_t namelen        /* size of name buffer */
);
/* Returns 0 on success or -1 on error (errno not defined) */
```

See Section 8.4.4 for an example of `gethostname` in use.

8.3 Socket Options

One reason why the `socket` system call is so simple, even though sockets are complicated, is that all of the options are handled by a separate system call:

setsockopt—set socket options

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

int setsockopt(
    int socket_fd,          /* socket file descriptor */
    int level,              /* level to be accessed */
    int option,             /* option to set */
    const void *value,      /* value to set */
    socklen_t value_len     /* length of value */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

Four of the five arguments are pretty easy to explain: `socket_fd` is the socket file descriptor, `option` is the name of the option, `value` points to the value to set it to, and `value_len` is the length of the value that `value` points to. The value is usually an integer, but sometimes it's a structure specific to the option, as we'll see.

The second argument, `level`, identifies the protocol level to which the option belongs. One level, `SOL_SOCKET`, applies to the socket level itself, and it's the one you'll probably use the most. [SUS2002] defines six additional protocol levels in <netinet/in.h>:

`IPPROTO_IP`	Internet protocol
`IPPROTO_IPV6`	Internet Protocol Version 6
`IPPROTO_ICMP`	Control message protocol
`IPPROTO_RAW`	Raw IP Packets Protocol
`IPPROTO_TCP`	Transmission control protocol
`IPPROTO_UDP`	User datagram protocol

[SUS2002] defines some options for these six protocols, but implementations typically define additional ones. Unfortunately, it's not easy to get a list of all the options—you'll have to read through the documentation for the protocol you're using. For example, on Solaris, typing `man ip` brings up a manual page that describes options for `IPPROTO_IP`.

We'll just describe the standardized `SOL_SOCKET` options here; for the rest, consult your system's documentation or [Ste2003].

There's a matching call to get options, with almost the same arguments:

getsockopt—get socket options

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

int getsockopt(
     int socket_fd,         /* socket file descriptor */
     int level,             /* level to be accessed */
     int option,            /* option to get */
     void *value,           /* returned value */
     socklen_t *value_len   /* length of value */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

The one thing different between `setsockopt` and `getsockopt` is that with the latter the `value_len` argument is a pointer to the length, and it must be set before the call to the size of the buffer pointed to by the `value` argument. On return it's set to the size of the returned value.

Here's a quick example that sets a socket's `SO_REUSEADDR` option (I'll get to what that actually means below):

```
int socket_option_value = 1;

ec_neg1( setsockopt(fd, SOL_SOCKET, SO_REUSEADDR,
  &socket_option_value, sizeof(socket_option_value)) )
```

I'll briefly describe all of the portable `SOL_SOCKET` options. Most of the option values are integers, in which case `value` must point to an `int`, and `value_len` will be `sizeof(int)`. Some integers are Boolean values, for which 1 is true and 0 is false. Don't use the C constant `true` in place of a 1 because `true` may have the value –1, which is fine for C but not acceptable to `setsockopt`. In the following list, the notation (B) indicates a Boolean value, (I) indicates an integer, and (X) indicates a type that's explained further. The letter is followed by an "s" and/or a "g" to indicate whether it's used with `setsockopt`, `getsockopt`, or both.

Be aware that whether some options actually do anything (e.g., SO_KEEPALIVE) depends on the actual protocol in use and its implementation.

SO_ACCEPTCONN Socket is accepting connections; that is, listen has been called. (Bg)

SO_BROADCAST Sending of broadcast messages is allowed if the protocol supports it. (Bsg)

SO_DEBUG Debugging information is recorded by the underlying protocol implementation. (Bsg)

SO_DONTROUTE Sent messages bypass the standard routing facilities and go directly to the network interface according to the destination address. (Bsg)

SO_ERROR Socket error status, which is cleared after it's retrieved. (Ig)

SO_KEEPALIVE Keep the connection active by periodically sending messages. If there's no response, the socket is disconnected. (Writing to a disconnected socket, like writing to a pipe with no reader, generates a SIGPIPE signal.) (Bsg)

SO_LINGER If on, causes a close to block if there are any unsent messages until they're sent or the linger-time expires. (Xsg) The value is a linger structure:

```
struct linger {
    int l_onoff;    /* on (1) or off (0) */
    int l_linger;   /* linger-time in seconds */
};
```

SO_OOBINLINE Received out-of-band data stays inline (see Section 8.7). (Bsg)

SO_RCVBUF Receive buffer size. (Isg)

SO_RCVLOWAT Receive low-water-mark. Blocking receive operations (e.g., read) block until the lesser of this amount and the requested amount are received. The default is 1. (Isg)

SO_RCVTIMEO Uses a timeval structure (Section 1.7.1) for the maximum time to wait for a blocking receive operation to complete. Zero time (the default) means infinite. If the time expires, the operation returns with a partial count or with –1 and errno set to EAGAIN or EWOULDBLOCK. (Xsg)

`SO_REUSEADDR` `bind` allows reuse of local addresses. Otherwise, `bind` returns –1 with `errno` set to `EADDRINUSE` if a previous `bind` has occurred within a system-defined period (for example, a few minutes). Very convenient for debugging and testing. (Bsg)

`SO_SNDBUF` Send buffer size. This option takes an int value. (Isg)

`SO_SNDLOWAT` Send low-water-mark. Nonblocking send operations (e.g., `write`) send no data unless the lesser of this amount and the requested amount can be sent immediately. (Isg)

`SO_SNDTIMEO` Uses a `timeval` structure (Section 1.7.1) for the maximum time to wait for a blocking send operation to complete. Zero time (the default) means infinite. If the time expires, the operation returns with a partial count or with –1 and `errno` set to `EAGAIN` or `EWOULDBLOCK`. (Xsg)

`SO_TYPE` Socket type (e.g., `SOCK_STREAM`). (Isg)

To show how these options are used with `getsockopt`, here's a program that displays the values for several different kinds of sockets:

```
typedef enum {OT_INT, OT_LINGER, OT_TIMEVAL} OPT_TYPE;

static void show(int skt, int level, int option, const char *name,
  OPT_TYPE type)
{
    socklen_t len;
    int n;
    struct linger lng;
    struct timeval tv;

    switch (type) {
    case OT_INT:
        len = sizeof(n);
        if (getsockopt(skt, level, option, &n, &len) == -1)
            printf("%s FAILED (%s)\n", name, strerror(errno));
        else
            printf("%s = %d\n", name, n);
        break;
    case OT_LINGER:
        len = sizeof(lng);
        if (getsockopt(skt, level, option, &lng, &len) == -1)
            printf("%s FAILED (%s)\n", name, strerror(errno));
```

```
        else
            printf("%s = l_onoff: %d; l_linger: %d secs.\n", name,
              lng.l_onoff, lng.l_linger);
        break;
    case OT_TIMEVAL:
        len = sizeof(tv);
        if (getsockopt(skt, level, option, &tv, &len) == -1)
            printf("%s FAILED (%s)\n", name, strerror(errno));
        else
            printf("%s = %ld secs.; %ld usecs.\n", name,
              (long)tv.tv_sec, (long)tv.tv_usec);
    }
}

static void showall(int skt, const char *caption)
{
    printf("\n%s\n", caption);
    show(skt, SOL_SOCKET, SO_ACCEPTCONN, "SO_ACCEPTCONN", OT_INT);
    show(skt, SOL_SOCKET, SO_BROADCAST, "SO_BROADCAST", OT_INT);
    show(skt, SOL_SOCKET, SO_DEBUG, "SO_DEBUG", OT_INT);
    show(skt, SOL_SOCKET, SO_DONTROUTE, "SO_DONTROUTE", OT_INT);
    show(skt, SOL_SOCKET, SO_ERROR, "SO_ERROR", OT_INT);
    show(skt, SOL_SOCKET, SO_KEEPALIVE, "SO_KEEPALIVE", OT_INT);
    show(skt, SOL_SOCKET, SO_LINGER, "SO_LINGER", OT_LINGER);
    show(skt, SOL_SOCKET, SO_OOBINLINE, "SO_OOBINLINE", OT_INT);
    show(skt, SOL_SOCKET, SO_RCVBUF, "SO_RCVBUF", OT_INT);
    show(skt, SOL_SOCKET, SO_RCVLOWAT, "SO_RCVLOWAT", OT_INT);
    show(skt, SOL_SOCKET, SO_RCVTIMEO, "SO_RCVTIMEO", OT_TIMEVAL);
    show(skt, SOL_SOCKET, SO_REUSEADDR, "SO_REUSEADDR", OT_INT);
    show(skt, SOL_SOCKET, SO_SNDBUF, "SO_SNDBUF", OT_INT);
    show(skt, SOL_SOCKET, SO_SNDLOWAT, "SO_SNDLOWAT", OT_INT);
    show(skt, SOL_SOCKET, SO_SNDTIMEO, "SO_SNDTIMEO", OT_TIMEVAL);
    show(skt, SOL_SOCKET, SO_TYPE, "SO_TYPE", OT_INT);
}

int main(void)
{
    int skt;

    ec_neg1( skt = socket(AF_INET, SOCK_STREAM, 0) )
    showall(skt, "AF_INET SOCK_STREAM");
    ec_neg1( close(skt) )
    ec_neg1( skt = socket(AF_INET, SOCK_DGRAM, 0) )
    showall(skt, "AF_INET SOCK_DGRAM");
    ec_neg1( close(skt) )
    exit(EXIT_SUCCESS);

EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
EC_CLEANUP_END
}
```

The output on Solaris was:

```
AF_INET SOCK_STREAM
SO_ACCEPTCONN = 0
SO_BROADCAST = 0
SO_DEBUG = 0
SO_DONTROUTE = 0
SO_ERROR = 0
SO_KEEPALIVE = 0
SO_LINGER = l_onoff: 0; l_linger: 0 secs.
SO_OOBINLINE = 0
SO_RCVBUF = 65536
SO_RCVLOWAT FAILED (Option not supported by protocol)
SO_RCVTIMEO FAILED (Option not supported by protocol)
SO_REUSEADDR = 0
SO_SNDBUF = 65536
SO_SNDLOWAT FAILED (Option not supported by protocol)
SO_SNDTIMEO FAILED (Option not supported by protocol)
SO_TYPE = 2

AF_INET SOCK_DGRAM
SO_ACCEPTCONN = 0
SO_BROADCAST = 0
SO_DEBUG = 0
SO_DONTROUTE = 0
SO_ERROR = 0
SO_KEEPALIVE = 0
SO_LINGER = l_onoff: 0; l_linger: 0 secs.
SO_OOBINLINE = 0
SO_RCVBUF = 65536
SO_RCVLOWAT FAILED (Option not supported by protocol)
SO_RCVTIMEO FAILED (Option not supported by protocol)
SO_REUSEADDR = 0
SO_SNDBUF = 65536
SO_SNDLOWAT FAILED (Option not supported by protocol)
SO_SNDTIMEO FAILED (Option not supported by protocol)
SO_TYPE = 1
```

8.4 Simple Socket Interface (SSI)

Instead of calling `getaddrinfo`, `socket`, `bind`, or `connect` and the other related functions every time we want to use sockets, let's code some higher-level functions to hide some of the tedious details. We'll call the interface to these functions SSI, for Simple Socket Interface.

8.4.1 SSI Function Calls

Internally, the SSI functions keep the state of an open connection in a structure of type `SSI`, which we'll look into shortly. You get a pointer to an `SSI` when you call `ssi_open`, and you close the SSI when you're done with `ssi_close`:

ssi_open—open SSI connection

```
SSI *ssi_open(
    const char *name,      /* server name */
    bool server            /* called from server? */
);
/* Returns pointer to SSI or NULL on error (sets errno) */
```

ssi_close—close SSI connection

```
bool ssi_close(
    SSI *ssip              /* pointer to SSI */
);
/* Returns true on success or false on error (sets errno) */
```

As we'll see, `ssi_open` encapsulates the building of the socket address (e.g., with a call to `getaddrinfo`), and the calls to `socket`, `bind`, `listen`, and `connect`.

The name passed to `ssi_open` is taken as an `AF_INET` host name if it begins with two slashes (which aren't part of the name). The name must be followed by a colon and a port number. If it doesn't begin with two slashes, it's a local `AF_UNIX` name like those in Section 8.1.1. Some examples:

`//www.basepath.com:80`	`AF_INET` connection to port 80 on host *www.basepath.com*
`//firecracker:31000`	`AF_INET` connection to port 31000 on firecracker
`//216.109.125.43:21`	`AF_INET` connection to port 21 on 216.109.125.43
`MyServer`	`AF_UNIX` connection

A server calls `ssi_wait_server` to wait for a client's file descriptor to be ready; it encapsulates the calls to `select` and `accept` that we saw in the example in Section 8.1.3:

ssi_wait_server—wait for ready file descriptor

```
int ssi_wait_server(
    SSI *ssip          /* pointer to SSI */
);
/* Returns file descriptor or -1 on error (sets errno) */
```

A client calls `ssi_get_server_fd` to get the file descriptor for its connection to the server:

ssi_get_server_fd—get server's file descriptor

```
int ssi_get_server_fd(
    SSI *ssip          /* pointer to SSI */
);
/* Returns file descriptor or -1 on error (sets errno) */
```

Finally, a server calls `ssi_close_fd` when it gets an EOF from a client's file descriptor or otherwise knows it will no longer be needed:

ssi_close_fd—close client file descriptor

```
bool ssi_close_fd(
    SSI *ssip,         /* pointer to SSI */
    int fd             /* file descriptor */
);
/* Returns true on success or false on error (sets errno) */
```

These functions are all we need to implement simple servers and clients using connected (`SOCK_STREAM`) clients over `AF_UNIX` or `AF_INET`. We'll show how they're used by showing a simple Web browser and a simple Web server. Then we'll show the SSI implementation.

8.4.2 A Brief Introduction to HTTP

HTTP, the Hypertext Transfer Protocol, is defined by the Internet Engineering Task Force (IETF) in a document called RFC 2616. You can read the whole RFC, and many others, at their Web site [RFC]. What follows is a highly simplified explanation of HTTP—just enough to understand the examples in this section.

For HTTP, the client is normally a Web browser, and the server is a Web server. Once a connection has been made, the client initiates an exchange by sending a string to the server of the form

`GET` *path* `HTTP/`*version*`\r\n\r\n`

where *path* is the desired document (e.g., /index.html) and *version* is the desired version of HTTP to be used (e.g., 1.0).

The server determines whether the document exists and if the client can have access to it. If not, it responds with a status line like

```
HTTP/1.1 404 Not Found\r\n
```

which is followed by an HTML document that explains the error. (We'll get to how documents are sent in a moment.)

If the file can be sent, the status line is like

```
HTTP/1.1 200 OK\r\n
```

Each document, whether HTML text, a JPEG, or whatever, is preceded by a header that describes it, which has this general form (in our examples, anyway):

```
Server: servername\r\n
Content-Length: length\r\n
Content-Type: type\r\n\r\n
```

The *servername* can be anything; we'll use "AUP-ws" for our servers. The *length* is the length of the document in bytes so the client knows how much to read. That way the connection can be left open, since the client doesn't need an EOF to tell it to stop reading. The *type* is a so-called MIME (Multipurpose Internet Mail Extensions) type, only two of which we care about: "text/html" and "image/jpeg." (There are dozens more.) After the header comes the document, exactly as it exists on the server's file system.

8.4.3 SSI Web Browser

Here's a simple Web browser called `minibr`. As with the earlier examples, it can retrieve HTML; however, it doesn't know how to interpret the tags for a nice screen display, so it just dumps everything to the standard output:

```
int main(void)
{
    char url[100], s[500], *path = "", *p;
    SSI *ssip;
    int fd;
    ssize_t nread;

    while (true) {
        printf("URL: ");
        if (fgets(url, sizeof(url), stdin) == NULL)
            break;
        if ((p = strrchr(url, '\n')) != NULL)
            *p = '\0';
```

```
        if ((p = strchr(url, '/')) != NULL) {
            path = p + 1;
            *p = '\0';
        }
        snprintf(s, sizeof(s), "//%s:80", url);
        ec_null( ssip = ssi_open(s, false) )
        ec_neg1( fd = ssi_get_server_fd(ssip) )
        snprintf(s, sizeof(s), "GET /%s HTTP/1.0\r\n\r\n", path);
        ec_neg1( writeall(fd, s, strlen(s)) )
        while (true) {
            switch (nread = read(fd, s, sizeof(s))) {
            case 0:
                printf("EOF\n");
                break;
            case -1:
                EC_FAIL
            default:
                ec_neg1( writeall(STDOUT_FILENO, s, nread) )
                continue;
            }
            break;
        }
        ec_false( ssi_close(ssip) )
    }
    ec_false( !ferror(stdin) )
    exit(EXIT_SUCCESS);

EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
EC_CLEANUP_END
}
```

The code just after the call to `fgets` splits the entered URL into its host name and path (explained in Section 8.2.5). The host name, prepended with slashes and appended with port 80, is passed to `ssi_open`. The path is formatted into the HTTP GET request that's sent to the Web server after the connection is made.

Here's a sample `minibr` session with each retrieved page greatly abbreviated:

```
URL: www.basepath.com
HTTP/1.1 200 OK
Date: Sat, 19 Jul 2003 19:01:41 GMT
Server: Apache/1.3.27 (Unix) FrontPage/5.0.2.2510 mod_jk/1.1.0
Last-Modified: Thu, 15 May 2003 19:56:49 GMT
ETag: "61744-191-3ec3f101"
Accept-Ranges: bytes
Content-Length: 401
Connection: close
Content-Type: text/html
```

```
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">
<html>

<head>
...
URL: 216.109.125.70
HTTP/1.1 200 OK
Date: Sat, 19 Jul 2003 19:02:49 GMT
P3P: policyref="http://p3p.yahoo.com/w3c/p3p.xml", CP="CAO DSP COR CUR ADM
...
Cache-Control: private
Connection: close
Content-Type: text/html

<html><head>

<title>Yahoo!</title>
...
```

8.4.4 SSI Web Server

So far we've been showing clients—now it's time for a server. Our simple Web server looks for a GET request from a client, extracts the path from it, and writes a response followed by the requested file to the client's file descriptor, as described in Section 8.4.2.

It starts with some macros for the HTTP responses that it can provide and some HTML for "not found" errors:

```
#define HEADER\
    "HTTP/1.0 %s\r\n"\
    "Server: AUP-ws\r\n"\
    "Content-Length: %ld\r\n"

#define CONTENT_TEXT\
    "Content-Type: text/html\r\n\r\n"

#define CONTENT_JPEG\
    "Content-Type: image/jpeg\r\n\r\n"

#define HTML_NOTFOUND\
    "<!DOCTYPE html PUBLIC \"-//IETF//DTD HTML 2.0//EN\">\n"\
    "<html><head><title>Error 404</title>\n"\
    "</head><body>\n"\
    "<h2>AUP-ws server can't find document</h2>"\
    "</body></html>\r\n"
```

Note that `HEADER` contains formatting codes that are replaced by the status code and content length when it's actually used by the function `send_header`:

```
static void send_header(SSI *ssip, const char *msg, off_t len,
  const char *path, int fd)
{
    char buf[1000], *dot;

    snprintf(buf, sizeof(buf), HEADER, msg, (long)len);
    ec_neg1( writeall(fd, buf, strlen(buf)) )
    dot = strrchr(path, '.');
    if (dot != NULL && (strcasecmp(dot, ".jpg") == 0 ||
      strcasecmp(dot, ".jpeg") == 0))
        ec_neg1( writeall(fd, CONTENT_JPEG, strlen(CONTENT_JPEG)) )
    else
        ec_neg1( writeall(fd, CONTENT_TEXT, strlen(CONTENT_TEXT)) )
    return;

EC_CLEANUP_BGN
    EC_FLUSH("Error sending response (nonfatal)")
EC_CLEANUP_END
}
```

This function is called with a `msg` of either "404 Not Found" or "200 OK," as I explained in Section 8.4.2. The `len` argument is the length of the document, and `path` is its file name, used only to determine whether the file is a JPEG or HTML. The caller of `send_header` is responsible for sending the file.

`send_header` is called by the function `handle_request` that handles GET requests:

```
#define DEFAULT_DOC "index.html"
#define WEB_ROOT "/aup/webroot/"

static bool handle_request(SSI *ssip, char *s, int fd)
{
    char *tkn, buf[1000], path[500];
    int ntkn;
    FILE *in;
    struct stat statbuf;
    ssize_t nread;

    for (ntkn = 1; (tkn = strtok(s, " ")) != NULL; ntkn++) {
        s = NULL;
        switch (ntkn) {
        case 1:
            if (strcasecmp(tkn, "get") != 0) {
                printf("Unknown request\n");
```

```
                return false;
            }
            continue;
        case 2:
            break;
        }
        break;
    }
    snprintf(path, sizeof(path) - 1 - strlen(DEFAULT_DOC),
      "%s%s", WEB_ROOT, tkn);
    if (stat(path, &statbuf) == 0 && S_ISDIR(statbuf.st_mode)) {
        if (path[strlen(path) - 1] != '/')
            strcat(path, "/");
        strcat(path, DEFAULT_DOC);
    }
    if (stat(path, &statbuf) == -1 ||
      (in = fopen(path, "rb")) == NULL) {
          send_header(ssip, "404 Not Found", strlen(HTML_NOTFOUND), "",
            fd);
        ec_neg1( writeall(fd, HTML_NOTFOUND, strlen(HTML_NOTFOUND)) )
        return false;
    }
    send_header(ssip, "200 OK", statbuf.st_size, path, fd);
    while ((nread = fread(buf, 1, sizeof(buf), in)) > 0)
        ec_neg1( writeall(fd, buf, nread) )
    ec_eof( fclose(in) )
    return true;

EC_CLEANUP_BGN
    EC_FLUSH("Error sending response (nonfatal)")
    return false;
EC_CLEANUP_END
}
```

The `for` loop is just a way to check that it's indeed a GET request and to strip out the path name, which is then taken as a subdirectory of `WEB_ROOT`, to ensure that not just any path on the local system will be served up. If it's a directory, the default HTML file index.html is appended; otherwise, it's assumed to name an HTML or JPEG file.

If the path doesn't exist, the 404 header is sent, followed by some HTML. Otherwise, the 200 header is sent followed by the file. Note that it might be text or a binary JPEG.

Finally, here's the `main` program that contains calls to SSI functions to handle the actual work of connecting to clients:

```
#define PORT ":8000"

int main(void)
{
    SSI *ssip = NULL;
    char msg[1600];
    ssize_t nrcv;
    int fd;
    char host[100] = "//";

    ec_neg1( gethostname(&host[2], sizeof(host) - 2 - strlen(PORT)) )
    strcat(host, PORT);
    printf("Connecting to host \"%s\"\n", host);
    ec_null( ssip = ssi_open(host, true) )
    printf("\t...connected\n");
    while (true) {
        ec_neg1( fd = ssi_wait_server(ssip) )
        switch (nrcv = read(fd, msg, sizeof(msg) - 1)) {
        case -1:
            printf("Read error (nonfatal)\n");
            /* fall through */
        case 0:
            ec_false( ssi_close_fd(ssip, fd) )
            continue;
        default:
            msg[nrcv] = '\0';
            (void)handle_request(ssip, msg, fd);
        }
    }
    ec_false( ssi_close(ssip) )
    printf("Done.\n");
    exit(EXIT_SUCCESS);

EC_CLEANUP_BGN
    return EXIT_FAILURE;
EC_CLEANUP_END
}
```

The server name for `ssi_open` is formed like this:

//host`:8000`

We use port 8000 because port 80 was already in use by a real Web server (Apache) on our system and because ports under 1024 are restricted to the superuser.

The function connects and then goes into a loop in which it waits for a ready file descriptor, reads a command, and gives it to `handle_request`. It can get an EOF if a client terminates or simply closes the connection, which some clients do from

time to time. In that case it has to call `ssi_close_fd` to tell `ssi_wait_server` not to consider that file descriptor any more, as I explained in Section 8.1.3.

This Web server really works, and you can access it from any browser, not just the simple one in the previous section. For example, Figure 8.3 shows it being accessed via the URL *http://suse2:8000:*

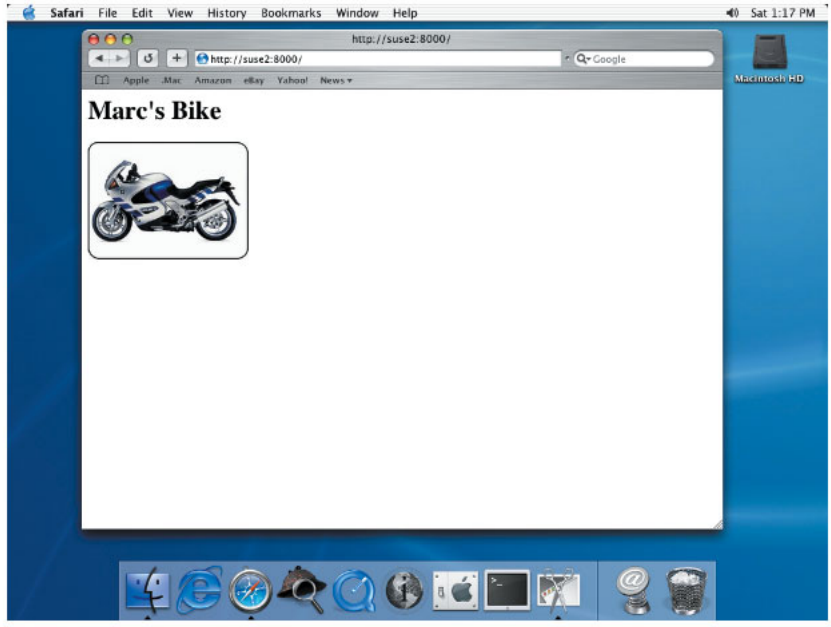


Figure 8.3 SSI Web server accessed via Safari browser on a Macintosh.

The HTML in /aup/webroot/index.html on the server (called suse2 because it runs SuSE Linux) was:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">
<html>
<head>
</head>
<body>
<h1>Marc's Bike</h1>
<p><img src="picture.jpg">
</body>
</html>
```

8.4.5 SSI Implementation

I've discussed all the issues involved in implementing the SSI functions and even shown all the code in one example or another. You may want to review Sections 8.1.3, 8.2.2, and 8.2.6.

This is the structure for the type `SSI`:

```
#define SSI_NAME_SIZE 200

typedef struct {
    bool ssi_server;                    /* server? (vs. client) */
    int ssi_domain;                     /* AF_INET or AF_UNIX */
    int ssi_fd;                         /* socket fd */
    char ssi_name_server[SSI_NAME_SIZE]; /* server name */
    fd_set ssi_fd_set;                  /* set for server's select */
    int ssi_fd_hwm;                     /* high-water-mark for fds seen */
} SSI;
```

We'll see how the members are used as we go though the code.

First comes a handy function that can make socket addresses for `AF_UNIX` or `AF_INET`. The name argument is of the form host:port for `AF_INET`, and just a plain name for `AF_UNIX`. The `will_bind` argument indicates whether the socket address is for use with `bind` or `connect`, as explained in Section 8.2.6.

```
bool make_sockaddr(struct sockaddr *sa, socklen_t *len, const char *name,
  int domain, bool will_bind)
{
    struct addrinfo *infop = NULL;

    if (domain == AF_UNIX) {
        struct sockaddr_un *sunp = (struct sockaddr_un *)sa;

        if (strlen(name) >= sizeof(sunp->sun_path)) {
            errno = ENAMETOOLONG;
            EC_FAIL
        }
        strcpy(sunp->sun_path, name);
        sunp->sun_family = AF_UNIX;
        *len = sizeof(*sunp);
    }
    else {
        struct addrinfo hint;
        char nodename[SSI_NAME_SIZE], *servicename;

        memset(&hint, 0, sizeof(hint));
        hint.ai_family = domain;
```

```
        hint.ai_socktype = SOCK_STREAM;
        if (will_bind)
            hint.ai_flags = AI_PASSIVE;
        strcpy(nodename, name);
        servicename = strchr(nodename, ':');
        if (servicename == NULL) {
            errno = EINVAL;
            EC_FAIL
        }
        *servicename = '\0';
        servicename++;
        ec_ai( getaddrinfo(nodename, servicename, &hint, &infop) )
        memcpy(sa, infop->ai_addr, infop->ai_addrlen);
        *len = infop->ai_addrlen;
        freeaddrinfo(infop);
    }
    return true;

EC_CLEANUP_BGN
    if (infop != NULL)
        freeaddrinfo(infop);
    return false;
EC_CLEANUP_END
}
```

I deliberately kept `make_sockaddr` independent of the `SSI` structure in case you want to pull it out and use it in your own applications.

Next come two functions used by `ssi_open` and `ssi_close` to maintain the high-water-mark of file descriptors, just as we did in Section 8.1.3:

```
static void set_fd_hwm(SSI *ssip, int fd)
{
    if (fd > ssip->ssi_fd_hwm)
        ssip->ssi_fd_hwm = fd;
}

static void reset_fd_hwm(SSI *ssip, int fd)
{
    if (fd == ssip->ssi_fd_hwm)
        ssip->ssi_fd_hwm--;
}
```

Now we're ready for `ssi_open`, which doesn't do anything we haven't already seen:

```
SSI *ssi_open(const char *name_server, bool server)
{
    SSI *ssip = NULL;
```

```
    struct sockaddr_storage sa;
    socklen_t sa_len;

    ec_null(  ssip = calloc(1, sizeof(SSI)) )
    ssip->ssi_server = server;
    if (strncmp(name_server, "//", 2) == 0) {
        ssip->ssi_domain = AF_INET;
        name_server += 2;
    }
    else {
        ssip->ssi_domain = AF_UNIX;
        if (ssip->ssi_server)
            (void)unlink(name_server);
    }
    if (strlen(name_server) >= sizeof(ssip->ssi_name_server)) {
        errno = ENAMETOOLONG;
        EC_FAIL
    }
    strcpy(ssip->ssi_name_server, name_server);
    ec_false( make_sockaddr((struct sockaddr *)&sa, &sa_len,
      ssip->ssi_name_server, ssip->ssi_domain, ssip->ssi_server) )
    ec_neg1(  ssip->ssi_fd = socket(ssip->ssi_domain, SOCK_STREAM, 0) )
    set_fd_hwm(ssip, ssip->ssi_fd);
    if (ssip->ssi_domain == AF_INET) {
        int socket_option_value = 1;

        ec_neg1( setsockopt(ssip->ssi_fd, SOL_SOCKET, SO_REUSEADDR,
          &socket_option_value, sizeof(socket_option_value)) )
    }
    if (ssip->ssi_server) {
        FD_ZERO(&ssip->ssi_fd_set);
        ec_neg1( bind(ssip->ssi_fd, (struct sockaddr *)&sa, sa_len) )
        ec_neg1( listen(ssip->ssi_fd, SOMAXCONN) )
        FD_SET(ssip->ssi_fd, &ssip->ssi_fd_set);
    }
    else
        ec_neg1( connect(ssip->ssi_fd, (struct sockaddr *)&sa, sa_len) )
    return ssip;

EC_CLEANUP_BGN
    free(ssip);
    return NULL;
EC_CLEANUP_END
}
```

The socket option `SO_REUSEADDR` was explained in Section 8.3.

Closing an `SSI` is straightforward:

```
bool ssi_close(SSI *ssip)
{
    if (ssip->ssi_server) {
        int fd;

        for (fd = 0; fd <= ssip->ssi_fd_hwm; fd++)
            if (FD_ISSET(fd, &ssip->ssi_fd_set))
                (void)close(fd);
        if (ssip->ssi_domain == AF_UNIX)
            (void)unlink(ssip->ssi_name_server);
    }
    else
        (void)close(ssip->ssi_fd);
    free(ssip);
    return true;
}
```

Next comes `ssi_wait_server`, which is just a reworking of the code from Section 8.1.3, only it returns a ready file descriptor (other than the server's socket, that is) instead of using it itself:

```
int ssi_wait_server(SSI *ssip)
{
    if (ssip->ssi_server) {
        fd_set fd_set_read;
        int fd, clientfd;
        struct sockaddr_un from;
        socklen_t from_len = sizeof(from);

        while (true) {
            fd_set_read = ssip->ssi_fd_set;
            ec_neg1( select(ssip->ssi_fd_hwm + 1, &fd_set_read, NULL, NULL,
              NULL) )
            for (fd = 0; fd <= ssip->ssi_fd_hwm; fd++)
                if (FD_ISSET(fd, &fd_set_read)) {
                    if (fd == ssip->ssi_fd) {
                        ec_neg1( clientfd = accept(ssip->ssi_fd,
                          (struct sockaddr *)&from, &from_len) );
                        FD_SET(clientfd, &ssip->ssi_fd_set);
                        set_fd_hwm(ssip, clientfd);
                        continue;
                    }
                    else
                        return fd;
                }
        }
    }
```

```
    else {
        errno = ENOTSUP;
        EC_FAIL
    }

EC_CLEANUP_BGN
    return -1;
EC_CLEANUP_END
}
```

The other two functions are trivial:

```
int ssi_get_server_fd(SSI *ssip)
{
    return ssip->ssi_fd;
}

bool ssi_close_fd(SSI *ssip, int fd)
{
    ec_neg1( close(fd) );
    FD_CLR(fd, &ssip->ssi_fd_set);
    reset_fd_hwm(ssip, fd);
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

That's the whole implementation. For connected sockets (`SOCK_STREAM`), we won't have to use any of the raw system calls in any further examples.

8.5 Socket Implementation of SMI

It's interesting to implement the Simple Messaging Interface (SMI) from Chapter 7 to use sockets so we can compare it to the five other implementations shown there. We'll use the SSI functions from the previous section rather than call the various socket-related system calls directly, so this job is pretty easy.

This section assumes you're very familiar with Chapter 7 and the SMI implementations there; if you're not, skip this section until you've had time to get through Chapter 7—otherwise it won't make much sense.

For sockets, the internal `SMIQ_SKT` structure is simple, mostly because the work of keeping track of a client that the other implementations were burdened with is

in this case done by the `accept` system call and because we already have SSI to handle a lot of the details:

```
typedef struct {
    SMIENTITY sq_entity;            /* entity */
    SSI *sq_ssip;                   /* structure for SSI */
    struct client_id sq_client;     /* client ID */
    size_t sq_msgsize;              /* msg size */
    struct smi_msg *sq_msg;         /* msg buffer */
} SMIQ_SKT;
```

There's not much to do to open the `SMIQ_SKT` since `ssi_open` does most of the work:

```
SMIQ *smi_open_skt(const char *name, SMIENTITY entity, size_t msgsize)
{
    SMIQ_SKT *p = NULL;

    ec_null(  p = calloc(1, sizeof(SMIQ_SKT)) )
    p->sq_msgsize = msgsize + offsetof(struct smi_msg, smi_data);
    ec_null( p->sq_msg = calloc(1, p->sq_msgsize) )
    p->sq_entity = entity;
    ec_null( p->sq_ssip = ssi_open(name, entity == SMI_SERVER) )
    return (SMIQ *)p;

EC_CLEANUP_BGN
    (void)smi_close_skt((SMIQ *)p);
    return NULL;
EC_CLEANUP_END
}
```

Ditto for closing:

```
bool smi_close_skt(SMIQ *sqp)
{
    SMIQ_SKT *p = (SMIQ_SKT *)sqp;
    SSI *ssip;

    if (p != NULL) {
        ssip = p->sq_ssip;
        free(p->sq_msg);
        free(p);
        if (ssip != NULL)
            ec_false( ssi_close(ssip) )
    }
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

As we've seen for some of the other implementations in Chapter 7, all `smi_send_getaddr_skt` has to do is save the `client_id`, if it's called by a server, and return the message address:

```
bool smi_send_getaddr_skt(SMIQ *sqp, struct client_id *client,
  void **addr)
{
    SMIQ_SKT *p = (SMIQ_SKT *)sqp;

    if (p->sq_entity == SMI_SERVER)
        p->sq_client = *client;
    *addr = p->sq_msg;
    return true;
}
```

`smi_send_release_skt` writes the message (using `writeall` from Section 2.9) either to the client's file descriptor, if it's from the server, or to the server's socket file descriptor, if it's from a client:

```
bool smi_send_release_skt(SMIQ *sqp)
{
    SMIQ_SKT *p = (SMIQ_SKT *)sqp;
    int fd;

    if (p->sq_entity == SMI_SERVER)
        ec_neg1( fd = p->sq_client.c_id1 )
    else
        ec_neg1( fd = ssi_get_server_fd(p->sq_ssip) )
    ec_neg1( writeall(fd, p->sq_msg, p->sq_msgsize) )
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

For a server, receiving a message by the server means having to wait for a ready file descriptor, which SSI handles for us via `ssi_wait_server`. A server also has to save the file descriptor that was ready in the `client_id` part of the message so the caller (the server) will know who to respond to. All a client has to do is call `ssi_get_server_fd` to get the server's socket file descriptor:

```
bool smi_receive_getaddr_skt(SMIQ *sqp, void **addr)
{
    SMIQ_SKT *p = (SMIQ_SKT *)sqp;
    ssize_t /*nremain,*/ nread;
    int fd;
```

```
    *addr = p->sq_msg;
    while (true) {
        if (p->sq_entity == SMI_SERVER)
            ec_neg1( fd = ssi_wait_server(p->sq_ssip) )
        else
            ec_neg1( fd = ssi_get_server_fd(p->sq_ssip) )
        ec_neg1( nread = readall(fd, p->sq_msg, p->sq_msgsize) )
        if (nread == 0) {
            if (p->sq_entity == SMI_SERVER) {
                ec_false( ssi_close_fd(p->sq_ssip, fd) )
                continue;
            }
            else {
                errno = ENOMSG;
                EC_FAIL
            }
        }
        else
            break;
    }
    if (p->sq_entity == SMI_SERVER)
        p->sq_msg->smi_client.c_id1 = fd;
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

And the final function has no work to do at all:

```
bool smi_receive_release_skt(SMIQ *sqp)
{
    return true;
}
```

With this SMI implementation, all of the message-sending examples from Chapter 7 work on sockets. We already compared the performance relative to the other IPC methods in Section 7.15.

Note that our SMI examples, which were designed for IPC within a machine, use plain server names. This means that when the name gets into `ssi_open` it's treated as being in the `AF_UNIX` domain. You could just as easily use the SMI interface for `AF_INET` message-passing, between machines, although if your names are written that way—beginning with two slashes—only the socket implementation of SMI will work.

8.6 Connectionless Sockets

So far all of our examples and the implementation of SSI used connected sockets of type `SOCK_STREAM`. There was a server that called `listen` and `accept`, and one or more clients that connected to it. The connection stayed open until one side or the other closed it.

By contrast, connectionless sockets don't use a connection so `listen` and `accept` aren't called. A sender specifies the address it wants to send to. A receiver specifies the address it wants to receive from, or it receives from anyone and is told the address of the sender. Receivers always have to `bind` to an external name since that's the only way a sender can reach them. Thus, connectionless sockets are more symmetrical than connected ones; there could be a client/server relationship, but it's more useful sometimes to think of the participants as peers.

8.6.1 About Datagrams

Connectionless communication uses *datagrams*, which are independent chunks of data containing a destination address. There's no attempt to keep datagrams in order, and no guarantee that they'll even arrive. By contrast, a `SOCK_STREAM` guarantees both ordering and arrival. Think of a datagram like a postal letter or an email, and a `SOCK_STREAM` like a telephone call.

In many applications, the lack of guarantees of order and arrival doesn't really matter. For example, suppose the application is for reserving library books. There's a form on the screen which, when submitted, sends a datagram to the library's server, which responds with a confirmation. Reordering the datagrams is perhaps a bit unfair, if one reservation preceded another by a second or so, but since the patrons are unaware of each other's activities, there won't be any complaints. In the rare case that a datagram doesn't arrive at all, the patron won't get the confirmation and will just send the form again. So, in this application, datagrams are much more efficient than setting up a connection just to reserve a book and then immediately disconnecting. On the other hand, if the patron is going to browse the catalog of books for a while interactively, then setting up a connection is probably the best choice.

Figure 8.4 shows two peers sending datagrams to each other's sockets. Compare this to Figure 8.1, which showed connected sockets. The symbols for listening and accepting are gone, and both sockets are bound to names, not just the server's socket.

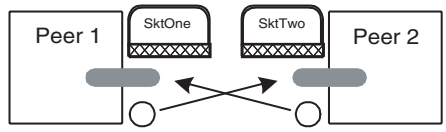


Figure 8.4 Peers sending datagrams.

8.6.2 `sendto` and `recvfrom` System Calls

In all of our previous socket examples, we used `write` (or `writeall`) to send data to a socket, which worked because we had a file descriptor representing the writing end of the connection. But, if the sockets aren't connected, we only have one file descriptor, to our own socket; we need to specify the target as a socket address instead. So, `write` won't do. We need another system call named `sendto` that takes a socket address:

sendto—send message to socket

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

ssize_t sendto(
     int socket_fd,                 /* socket file descriptor */
     const void *message,           /* message to send */
     size_t length,                 /* length of message */
     int flags,                     /* flags */
     const struct sockaddr *sa,     /* target address */
     socklen_t sa_len               /* length of target address */
);
/* Returns number of bytes sent or -1 on error (sets errno) */
```

The argument `socket_fd` is the sender's socket; the receiver is specified with arguments `sa` and `sa_len`. A successful return from `sendto` doesn't mean the message arrived, only that there weren't any locally detected errors.

Your implementation may define some flags, but the only important portable one is `MSG_OOB`, which sends out-of-band (urgent) data; see Section 8.7.

For connectionless sockets, the message of `length` bytes is considered as an indivisible datagram. If the socket channel is full, `sendto` normally blocks until it all

will fit; if the `O_NONBLOCK` flag is set and it won't fit, none of it is sent, –1 is returned, and `errno` is set to `EAGAIN` or `EWOULDBLOCK`.

You can use `sendto` with a connected socket, but in that case the destination socket address is ignored and the output goes to the `socket_fd` socket, just as for `write`. The only advantage over `write` is that you can specify flags, but, if that's all you want, it would be more straightforward to use `send` (Section 8.9.1).

Now we can show an example that does what Figure 8.4 shows, with the addition that each peer reads incoming data. Peer 2 gets a message, makes a little change to it, and sends it to Peer 1. Peer 1 puts together four messages to send to Peer 2 and displays the responses. Note that, unlike the connected-socket examples, here both peers do a `bind`, and neither calls `listen`, `accept`, or `connect`.

```
#define SOCKETNAME1 "SktOne"
#define SOCKETNAME2 "SktTwo"

#define MSG_SIZE 100

int main(void)
{
    struct sockaddr_un sa1, sa2;

    strcpy(sa1.sun_path, SOCKETNAME1);
    sa1.sun_family = AF_UNIX;
    strcpy(sa2.sun_path, SOCKETNAME2);
    sa2.sun_family = AF_UNIX;
    (void)unlink(SOCKETNAME1);
    (void)unlink(SOCKETNAME2);
    if (fork() == 0) { /* Peer 1 */
        int fd_skt;
        ssize_t nread;
        char msg[MSG_SIZE];
        int i;

        sleep(1); /* let peer 2 startup first */
        ec_neg1( fd_skt = socket(AF_UNIX, SOCK_DGRAM, 0) )
        ec_neg1( bind(fd_skt, (struct sockaddr *)&sa1, sizeof(sa1)) )
        for (i = 1; i <= 4; i++) {
            snprintf(msg, sizeof(msg), "Message #%d", i);
            ec_neg1( sendto(fd_skt, msg, sizeof(msg), 0,
              (struct sockaddr *)&sa2, sizeof(sa2)) )
            ec_neg1( nread = read(fd_skt, msg, sizeof(msg)) )
            if (nread != sizeof(msg)) {
                printf("Peer 1 got short message\n");
                break;
            }
```

```
            printf("Got \"%s\" back\n", msg);
        }
        ec_neg1( close(fd_skt) )
        exit(EXIT_SUCCESS);
    }
    else { /* Peer 2 */
        int fd_skt;
        ssize_t nread;
        char msg[MSG_SIZE];

        ec_neg1( fd_skt = socket(AF_UNIX, SOCK_DGRAM, 0) )
        ec_neg1( bind(fd_skt, (struct sockaddr *)&sa2, sizeof(sa2)) )
            ec_neg1( nread = read(fd_skt, msg, sizeof(msg)) )
        while (true) {
            if (nread != sizeof(msg)) {
                printf("Peer 2 got short message\n");
                break;
            }
            msg[0] = 'm';
            ec_neg1( sendto(fd_skt, msg, sizeof(msg), 0,
              (struct sockaddr *)&sa1, sizeof(sa1)) )
        }
        ec_neg1( close(fd_skt) )
        exit(EXIT_SUCCESS);
    }

EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
EC_CLEANUP_END
}
```

Putting both processes in the same program isn't normal; we did it just to make the example simpler. However, we had to stick in the call to `sleep` at the start of the child process (Peer 1) to let Peer 2 go first. Not the best way to sequence things but OK for our present purposes. In a real application getting the server started before the clients isn't usually a problem. If it is, a correct way to synchronize the two is in Section 9.2.

This is the output:

```
Got "message #1" back
Got "message #2" back
Got "message #3" back
Got "message #4" back
```

For datagrams `read` isn't really a good choice because it doesn't tell us who sent the data. This wasn't a problem for connected sockets because each return from `accept` gave us a separate file descriptor for each client that we could both `read`

and `write`. In the example just given, Peer 2 merely assumed it was supposed to `sendto` Peer 1, but, in general, applications are more complicated than that.

Sure enough, there's the opposite of `sendto` called `recvfrom`:

recvfrom—receive message from socket

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

ssize_t recvfrom(
     int socket_fd,          /* socket file descriptor */
     void *buffer,           /* buffer for received message */
     size_t length,          /* length of buffer */
     int flags,              /* flags */
     struct sockaddr *sa,    /* address of sender */
     socklen_t *sa_len       /* address length */
);
/* Returns number of bytes received, 0, or -1 on error (sets errno) */
```

`recvfrom`'s first three arguments are like those of `read`. In addition, it has a `flags` argument and two arguments that receive the sender's socket address. Prior to the call, you must set `sa` to something big enough to hold the sender's socket address, and then `sa_len` to the size of that storage. On return `sa_len` is set to the actual size of the address. Then, if you want, you can pass `sa` and `sa_len` directly to `sendto` for a response.

Like `sendto`, `recvfrom` always returns a complete message for a connectionless socket. It waits for one if `O_NONBLOCK` is clear, or, if set, returns –1 with `errno` to `EAGAIN` or `EWOULDBLOCK`.

You could use `recvfrom` for a connected socket if you want to know the source address, but there's not a whole lot you can do with that address since it would be ignored in a call to `sendto` anyway, as previously explained. Another way to get the address of a socket whose file descriptor you have would be to call `getsockname` (Section 8.9.2).

There are three portable flags that you can use with `recvfrom`:

`MSG_OOB` Receives out-of-band data, if any; see Section 8.7.

`MSG_PEEK` Returns the message, but leaves it unread for the next `read`, `recvfrom`, or other input operation.

`MSG_WAITALL` For connected sockets only, causes `recvfrom` to block until the entire requested length is available, unless `MSG_PEEK` is also set

or the call is interrupted by a signal, termination of the connection, or an error. Has no effect on connectionless sockets because the message is always indivisible.

Here's my example reworked to use `recvfrom` and to have several peers sending to the same socket, so it's a client/server arrangement:

```
#define SOCKETNAME_SERVER "SktOne"
#define SOCKETNAME_CLIENT "SktTwo"

static struct sockaddr_un sa_server;

#define MSG_SIZE 100

static void run_client(int nclient)
{
    struct sockaddr_un sa_client;
    int fd_skt;
    ssize_t nrecv;
    char msg[MSG_SIZE];
    int i;

    if (fork() == 0) { /* client */
        sleep(1); /* let server startup first */
        ec_neg1( fd_skt = socket(AF_UNIX, SOCK_DGRAM, 0) )
        snprintf(sa_client.sun_path, sizeof(sa_client.sun_path),
          "%s-%d", SOCKETNAME_CLIENT, nclient);
        (void)unlink(sa_client.sun_path);
        sa_client.sun_family = AF_UNIX;
        ec_neg1( bind(fd_skt, (struct sockaddr *)&sa_client,
          sizeof(sa_client)) )
        for (i = 1; i <= 4; i++) {
            snprintf(msg, sizeof(msg), "Message #%d", i);
            ec_neg1( sendto(fd_skt, msg, sizeof(msg), 0,
              (struct sockaddr *)&sa_server, sizeof(sa_server)) )
            ec_neg1( nrecv = read(fd_skt, msg, sizeof(msg)) )
            if (nrecv != sizeof(msg)) {
                printf("client got short message\n");
                break;
            }
            printf("Got \"%s\" back\n", msg);
        }
        ec_neg1( close(fd_skt) )
        exit(EXIT_SUCCESS);
    }
    return;
```

```
EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
EC_CLEANUP_END
}

static void run_server(void)
{
    int fd_skt;
    ssize_t nrecv;
    char msg[MSG_SIZE];
    struct sockaddr_storage sa;
    socklen_t sa_len;

    ec_neg1( fd_skt = socket(AF_UNIX, SOCK_DGRAM, 0) )
    ec_neg1( bind(fd_skt, (struct sockaddr *)&sa_server, sizeof(sa_server)) )
    while (true) {
        sa_len = sizeof(sa);
        ec_neg1( nrecv = recvfrom(fd_skt, msg, sizeof(msg), 0,
          (struct sockaddr *)&sa, &sa_len) )
        if (nrecv != sizeof(msg)) {
            printf("server got short message\n");
            break;
        }
        msg[0] = 'm';
        ec_neg1( sendto(fd_skt, msg, sizeof(msg), 0,
          (struct sockaddr *)&sa, sa_len) )
    }
    ec_neg1( close(fd_skt) )
    exit(EXIT_SUCCESS);

EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
EC_CLEANUP_END
}

int main(void)
{
    int nclient;

    strcpy(sa_server.sun_path, SOCKETNAME_SERVER);
    sa_server.sun_family = AF_UNIX;
    (void)unlink(SOCKETNAME_SERVER);
    for (nclient = 1; nclient <= 3; nclient++)
        run_client(nclient);
    run_server();
    exit(EXIT_SUCCESS);
}
```

In this new example, only the server's socket address is known to the server and the clients. A client socket address is known only to that client; the server gets it from its call to `recvfrom`. I left the `read` calls in the clients alone, but those could have been calls to `recvfrom` as well. There's more output since there are now three clients:

```
Got "message #1" back
Got "message #2" back
Got "message #3" back
Got "message #4" back
Got "message #1" back
Got "message #2" back
Got "message #3" back
Got "message #4" back
Got "message #1" back
Got "message #2" back
Got "message #3" back
Got "message #4" back
```

8.6.3 `sendmsg` and `recvmsg`

There are variants of `sendto` and `recvfrom` that use the same flags and have the same return value but which use a single argument that points to a `msghdr` structure that contains both the socket address and the message. They're capable of scatter reading and gather writing, just like `readv` and `writev`, which we saw way back in Section 2.15.

sendmsg—send message to socket using msghdr structure

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

ssize_t sendmsg(
    int socket_fd,                /* socket file descriptor */
    const struct msghdr *message, /* message */
    int flags                     /* flags */
);
/* Returns number of bytes sent or -1 on error (sets errno) */
```

recvmsg—receive message from socket using msghdr structure

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

ssize_t recvmsg(
    int socket_fd,                /* socket file descriptor */
    struct msghdr *message,       /* message */
    int flags                     /* flags */
);
/* Returns number of bytes received, 0, or -1 on error (sets errno) */
```

struct msghdr—structure for sendmsg and recvmsg

```
struct msghdr {
    void *msg_name;           /* optional address */
    socklen_t msg_namelen;    /* size of address */
    struct iovec *msg_iov;    /* scatter/gather array */
    int msg_iovlen;           /* number of elements in msg_iov */
    void *msg_control;        /* ancillary data */
    socklen_t msg_controllen; /* ancillary data buffer len */
    int msg_flags;            /* flags on received message */
};
```

For `sendmsg`, the socket address is pointed to by the misnamed `msg_name` member, and the socket-address length goes in `msg_namelen`. The data to be sent is pointed to via the `msg_iov` member, which points to a `iovec` structure and is used just as with `writev`. Member `msg_iovlen` is the number of elements in the `msg_iov` array. I explained setting up these members in Section 2.15. Member `msg_flags` is ignored.

For `recvmsg`, you can set `msg_name` to `NULL` or to a buffer of length `msg_namelen` to receive the sender's socket address, similar to the way `recvfrom` works. When `recvmsg` returns, `msg_namelen` is changed to the actual socket-address length. Members `msg_iov` and `msg_iovlen` are used as with `readv`. There's one portable flag that can be set in the returned `msg_flags` member: `MSG_TRUNC`, which means that the message was too long to fit in the supplied buffers.

The socket-address members, `msg_name` and `msg_namelen`, are used as with connectionless sockets; for connected sockets they're ignored.

The two other members I didn't explain, `msg_control` and `msg_controllen`, are used for ancillary data that has to do with access rights. I won't go into it here, but you can read about it in [SUS2002] or in your system's documentation.

Here's just the `run_server` function from the previous section redone to use `recvmsg` and `sendmsg`:

```
static void run_server(void)
{
    int fd_skt;
    ssize_t nrecv;
    char msg[MSG_SIZE];
    struct sockaddr_storage sa;
```

```
    struct msghdr m;
    struct iovec v;

    ec_neg1( fd_skt = socket(AF_UNIX, SOCK_DGRAM, 0) )
    ec_neg1( bind(fd_skt, (struct sockaddr *)&sa_server, sizeof(sa_server)) )
    while (true) {
        memset(&m, 0, sizeof(m));
        m.msg_name = &sa;
        m.msg_namelen = sizeof(sa);
        v.iov_base = msg;
        v.iov_len = sizeof(msg);
        m.msg_iov = &v;
        m.msg_iovlen = 1;
        ec_neg1( nrecv = recvmsg(fd_skt, &m, 0) )
        if (nrecv != sizeof(msg)) {
            printf("server got short message\n");
            break;
        }
        ((char *)m.msg_iov->iov_base)[0] = 'm';
        ec_neg1( sendmsg(fd_skt, &m, 0) )
    }
    ec_neg1( close(fd_skt) )
    exit(EXIT_SUCCESS);

EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
EC_CLEANUP_END
}
```

There's additional work to set up the `msghdr` structure, but the `recvmsg` and `sendmsg` calls are then very simple. Imagine an application where there are a lot of different messages being sent to different addresses, and you can begin to see the benefits of keeping the addresses in the same structure as the message data.

8.6.4 Using connect with Connectionless Sockets

Normally a client uses `connect` (Section 8.1.2) to connect with a server's socket, but it also works with connectionless sockets to establish a default address for subsequent `send`s and `recv`s, which don't take a socket-address argument. The specifics of those two system calls are in Section 8.9.1.

Calling `connect` with a `NULL` socket address removes the default address.

8.7 Out-of-Band Data

Occasionally it's necessary to send an urgent message on a socket connection. If it's sent inline, it won't be read until everything ahead of it is read first, so there's a way to send it *out-of-band*. The receiver also must be receiving out-of-band messages.

To send an out-of-band message, set the `MSG_OOB` flag with `sendto`, `sendmsg`, or `send`.

A receiver using `select` (Section 8.1.3) finds out about an out-of-band message through the so-called "error" set of file descriptors passed as the fourth argument to `select`. Then it gets the message by setting the `MSG_OOB` flag in a call to `recvfrom`, `recvmsg`, or `recv`. Or, it can just check for an out-of-band message every time it's about to receive ordinary data by calling one of the receiving functions with the `O_NONBLOCK` flag set on the socket file descriptor (via `fcntl`, Section 3.8.3) and `MSG_OOB` set for the receiving call.

If the `SO_OOBINLINE` option is set (Section 8.3), out-of-band data may be placed inline, along with the other data. The protocol may precede the out-of-band data with an out-of-band mark, ignored when data is read, but whose presence can be detected with a call just for that purpose:

sockatmark—test presence of out-of-band mark

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

int sockatmark(
    int socket_fd      /* socket file descriptor */
);
/* Returns 1 if at mark, 0 if not, or -1 on error (sets errno) */
```

The problem with `sockatmark` is that, if it returns 0 because the socket is empty, out-of-band data may arrive after the call to `sockatmark` but before the next receive operation. Since the out-of-band mark is ignored when data is received, the mark will be lost. The way to avoid this is ensure that data is ready to be received before calling `sockatmark`, with, for example, a call to `select`. That way a 0 return from `sockatmark` means unambiguously that there is data, but it is not preceded by the out-of-band mark.

You can also arrange to get a `SIGURG` signal when out-of-band data has arrived. To do this, call `fcntl` with the `F_SETOWN` operation to set the process or process-group that should get the signal. For more details, see [SUS2002] or your system's documentation.

8.8 Network Database Functions

These *network database* functions provide information about hosts, networks, protocols, and services. I already introduced the most important of these functions, `getaddrinfo`, in Section 8.2.6. Another handy function, `gethostname`, was in Section 8.2.7. Here I'll describe the rest, grouped according to the kind of information they deal with.

8.8.1 Host Functions

The three functions for scanning the host database are:

sethostent—start host-database scan

```
#include <netdb.h>

void sethostent(
    int stayopen           /* leave connection open? */
);
```

gethostent—get next host-database entry

```
#include <netdb.h>

struct hostent *gethostent(void);
/* Returns next entry or NULL on end of database (errno not defined) */
```

endhostent—end host-database scan

```
#include <netdb.h>

void endhostent(void);
```

struct hostent—structure for host-database functions

```
struct hostent {
    char *h_name;          /* official host name */
    char **h_aliases;      /* array of alternative host names */
    int h_addrtype;        /* address family (not type) */
    int h_length;          /* length of each address */
    char **h_addr_list;    /* array of pointers to network addresses */
};
```

You can call `gethostent` in a loop to scan through all the known host names. You start the scan with `sethostent`, using the argument to indicate whether the connection to the host database should stay open, or whether `gethostent` should open and close it each time it's called. You call `endhostent` at the end. Thus, it sounds like a lot could be happening, but in practice all these functions are likely to do is scan the /etc/hosts file on the local machine.

There are two arrays in the `hostent` structure, each terminated with a `NULL` pointer. One, `h_aliases`, is an array of alternate host names. The other, `h_addr_list`, is an array of pointers to network addresses. For `AF_INET`, these are IP addresses stored as 32-bit binary numbers in network byte order; therefore, use `ntohl` (Section 8.1.4) to convert them to local byte order, or use `inet_ntoa` (Section 8.2.3) or `inet_ntop` (Section 8.9.5) to get them as a dotted string

Here's an example:

```
static void hostdb(void)
{
    struct hostent *h;

    sethostent(true);
    while ((h = gethostent()) != NULL)
        display_hostent(h);
    endhostent();
}

static void display_hostent(struct hostent *h)
{
    int i;

    printf("name: %s; type: %d; len: %d\n", h->h_name, h->h_addrtype,
      h->h_length);
    for (i = 0; h->h_aliases[i] != NULL; i++)
        printf("\t%s\n", h->h_aliases[i]);
    if (h->h_addrtype == AF_INET) {
        for (i = 0; h->h_addr_list[i] != NULL; i++)
            printf("\t%s\n",
              inet_ntoa(*((struct in_addr *)h->h_addr_list[i])));
    }
}
```

The output from a little main program (not shown) that called `hostdb` was:

```
name: localhost; type: 2; len: 4
        127.0.0.1
name: sol; type: 2; len: 4
        loghost
        192.168.0.10
name: bsd; type: 2; len: 4
        192.168.0.15
name: suse2; type: 2; len: 4
        suse2.MSHOME
        192.168.0.19
```

For comparison, here's what the /etc/hosts file looked like on the same system:

```
127.0.0.1       localhost
192.168.0.10    sol     loghost
192.168.0.15    bsd
192.168.0.19    suse2   suse2.MSHOME
```

To look up a host by its name, you can call `gethostbyname`. It uses DNS if such access is available, not only the /etc/hosts file. In fact, it was the primary way you got network addresses until `getaddrinfo` came along, and it's still used a lot more than `getaddrinfo` is because `gethostbyname` is very old, and that's how almost everyone learned to do things. But, unlike with `getaddrinfo`, with `gethostbyname` all you get is the IP address—you still have to build the `sockaddr_in` structure yourself, as we did in Section 8.2.3. In addition, `gethostbyname` can't handle IPv6 addresses. Here's the synopsis:

gethostbyname—look up host by name

```
#include <netdb.h>

struct hostent *gethostbyname(
    const char *nodename,      /* node name */
);
/* Returns pointer to hostent or NULL on error (sets h_errno) */
```

The parenthetical remark at the end of the synopsis is no misprint—this function really does set `h_errno`, not `errno`, and the codes it uses aren't `errno` codes, either. (I didn't code an "ec" macro for `h_errno` because I don't use the function.)

Here's an example (`display_hostent` is from the previous example):

```
static void gethostbyname_ex(void)
{
    struct hostent *h;

    if ((h = gethostbyname("www.yahoo.com")) == NULL) {
        if (h_errno == HOST_NOT_FOUND)
            printf("host not found\n");
        else
            printf("h_errno = %d\n", h_errno);
    }
    else
        display_hostent(h);
}
```

Here's the output we got from a call to this function:

```
name: www.yahoo.akadns.net; type: 2; len: 4
        www.yahoo.com
        66.218.71.95
        66.218.70.49
        66.218.71.88
        66.218.71.81
        66.218.71.86
        66.218.71.92
        66.218.71.94
        66.218.71.89
        66.218.70.48
        66.218.71.93
        66.218.70.50
        66.218.71.87
        66.218.71.84
```

You may want to compare this to the output shown in Section 8.2.6, where we did pretty much the same thing with a call to `getaddrinfo`.

The opposite of `gethostbyname` is `gethostbyaddr`, which uses the host database (perhaps DNS) to translate an IP address to a name:

gethostbyaddr—look up host by address

```
#include <netdb.h>

struct hostent *gethostbyaddr(
     const void *addr,      /* IP address */
     socklen_t len,         /* length of address */
     int family             /* family (called "type" in SUS) */
);
/* Returns pointer to hostent or NULL on error (sets h_errno) */
```

Here's an example program (`inet_addr` is in Section 8.2.3):

```
static void gethostbyaddr_ex(void)
{
    struct hostent *h;
    in_addr_t a;

    ec_neg1( a = inet_addr("66.218.71.94") )
    if ((h = gethostbyaddr(&a, sizeof(a), AF_INET)) == NULL) {
        if (h_errno == HOST_NOT_FOUND)
            printf("address not found\n");
        else
            printf("h_errno = %d\n", h_errno);
    }
    else
        display_hostent(h);
    return;
```

```
EC_CLEANUP_BGN
    EC_FLUSH("gethostbyaddr_ex")
EC_CLEANUP_END
}
```

The output:

```
name: w15.www.scd.yahoo.com; type: 2; len: 4
        66.218.71.94
```

What happened is that when we got the IP addresses for *www.yahoo.com,* 66.218.71.94 was on the list, but when we asked for that IP's name we got *w15.www.scd.yahoo.com*. That's the way DNS works—apparently Yahoo uses a lot of servers, which, of course, it must.

`gethostbyaddr` is just as obsolete as `gethostbyname`. The modern function is the counterpart to `getaddrinfo` (Section 8.2.6), called `getnameinfo`:

getnameinfo—get name information

```
#include <sys/socket.h>
#include <netdb.h>

int getnameinfo(
    const struct sockaddr *sa,     /* socket address */
    socklen_t sa_len,              /* socket-address length */
    char *nodename,                /* node name */
    socklen_t nodelen,             /* node-name buffer length */
    char *servname,                /* service name */
    socklen_t servlen,             /* service-name buffer length */
    unsigned flags                 /* flags */
);
/* Returns 0 on success or error number on error */
```

`getnameinfo` takes a whole socket address, not just a bare IP address, because it potentially works with lots of different families. Most importantly, it works with IPv6 addresses, which `gethostbyname` doesn't. The answers come back into two buffers: `nodename`, of length `nodelen`, for the node (or host) name, and `servname`, of length `servlen`, for the service name. If a buffer pointer is `NULL` or its length is 0, you don't get that information back.

Like `getaddrinfo`, the error numbers returned by `getnameinfo` aren't `errno` codes, but special "EAI" codes, and you can use the function `gai_strerror` (Section 8.2.6) on them if you want. Or, use the `ec_ai` macro that we've provided just for these two functions.

Various flags that you can OR together control what `getnameinfo` returns:

NI_NOFQDN	Return just the node name part of local hosts, not the fully qualified domain name.
NI_NUMERICHOST	Return the numeric form (dotted notation for IPv4, colon notation for IPv6) of the address instead of the name.
NI_NAMEREQD	Return an error if the host's name can't be found. Normally the numeric form is returned in this case.
NI_NUMERICSERV	Return the port number (as a string) instead of the service name.
NI_DGRAM	Look for a SOCK_DGRAM (UDP) service. Normally it looks for a SOCK_STREAM (TCP) service.

Here's an example:

```
static void getnameinfo_ex(void)
{
    struct sockaddr_in sa;
    char nodename[200], servname[200];

    sa.sin_family = AF_INET;
    sa.sin_port = htons(80);
    sa.sin_addr.s_addr =  inet_addr("216.109.125.70");
    ec_ai( getnameinfo((struct sockaddr *)&sa, sizeof(sa), nodename,
      sizeof(nodename), servname, sizeof(servname), 0) )
    printf("node: %s; service: %s\n", nodename, servname);
    return;

EC_CLEANUP_BGN
    EC_FLUSH("getnameinfo_ex")
EC_CLEANUP_END
}
```

The output:

```
node: w17.www.dcn.yahoo.com; service: http
```

Another host-oriented function to talk about is:

gethostid—get identifier for local host

```
#include <unistd.h>

long gethostid(void);
/* Returns identifier (no error return) */
```

`gethostid` looks like it might return the local machine's IP, but it doesn't necessarily do that. All it's required to do is return a unique identifier, and whether that even works probably depends on whether such an identifier has been set at boot time. I've documented it here just so you don't accidentally confuse it with a useful function.

There's another function, not directly associated with networking, that provides information to identify the system:

uname—get info about current system

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

int uname(
    struct utsname *info      /* returned info */
);
/* Returns non-negative value on success or -1 on error (sets errno) */
```

struct utsname—structure for uname

```
struct utsname {
    char sysname[];           /* OS name */
    char nodename[];          /* node name within network */
    char release[];           /* release number (as string) */
    char version[];           /* version number (as string) */
    char machine[];           /* hardware type or computer model */
};
```

Unfortunately, none of the members of the `utsname` structure[8] are standardized, so you can't use them to control your application's processing. It would be nice, say, to test the machine member to see if you're on an Intel CPU; however, each system that runs on that CPU formats the string the way it wants, and the word "Intel" or "x86" might not even be there. What you can do is use the strings for display purposes—maybe to label performance-testing output. And, naturally, the `uname` system call is the guts of the `uname` command, our version (lacking options) of which is:

```
int main(void)
{
    struct utsname info;

    ec_neg1( uname(&info) )
    printf("sysname = %s\n", info.sysname);
    printf("nodename = %s\n", info.nodename);
```

8. Empty brackets as shown in the synopsis aren't legal C, but you get the idea. Each implementation allocates whatever space it needs.

```
    printf("release = %s\n", info.release);
    printf("version = %s\n", info.version);
    printf("machine = %s\n", info.machine);
    exit(EXIT_SUCCESS);

EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
EC_CLEANUP_END
}
```

Here's the output[9] from running this program on our four test systems; see if you can spot any patterns:

```
sysname = SunOS
nodename = sol
release = 5.8
version = Generic_108529-13
machine = i86pc

sysname = Darwin
nodename = Marc-Rochkinds-Computer.local.
release = 6.6
version = Darwin Kernel Version 6.6: Thu May  1 21:48:54 PDT 2003;
  root:xnu/xnu-344.34.obj~1/RELEASE_PPC
machine = Power Macintosh

sysname = FreeBSD
nodename = bsd.MSHOME
release = 4.6-RELEASE
version = FreeBSD 4.6-RELEASE #0: Tue Jun
machine = i386

sysname = Linux
nodename = suse2
release = 2.4.18-4GB
version = #1 Wed Mar 27 13:57:05 UTC 2002
machine = i686
```

8.8.2 Network Functions

Functions in this category get information about networks that the local machine might be connected to. First come three functions like the ones for hosts (e.g., `gethostent`) for scanning for networks:

9. Output was folded to fit within the page margins.

setnetent—start network-database scan

```
#include <netdb.h>

void setnetent(
     int stayopen           /* leave connection open? */
);
```

getnetent—get network-database entry

```
#include <netdb.h>

struct netent *getnetent(void);
/* Returns pointer to netent or NULL on end (errno not defined) */
```

endnetent—end network-database scan

```
#include <netdb.h>

void endnetent(void);
```

struct netent—structure for network-database functions

```
struct netent {
     char *n_name;          /* official network name */
     char **n_aliases;      /* array of alternative network names */
     int n_addrtype;        /* address family (not type) */
     uint32_t n_net;        /* network number */
};
```

Here's an example:

```
static void netdb(void)
{
    struct netent *n;

    setnetent(true);
    while ((n = getnetent()) != NULL)
        display_netent(n);
    endnetent();
}

static void display_netent(struct netent *n)
{
    int i;

    printf("name: %s; type: %d; number: %lu\n", n->n_name, n->n_addrtype,
      (unsigned long)n->n_net);
    for (i = 0; n->n_aliases[i] != NULL; i++)
        printf("\t%s\n", n->n_aliases[i]);
}
```

With this output:

```
name: loopback; type: 2; number: 127
name: arpanet; type: 2; number: 10
        arpa
```

Most, if not all, machines have a loopback network for testing. This machine (running Solaris) also has a connection to an Internet network that it calls "arpanet" in honor of its history. Remember that the "type" displayed is really the family; it turns out that the macro `AF_INET` is defined as 2.

Analogously to the host database, there are functions for finding an entry by network number or by name:

getnetbyname—look up network by name

```
#include <netdb.h>

struct netent *getnetbyname(
     const char *name       /* network name (to match n_name member) */
);
/* Returns pointer to netent or NULL if not found (errno not defined) */
```

getnetbyaddr—look up network by number

```
#include <netdb.h>

struct netent *getnetbyaddr(
     uint32_t net,          /* network number (to match n_net member) */
     int type               /* family (to match n_addrtype member) */
);
/* Returns pointer to netent or NULL if not found (errno not defined) */
```

8.8.3 Protocol Functions

Following the pattern, the functions for scanning the protocol database are:

setprotoent—start protocol-database scan

```
#include <netdb.h>

void setprotoent(
     int stayopen           /* leave connection open? */
);
```

getprotoent—get protocol-database entry

```
#include <netdb.h>

struct protoent *getprotoent(void);
/* Returns pointer to protoent or NULL on end (errno not defined) */
```

endprotoent—end protocol-database scan

```
#include <netdb.h>

void endprotoent(void);
```

struct protoent—structure for protocol-database functions

```
struct protoent {
     char *p_name;          /* official protocol name */
     char **p_aliases;      /* array of alternative protocol names */
     int p_proto;           /* protocol number */
};
```

On some systems, you can use a protocol number for a protocol level when you call `setsockopt` or `getsockopt`, but that's nonstandard.

The example code is pretty obvious if you've been reading along:

```
static void protodb(void)
{
    struct protoent *p;

    setprotoent(true);
    while ((p = getprotoent()) != NULL)
        display_protoent(p);
    endprotoent();
}

static void display_protoent(struct protoent *p)
{
    int i;

    printf("name: %s; number: %d\n", p->p_name, p->p_proto);
    for (i = 0; p->p_aliases[i] != NULL; i++)
        printf("\t%s\n", p->p_aliases[i]);
}
```

But the output is pretty interesting. On SuSE Linux I got 135 protocols. Here's just part of the output:

```
...
name: mobile; number: 55
        MOBILE
name: tlsp; number: 56
        TLSP
name: skip; number: 57
        SKIP
name: ipv6-icmp; number: 58
        IPv6-ICMP
```

```
        ICMPV6
        icmpv6
        icmp6
name: ipv6-nonxt; number: 59
        IPv6-NoNxt
name: ipv6-opts; number: 60
        IPv6-Opts
name: cftp; number: 62
        CFTP
name: sat-expak; number: 64
        SAT-EXPAK
name: kryptolan; number: 65
        KRYPTOLAN
...
```

There are the two more predictable functions to round out the set:

getprotobyname—look up protocol by name

```
#include <netdb.h>

struct protoent *getprotobyname(
    const char *name          /* protocol name */
);
/* Returns pointer to protoent or NULL if not found (errno not defined) */
```

getprotobynumber—look up protocol by number

```
#include <netdb.h>

struct protoent *getprotobynumber(
    int proto                 /* protocol number */
);
/* Returns pointer to protoent or NULL if not found (errno not defined) */
```

8.8.4 Service Functions

There's one more set of scanning and get-by-name and number functions, this time for services. These scan the local system's /etc/services file.

setservent—start service-database scan

```
#include <netdb.h>

void setservent(
    int stayopen              /* leave connection open? */
);
```

getservent—get service-database entry

```
#include <netdb.h>

struct servent *getservent(void);
/* Returns pointer to servent or NULL on end (errno not defined) */
```

endservent—end service-database scan

```
#include <netdb.h>

void endservent(void);
```

struct servent—structure for service-database functions

```
struct servent {
     char *s_name;          /* official service name */
     char **s_aliases;      /* array of alternative service names */
     int s_port;            /* port number */
     char *s_proto;         /* name of protocol for this service */
};
```

getservbyname—look up service by name

```
#include <netdb.h>

struct servent *getservbyname(
     const char *name,      /* service name */
     const char *proto      /* protocol name */
);
/* Returns pointer to protoent or NULL if not found (errno not defined) */
```

getservbyport—look up service by port

```
#include <netdb.h>

struct servent *getservbyport(
     int port,              /* port */
     const char *proto      /* protocol name */
);
/* Returns pointer to protoent or NULL if not found (errno not defined) */
```

And our last example of this ilk is:

```
static void servdb(void)
{
    struct servent *s;

    setservent(true);
    while ((s = getservent()) != NULL)
        display_servent(s);
    endservent();
}

static void display_servent(struct servent *s)
{
    int i;
```

```
    printf("name: %s; port: %d; protocol: %s\n", s->s_name,
       ntohs(s->s_port), s->s_proto);
    for (i = 0; s->s_aliases[i] != NULL; i++)
        printf("\t%s\n", s->s_aliases[i]);
}
```

The output is huge because /etc/services files are typically huge. Here's a fragment:

```
...
name: ftp-data; port: 20; protocol: tcp
name: ftp-data; port: 20; protocol: udp
name: ftp; port: 21; protocol: tcp
name: fsp; port: 21; protocol: udp
name: ssh; port: 22; protocol: tcp
name: ssh; port: 22; protocol: udp
name: telnet; port: 23; protocol: tcp
name: telnet; port: 23; protocol: udp
name: smtp; port: 25; protocol: tcp
        mail
name: smtp; port: 25; protocol: udp
        mail
name: nsw-fe; port: 27; protocol: tcp
name: nsw-fe; port: 27; protocol: udp
name: msg-icp; port: 29; protocol: tcp
name: msg-icp; port: 29; protocol: udp
name: msg-auth; port: 31; protocol: tcp
name: msg-auth; port: 31; protocol: udp
...
```

8.8.5 Network Interface Functions

There's a set of functions to retrieve the names of the network interfaces and their index numbers. First, here's a function that gets them all as an array of `if_nameindex` structures and a corresponding function to free the array:

if_nameindex—get all network interface names and indexes

```
#include <net/if.h>

struct if_nameindex *if_nameindex(void);
/* Returns array or NULL on error (sets errno) */
```

if_freenameindex—free array allocated by if_nameindex

```
#include <net/if.h>

void if_freenameindex(
     struct if_nameindex *ptr   /* pointer to array */
);
```

struct if_nameindex—structure for network-interface functions

```
struct if_nameindex {
    unsigned if_index;     /* interface index */
    char *if_name;         /* interface name */
};
```

Here's a function that displays the indexes and interfaces:

```
static void ifdb(void)
{
    struct if_nameindex *ni;
    int i;

    ec_null( ni = if_nameindex() )
    for (i = 0; ni[i].if_index != 0 || ni[i].if_name != NULL; i++)
        printf("index: %d; name: %s\n", ni[i].if_index, ni[i].if_name);
    if_freenameindex(ni);
    return;

EC_CLEANUP_BGN
    EC_FLUSH("ifdb")
EC_CLEANUP_END
}
```

The output on Solaris was:

```
index: 1; name: lo0
index: 2; name: iprb0
```

and on SuSE Linux:

```
index: 1; name: lo
index: 2; name: eth0
```

These sort of match up with the network names displayed by the functions in Section 8.8.2: a loopback interface, and an Internet interface using Ethernet.

There's another function to map a name to its index:

if_nametoindex—map network interface name to index

```
#include <net/if.h>

unsigned if_nametoindex(
    const char *ifname     /* interface name */
);
/* Returns index or 0 on error (errno not defined) */
```

Be careful—this function returns 0 if the name isn't found, not –1.

To map an index to a name, you call:

if_indextoname—map network interface index to name

```
#include <net/if.h>

char *if_indextoname(
     unsigned ifindex,      /* interface index */
     char *ifname           /* interface name */
);
/* Returns name or NULL on error (sets errno) */
```

The `ifname` argument must be a buffer of at least `IF_NAMESIZE` bytes, which includes space for the terminating NUL byte. It also returns a pointer to that buffer.

8.9 Miscellaneous System Calls

This section describes some networking system calls that don't fit into the previous sections.

8.9.1 `send` and `recv`

These two functions behave exactly like `write` and `read`, except they allow you to specify flags. Because they don't have a socket address argument, they're normally used with connected sockets. For connectionless sockets, `sendto`, `sendmsg`, `recvfrom`, and `recvmsg` are more convenient.

send—send data to socket

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

ssize_t send(
     int socket_fd,         /* socket file descriptor */
     const void *data,      /* data to send */
     size_t length,         /* length of data */
     int flags              /* flags */
);
/* Returns number of bytes sent or -1 on error (sets errno) */
```

recv—receive data from socket

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

ssize_t recv(
     int socket_fd,         /* socket file descriptor */
     void *buffer,          /* buffer to receive data */
     size_t length,         /* length of buffer */
     int flags              /* flags */
);
/* Returns number of bytes received, 0, or -1 on error (sets errno) */
```

You can use the `MSG_OOB` flag with both `send` and `recv`; it was explained in Section 8.7. In addition, for `recv`, you can specify `MSG_PEEK` and/or `MSG_WAITALL`, which were explained in Section 8.6.2. We could have used `MSG_WAITALL` with `recv` instead of calling `readall` in the SMI implementation in Section 8.5.

8.9.2 `getsockname` and `getpeername`

`getsockname` gets the socket address that a socket has been bound to with a previous call to `bind`:

getsockname—get socket address

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

int getsockname(
     int socket_fd,         /* socket file descriptor */
     struct sockaddr *sa,   /* socket address */
     socklen_t *sa_len      /* address length */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

As usual for socket-address-returning functions, on input `sa` should point to a buffer big enough for the socket address and `sa_len` should be the size of that buffer. On output `sa_len` is the actual size of the address.

A similar function gets the socket address of the socket connected to a socket:

getpeername—get socket address of connected socket

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

int getpeername(
     int socket_fd,         /* socket file descriptor */
     struct sockaddr *sa,   /* socket address */
     socklen_t *sa_len      /* address length */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

If the connected socket is unbound, as is often the case, then what `getpeername` returns is unspecified. You don't get an error return in this case.

8.9.3 socketpair

AF_UNIX sockets act something like bidirectional FIFOs, and, as with FIFOs, each process that wants to communicate sets up its own socket. By contrast, the pipe system call returns two file descriptors that a child process can inherit. The socketpair system call sort of blends the two approaches—you get two socket file descriptors with one call:

socketpair—create pair of sockets

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

int socketpair(
     int domain,            /* domain (AF_UNIX, AF_INET, etc.) */
     int type,              /* SOCK_STREAM, SOCK_DGRAM, etc. */
     int protocol           /* specific to type; usually zero */
     int socket_vector[2]   /* returned socket file descriptors */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

The first three arguments are just like the ones for socket. What domains and types are handled is up to the implementation, and typically all that's supported is AF_UNIX and SOCK_STREAM.

8.9.4 shutdown

When you close a connected socket that still has data to be sent or received, the system keeps trying for a while until it gives up and discards the data. You can indicate before you close the socket that you don't want the data by calling shutdown:

shutdown—shut down socket send and/or receive operations

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

int shutdown(
     int socket_fd,         /* socket file descriptor */
     int how                /* SHUT_RD, SHUT_WR, or SHUT_RDWR */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

SHUT_RD disables further receives, SHUT_WR disables sends, and SHUT_RDWR disables both.

8.9.5 `inet_ntop` and `inet_pton`

The `inet_ntoa` and `inet_addr` functions discussed in Section 8.2.3 convert between binary IPv4 addresses and dotted strings. The following two functions, `inet_ntop` and `inet_pton` are smarter: They work with both IPv4 and IPv6 addresses.

inet_ntop—convert IPv4 or IPv6 binary address to string

```
#include <arpa/inet.h>

const char *inet_ntop(
    int domain,            /* AF_INET or AF_INET6 */
    const void *src,       /* pointer to binary address (input) */
    char *dst,             /* string (output) */
    socklen_t dst_len      /* length of dst buffer */
);
/* Returns string or NULL on error (sets errno) */
```

inet_pton—convert IPv4 or IPv6 string address to binary

```
#include <arpa/inet.h>

int inet_pton(
    int domain,            /* AF_INET or AF_INET6 */
    const char *src,       /* string (input) */
    void *dst              /* buffer for binary address (output) */
);
/* Returns 1 on success, 0 on bad string, or -1 on error (sets errno) */
```

You call `inet_ntop` with `src` pointing to the binary address, either a 32-bit number for IPv4 or a 16-byte array for IPv6. The result is placed in the buffer pointed to by `dst`. Two macros are available for sizing `dst`: `INET_ADDRSTRLEN` for IPv4, and `INET6_ADDRSTRLEN` for IPv6.

For `inet_pton`, the input is a string in dotted or colon notation, and the binary output goes into whatever `dst` points to, which better be big enough, because there's no length argument. You need 32 bits for IPv4 addresses and 128 bits (16 bytes) for IPv6.

Here's an example:

```
static void cvt(void)
{
    char ipv6[16], ipv6str[INET6_ADDRSTRLEN], ipv4str[INET_ADDRSTRLEN];
    uint32_t ipv4;
    int r;
```

```
    ec_neg1( r = inet_pton(AF_INET, "66.218.71.94", &ipv4) )
    if (r == 0)
        printf("Can't convert\n");
    else {
        ec_null( inet_ntop(AF_INET, &ipv4, ipv4str, sizeof(ipv4str)) )
        printf("%s\n", ipv4str);
    }
    ec_neg1( r = inet_pton(AF_INET6,
      "FEDC:BA98:7654:3210:FEDC:BA98:7654:3210", &ipv6) )
    if (r == 0)
        printf("Can't convert\n");
    else {
        ec_null( inet_ntop(AF_INET6, &ipv6, ipv6str, sizeof(ipv6str)) )
        printf("%s\n", ipv6str);
    }
    return;

EC_CLEANUP_BGN
    EC_FLUSH("cvt")
EC_CLEANUP_END
}
```

The output from calling this function:

```
66.218.71.94
fedc:ba98:7654:3210:fedc:ba98:7654:3210
```

8.10 High-Performance Considerations

I showed an example of a Web server in Section 8.4.4, using the multiple-client approach from Section 8.1.3. Imagine now that our Web server is trying to handle 1000 clients at once. Or 10,000. Will it work?

Let's make a list of what the so-called "C10K" ("clients 10,000") problems might be with our simple example:

- `select` has to look at up to 10,001 file descriptors. That's probably bigger than an `fd_set` can hold. Even if it can hold that number, it would take `select` a long time to process them all.
- There's a limit on how many file descriptors a process can have open, and it's usually much less than 10,001.
- To provide decent response, we'd want to engage multiple processes or multiple threads to handle all the work, but there's a limit on those resources, too.

- All the copying of data from files to sockets (in and out of the process's memory) is very time consuming. It would be a killer for 10,000 clients.
- Lots of other internal tables and other resources that our server might use start running out of space, or slowing way down, or both.

So, as you can see, the problems get pretty serious. This is one reason why a real Web server (or any other large-capacity server) is a hugely complex piece of software.

If you're trying to handle 10,000 clients, or even 500, there's a terrific article titled "The C10K Problem" that explains the problem and surveys most of the possible solutions [Keg2003].

Exercises

8.1. Modify the multiple-client example in Section 8.1.3 to work between computers (`AF_INET`). Run the server on one computer and run clients on at least one other computer.

8.2. Based on the information in [RFC1288], implement a simple version of the `finger` command, without options other than the `user@host` argument. Use connected sockets (`SOCK_STREAM`). For the next Exercise, you might need an option to set the port number.

8.3. Also based on [RFC1288], implement a simple `finger` server, but use an unused high-numbered port (e.g., 3079) instead of the standard port 79. Use connected sockets. Test your server with the `finger` command you wrote in the previous Exercise. If you have a machine you can play with, install your server on port 79 and test it with a standard `finger` command running on another machine.

8.4. Same as Exercise 8.2, but with connectionless sockets.

8.5. Same as Exercise 8.3, but with connectionless sockets.

8.6. Design and try to implement a simple communication example based on RFC 2549, which you can find at [RFC]. You may restrict yourself to the Concorde and First quality-of-service levels.

8.7. Implement a graphical Web browser using a GUI toolkit such as Qt. Warning: This is really hard! You may want to do this as a semester-long group project. Hint: Start your design with a way to handle nested tables. If you get that part right, everything else is relatively easy.

8.8. Implement the SMI with connectionless sockets, instead of with connected sockets as we did in Section 8.5. Run some timing tests like those in Section 7.15.

8.9. Implement a web crawler that starts with some URL and then scans what it finds there to discover additional URLs (e.g., by looking for strings that start with "href"). It adds the discovered URLs to a list, and then scans them, discovering even more URLs. It stops when it's found some specified number of URLs or when it's interrupted. (Obviously, it needs to continue when it encounters an error.) There should be a way to display the results—at least all the URLs discovered and whether they were scanned successfully (HTTP status of 200) or not and, if not, what the reason was. You *must* implement the Robot Exclusion Protocol (see *www.robotstxt.org* for details) so you don't crawl onto pages that you've been requested to stay away from. Provide a "unique host" option that crawls each host (the part of the URL up to the first slash) only once, using whatever URL was tried first for that host (e.g., *www.basepath.com/index.htm*). Using that option, try to find a starting URL that yields the most unique hosts that are successfully scanned. (I thought starting with *www.yahoo.com* would be a good idea, but I got exactly one hit because all the links there were relative to *www.yahoo.com*.) Be nice to others when you run your crawler by making sure that you're not hogging network resources.

8.10. Extend the program you wrote in Exercise 5.14 to include process attributes from Appendix A that are explained in this chapter.

This page intentionally left blank

9

Signals and Timers

9.1 Signal Basics

A *signal* is a notification that an event has occurred, such as a user typing an interrupt (Ctrl-c, normally), a floating-point exception, or an alarm going off. Usually, a signal is delivered to a process or thread asynchronously and whatever the process or thread is doing is interrupted. The signal might immediately terminate the process, or, by prearrangement, a function designated to catch it might be executed.

9.1.1 Introduction to Signals

To show how a program handles a signal, here's a simple example of a program that displays a number once every three seconds, but when an interrupt signal occurs, it displays a message and terminates:

```
static void fcn(int signum),
{
    (void)write(STDOUT_FILENO, "Got signal\n", 11);
    _exit(EXIT_FAILURE);
}

int main(void)
{
    int i;
    struct sigaction act;

    memset(&act, 0, sizeof(act));
    act.sa_handler = fcn;
    ec_neg1( sigaction(SIGINT, &act, NULL) )

    for (i = 1; ; i++) {
        sleep(3);
        printf("%d\n", i);
    }
```

```
    exit(EXIT_SUCCESS);

EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
EC_CLEANUP_END
}
```

The call to `sigaction` installed a signal-handling function for the `SIGINT` signal. When I ran this program, I typed a Ctrl-c after a "2" appeared. That caused whatever the program was doing (perhaps sleeping, or inside `printf`, or just looping) to be interrupted and the function `fcn` to be called immediately. It displayed a message and terminated the program with a call to `_exit`. (See Section 9.1.7 for why I didn't use `exit`.)

The output on the screen was:

```
1
2
Got signal
```

If I hadn't installed the signal handler, typing Ctrl-c would have terminated the process right away. Technically, the reason is that the default action for the `SIGINT` signal is to terminate the process.

I also could have called `sigaction` to arrange for `SIGINT` signals to be ignored:

```
int main(void)
{
    int i;
    struct sigaction act;

    memset(&act, 0, sizeof(act));
    act.sa_handler = SIG_IGN;
    ec_neg1( sigaction(SIGINT, &act, NULL) )

    for (i = 1; ; i++) {
        sleep(3);
        printf("%d\n", i);
    }
    exit(EXIT_SUCCESS);

EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
EC_CLEANUP_END
}
```

This time it kept displaying numbers, and typing Ctrl-c had no effect. We finally terminated it by typing Ctrl-\, which generated a quit signal (`SIGQUIT`), whose default behavior, which we didn't modify, is to terminate the process.

Well, so much for the basics. Signals are actually pretty complicated because:

- There are lots of different signals, and sometimes the circumstances under which they're generated and what they mean are complicated.
- Arranging the appropriate action for a signal can be complicated.
- Handling a signal can be complicated.

I'll go through the issues step-by-step, and when you've finished this chapter all will be clear.

9.1.2 A Signal's Lifecycle

A signal is born when whatever event it's associated with occurs and *generates* it. Its life ends when it is *delivered,* which means that whatever *action* specified for it has been taken. There are three possible actions:

1. Default action (`SIG_DFL`): terminate, stop, or continue the process, or ignore the signal.
2. Ignore the signal (`SIG_IGN`).
3. Catch the signal by executing a signal handler when the signal is delivered.

Most signals can be generated *naturally* by whatever user or system event the signal is associated with. For example, dividing by zero generates a `SIGFPE` event naturally, and termination of a child generates a `SIGCHLD` event naturally. Alternatively, any signal can be generated *synthetically* by one of five system calls: `kill`, `killpg`, `pthread_kill`, `raise`, or `sigqueue`. The next section, which enumerates all the signals, indicates the natural cause of each signal that has one. Five signals, `SIGKILL`, `SIGSTOP`, `SIGTERM`, `SIGUSR1`, and `SIGUSR2` have no natural cause and exist only to be synthesized, usually with the `kill` system call.

Between generation and delivery a signal is *pending*. If another signal of the same type arrives while a signal is pending, whether multiple signals of the same type are delivered is implementation dependent and should not be assumed in portable programs. There's more on this subject in Section 9.5.3.

A thread (Section 5.17) can keep pending signals in the pending state by *blocking* them.[1] The set of all currently blocked signals is called the *signal mask*. There are various system calls, which I'll explain in Section 9.1.5, that can be used to create and manipulate masks and make a particular mask the effective signal mask for the thread.

A signal is either generated for a specific thread or for an entire process. In the latter case, if there's more than one thread that has it unblocked, which thread it's delivered to is undefined. Section 9.1.3 gives more information about which events are sent to a thread vs. a process and under what circumstances.[2]

The *action* for a signal is process-wide, even if there are multiple threads, although, as I said, the signal mask is per-thread.

Normally, when a signal handler is executed, the signal it's handling is temporarily added to that thread's signal mask so that a second signal of that type won't be delivered until the handler returns. That way you don't have to worry about a recursive call to a handler for the same signal. However, a recursive call is possible if you're using the same function for several different signal types.

9.1.3 Types of Signals

There are 28 signals defined in [SUS2002], and most implementations define a few more that you can look up but which I won't describe here. Also, there are additional signals that are part of the Realtime Signals Extension (Section 9.5). It's useful to consider the SUS signals as falling into a few groups. In the following list, letters in parentheses indicate the default action for the signal, which is explained below. The natural cause for each signal is indicated; the 5 synthetic-only signals (explained in the previous section) are indicated explicitly. Remember that all signals with a natural cause can be also generated synthetically.

- Detected errors:
  `SIGBUS` – access to undefined portion of a memory object (A)
  `SIGFPE` – erroneous arithmetic operation (A)
  `SIGILL` – illegal instruction (A)
  `SIGPIPE` – write on a pipe with no reader (T)

1. If there's only one thread in the process, perhaps because it's not doing any multithreading, then whatever I say about a thread applies to the process as a whole.
2. What I say about threads in this chapter applies only to implementations of POSIX Threads. Some implementations of "Linux Threads," widely used on Linux and FreeBSD systems, are not faithful POSIX Threads implementations, and signals don't work with them in the standard way. The newest version now being released, NPTL, works fine, however.

`SIGSEGV` – invalid memory reference (A)
`SIGSYS` – bad system call (A)
`SIGXCPU` – CPU-time limit exceeded (A)
`SIGXFSZ` – file-size limit exceeded (A)

- User/application-generated:
  `SIGABRT` – call to `abort` (A)
  `SIGHUP` – hangup (T)
  `SIGINT` – interrupt (from keyboard) (T)
  `SIGKILL` – kill; synthetic only (T)
  `SIGQUIT` – quit (from keyboard) (A)
  `SIGTERM` – termination; synthetic only (T)
  `SIGUSR1` – user signal 1; synthetic only (T)
  `SIGUSR2` – user signal 2; synthetic only (T)
- Job control:
  `SIGCHLD` – child process terminated or stopped (I)
  `SIGCONT` – continue executing (from keyboard) (C)
  `SIGSTOP` – stop executing; synthetic only (S)
  `SIGTSTP` – terminal stop signal (from keyboard) (S)
  `SIGTTIN` – background process attempting read (S)
  `SIGTTOU` – background process attempting write (S)
- Timer:
  `SIGALRM` – alarm clock expired (T)
  `SIGVTALRM` – virtual timer expired (T)
  `SIGPROF` – profiling timer expired (T)
- Miscellaneous events:
  `SIGPOLL` – pollable event (T)
  `SIGTRAP` – trace/breakpoint trap (A)
  `SIGURG` – out-of-band data available at socket (I)

Here's what the parenthesized letters (default actions) mean:

I signal is ignored

T termination

A same as T, but with additional implementation-defined actions, such as writing of a core-dump file

S stop

C continue after stop

Natural generation of a detected-error signal results from a program error. For SIGBUS, SIGFPE, SIGILL, and SIGSEGV, the exact cause of the error isn't standardized, but it's usually something detected by the hardware. Also, these four signals are subject to some special rules when they arise naturally:

- If they were set to be ignored by sigaction, their behavior is undefined.
- The result of a signal-catching function returning is undefined.
- The result of one of them occurring while blocked is undefined.

Another way to say this is that if the hardware-detected error is real, your program won't necessarily get past it. It's not safe to ignore it, to continue processing after a signal-handler returns, or to postpone the action by blocking it. You deal with it right away in a signal handler that must exit (or long-jump; see Section 9.6), rather than return, or the process is immediately terminated, which is the default action.

Two of the user/application-generated signals, SIGINT and SIGQUIT, are normally associated with keyboard control sequences, as explained in Section 4.5.7. SIGHUP normally results from hanging up a terminal device. SIGABRT is generated by the abort system call (Section 9.1.9), and SIGTERM is the default signal for the kill command—it's the primary way that arbitrary processes are terminated when, for example, the system administrator needs to shut down the system. SIGUSR1 and SIGUSR2 aren't used by any system call and are available for application use.

The job control signals were discussed in Section 4.3.

SIGALRM is discussed in Section 9.7.1. The other two timer signals are discussed in Section 9.7.4.

Among the miscellaneous-event signals, SIGPOLL can be used with STREAMS (Section 4.9); it's enabled with a call to ioctl. Normally, it's not generated, so you don't need to worry about it. SIGURG was explained in Section 8.7. SIGTRAP is used by debuggers.

When a signal is delivered, you can't tell whether it was generated naturally or synthetically. When one of the detected-error signals is generated naturally, it's sent to the offending thread; the other signals are sent to the process. A synthetically generated signal can be sent to the process or to a thread, depending on which system call was used (Section 9.1.9) to generate it.

Programs that need to clean up before terminating should arrange to catch signals `SIGHUP`, `SIGINT`, and `SIGTERM`. Until the program is solid, `SIGQUIT` should be left alone so there will be a way to terminate the program (with a core dump) from the keyboard. Arrangements for the other signals are made much less often; usually they are left to terminate the process. But a really polished program will want to catch everything it can, to clean up, possibly log the error, and print a nice error message. Psychologically, a message like "Internal error 53: contact customer support" is more acceptable than the message "Bus error-core dumped" from the shell. See Section 9.1.8 for more on this subject and an example program.

9.1.4 Interrupted System Calls

The delivery of a nonignored signal can cause a system call to be interrupted. If the action results in the termination of the process, either because that was the default action for the signal or because the signal-handler terminated the process, as in the examples we showed above, the interrupted system call is never resumed. If the action is to stop the process, execution picks up when the process is continued.

However, if the action is to catch the signal and the signal handler returns, the interrupted system is normally not restarted. Instead, it usually returns –1 with `errno` set to `EINTR`. In some cases that's exactly what you want; for example, you might deliberately set an alarm to generate an `SIGALRM` signal to interrupt a waiting `read` after, say, 10 seconds. But in other cases an interrupted system call causes problems because the algorithm doesn't allow for an interrupted call.

The simplest rule to follow is to never return from a signal handler unless you've carefully controlled the context in which the signal occurs. If that's not practical, you can set the `SA_RESTART` flag (Section 9.1.6) when you call `sigaction` for the signal so that system calls won't be interrupted—they will instead resume where they left off when the signal handler returns.

Only system calls that block can be interrupted. In this context, "blocked" means that the call is waiting for some event whose arrival can't be predicted, such as input from a terminal or socket, termination of a process, arrival of a message, posting of a semaphore, and so on. System calls that merely take some time, such as reading a file or creating a process, are not blocking; although there is a short delay, it's spent processing or waiting for a processor, not waiting for some unpredictable event.

Not every system call that blocks is interruptible. An example is `pthread_mutex_lock`, which continues to wait even if a signal arrives and its handler returns. The only way to know for sure that a system call is interruptible is to read its documentation, preferably in the SUS.

9.1.5 Managing the Signal Mask

As with an `fd_set` used by `select` (Section 4.2.3), signal masks have a collection of functions for manipulating the various bits:[3]

sigemptyset—initialize empty signal set

```
#include <signal.h>

int sigemptyset(
    sigset_t *set          /* signal set */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

sigfillset—initialize full signal set

```
#include <signal.h>

int sigfillset(
    sigset_t *set          /* signal set */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

sigaddset—add signal to signal set

```
#include <signal.h>

int sigaddset(
    sigset_t *set,         /* signal set */
    int signum             /* signal */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

sigdelset—delete signal from signal set

```
#include <signal.h>

int sigdelset(
    sigset_t *set,         /* signal set */
    int signum             /* signal */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

3. An alternative would be to just use an `unsigned long`, but at the time these functions were introduced `long`s were 32 bits on nearly all machines (`long long` hadn't yet been introduced), and 32 was considered to be too small.

sigismember—test for signal in signal set

```
#include <signal.h>

int sigismember(
     const sigset_t *set,       /* signal set */
     int signum                 /* signal */
);
/* Returns 1 if member, 0 if not, or -1 on error (sets errno) */
```

Given a `sigset_t`, you start with `sigemptyset` or `sigfillset`, and then you call `sigaddset` or `sigdelset` to add or delete members. You can call `sigismember` to test whether a signal is a member.

There's only one signal mask at a time for a thread, and you set it with `pthread_sigmask`:

pthread_sigmask—change thread's signal mask

```
int pthread_sigmask(
     int how,                   /* how signal mask is to be changed */
     const sigset_t *set,       /* input set */
     sigset_t *oset             /* previous signal mask */
);
/* Returns 0 on success, error number on failure (errno not set) */
```

How the signal mask is changed by the input `set` is determined by the `how` argument:

`SIG_BLOCK` New signal mask becomes union of current signal mask and `set`.

`SIG_SETMASK` New signal mask becomes `set`, entirely replacing current signal mask.

`SIG_UNBLOCK` New signal mask becomes union of current signal mask and complement of `set`.

In other words, `SIG_BLOCK` adds the signals in the `set` argument to the signal mask, `SIG_UNBLOCK` removes the signals in `set` from the signal mask, and `SIG_SETMASK` just sets the signal mask to `set`.

If `oset` isn't `NULL`, the previous signal mask is returned to what `oset` points to. Also, `set` can be `NULL`, in which case the signal mask isn't changed (regardless of `how`) but is returned through `oset` (if it isn't also `NULL`); this is a way of just getting the signal mask without changing it.

You can't block the SIGKILL or SIGSTOP signals, as they're always delivered to a process and always terminate or stop the process. (They can't be caught or ignored either.)

If there's only one thread in the process, you have the option of calling an older function (dating from before POSIX Threads were introduced) that works identically to pthread_sigmask, except that it uses errno instead of returning the error code:

sigprocmask—change thread's signal mask (single thread only)

```
#include <signal.h>

int sigprocmask(
     int how,               /* how signal mask is to be changed */
     const sigset_t *set,   /* input set */
     sigset_t *oset         /* previous signal mask */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

Often, programs not using threads at all aren't linked with the "pthread" library, so pthread_sigmask isn't available and you have no choice but to use sigprocmask.[4]

You don't block a signal with the signal mask because you want your process to ignore it—that's what the SIG_IGN action is for. Instead, blocking is a temporary state that's used to protect some part of your code from the arrival of a signal. One example I already mentioned: When a signal handler is executing, the signal that caused it to be called is temporarily and automatically added to the signal mask and then removed when (and if) the function returns.[5]

Another example is when your application starts up, before it has a chance to set the action for all the signals it cares about. All it takes is a call to sigfillset and a call to pthread_sigmask (or sigprocmask) to get temporary relief.

There are other examples where signal masks are important throughout this chapter.

4. On my version of Solaris, pthread_sigmask was defined even without the pthread library, but it didn't do anything. I had to change it to sigprocmask.

5. If you jump out of a signal handler with longjmp, what happens to the signal mask is unspecified. Use siglongjmp instead (Section 9.6).

9.1.6 `sigaction` System Call

You determine the action to be taken when a signal is delivered with the `sigaction` system call, which you call for each signal whose action you want to set:

sigaction—set signal action

```
#include <signal.h>

int sigaction(
     int signum,                    /* signal */
     const struct sigaction *act,   /* new action */
     struct sigaction *oact         /* old action */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

struct sigaction—structure for sigaction

```
struct sigaction {
     void (*sa_handler)(int);   /* SIG_DFL, SIG_IGN, or function pointer */
     sigset_t sa_mask;          /* additional signals to be blocked */
     int sa_flags;              /* flags */
     void (*sa_sigaction)(int, siginfo_t *, void *); /* Realtime Signal
                                                        handler */
};
```

The argument `act` points to a structure that specifies the new action for the whole process—signal actions are not kept for individual threads. If `oact` is non-`NULL`, the old action is returned to where it points. If all you want is the old action, you can set `act` to `NULL`, in which case the action isn't changed. In most of our examples we've set `oact` to `NULL`, but Section 9.7.2 has an example where it's used to save the old action so it can be restored.

You can't change the action for `SIGKILL`, which always terminates the process (not just a thread), or `SIGSTOP`, which always stops a process (not just a thread).

In the `sigaction` structure, `sa_handler` specifies the action with one of these values:

`SIG_DFL` The signal gets its default action, which depends on the signal, as described in Section 9.1.3, but is always ignore, terminate, stop, or continue. Again, these always apply to the entire process, never to just an individual thread.

`SIG_IGN` Ignore the signal, so that delivery has no effect. There's also a side-effect when the `SIGCHLD` signal is set to `SIG_IGN`, which is the same as the effect of the `SA_NOCLDWAIT` flag; see below. Confus-

ingly, setting SIGCHLD to SIG_DFL does *not* cause this side-effect, even though the default for this signal is to ignore it.

function A pointer to a signal-handling function; the signal is said to be *caught*.

A signal-handler looks like the one in the example in Section 9.1.1:

```
static void fcn(int signum)
{
    (void)write(STDOUT_FILENO, "Got signal\n", 11);
    _exit(EXIT_FAILURE);
}
```

The function can be static or not, as long as it has the right prototype. When the signal is delivered, the function is called with the argument set to the number of that signal (e.g., SIGINT, SIGUSR1). The different signals and their macro names are described in Section 9.1.3. You can have a separate function for each signal number, one function for them all, or anything in-between. There's more about what you can do inside a signal handler in Section 9.1.7.

If the Realtime Signals option is supported, you can set the SA_SIGINFO flag (see below) and then use the sa_sigaction member for the signal handler instead of the sa_handler member, which provides a lot more information to the signal handler. This feature is discussed in Section 9.5. An implementation might use the same storage for sa_sigaction and sa_handler, so make sure you set only one of them. Don't set one and then zero the other.

As I mentioned, the signal that caused the handler to be called is blocked while the handler is executing, but you can arrange to block additional signals by specifying them in the sa_mask argument, which you set up with the signal-mask manipulation functions in Section 9.1.8.

When a thread receives a signal that is caught, the signal handler executes within that thread, and that thread's signal mask is what's temporarily modified during the execution of the signal handler. Recall that there are two ways a caught signal can be delivered to a thread: It is delivered to the process, and a thread that has it unblocked is chosen in an unspecified way, or it is delivered to a specific thread.

The following is a list of the portable flags for the sa_flags member. Note that the first two apply to the SIGCHLD signal only.

SA_NOCLDSTOP Don't generate a SIGCHLD signal when a child stops or continues.

`SA_NOCLDWAIT` Don't transform a terminated child process into a zombie. Explained in Section 5.9. Explicitly setting `SIGCHLD` signals to `SIG_IGN` has the identical effect.

`SA_NODEFER` Don't add the signal to the signal mask on entry to the signal handler unless it is explicitly included in the `sa_mask` member. This flag is only present so that the obsolete `signal` function (Section 9.4) can be implemented.

`SA_ONSTACK` Deliver the signal on the alternate signal stack if one has been declared with `sigaltstack`. See Section 9.3.

`SA_RESETHAND` Reset the signal's action to `SIG_DFL` and clear the `SA_SIGINFO` flag when a signal handler is entered. Ineffective for `SIGILL` and `SIGTRAP` signals. Also forces the `SA_NODEFER` behavior and, like that flag, is only present to allow the implementation of `signal`.

`SA_RESTART` Don't allow the signal to interrupt a system call; see Section 9.1.4. There's an example in Section 9.7.4.

`SA_SIGINFO` Use the `sa_sigaction` member instead of the `sa_handler` member; see Section 9.5.

To summarize the flags:

- `SA_NOCLDSTOP` and `SA_NOCLDWAIT` are for the `SIGCHLD` signal only.
- `SA_NODEFER` and `SA_RESETHAND` are compatible with an obsolete and unreliable signal mechanism that you should never use (unless you're doing Exercise 9.4).
- `SA_ONSTACK` is for very specialized uses.
- `SA_SIGINFO` is for use with the Realtime Signals option.
- `SA_RESTART` is occasionally useful.

To summarize the actions and their effect on threads:

- The action for a signal is always on a process-wide basis and is either catch, ignore, terminate, stop, or continue.
- Signal masks are on a per-thread basis.
- You can set catch and ignore explicitly with `sigaction`; you get ignore, terminate, stop, or continue if the action is `SIG_DFL`, but you don't get to choose which of the four it is (see Section 9.1.3).
- Ignore, terminate, stop, and continue always apply to the entire process.

- A caught signal executes a signal handler in only one thread. If the signal is delivered to the process (not targeted to a specific thread) and more than one thread has it unblocked, it is delivered to a thread chosen essentially at random.

As an example, here's a function to ignore `SIGINT` and `SIGQUIT` signals. It's called from the shell in Section 6.4:

```
static struct sigaction entry_int, entry_quit;

static bool ignore_sig(void)
{
    static bool first = true;
    struct sigaction act_ignore;

    memset(&act_ignore, 0, sizeof(act_ignore));
    act_ignore.sa_handler = SIG_IGN;
    if (first) {
        first = false;
        ec_neg1( sigaction(SIGINT, &act_ignore, &entry_int) )
        ec_neg1( sigaction(SIGQUIT, &act_ignore, &entry_quit) )
    }
    else {
        ec_neg1( sigaction(SIGINT, &act_ignore, NULL) )
        ec_neg1( sigaction(SIGQUIT, &act_ignore, NULL) )
    }
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

Note the static `bool` variable, `first`, to ensure that we capture the original action only the first time `sigaction` is called for each signal.

Here's a companion function to restore the actions to what they were before `ignore_sig` was called:

```
static bool entry_sig(void)
{
    ec_neg1( sigaction(SIGINT, &entry_int, NULL) )
    ec_neg1( sigaction(SIGQUIT, &entry_quit, NULL) )
    return true;
```

```
EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

9.1.7 Signal Handlers

After a signal handler has been installed for a signal by a call to `sigaction`, that signal handler is called when the caught signal is delivered. Unless the `SA_NODEFER` has been set, which it rarely is, the caught signal is blocked while the signal handler is executing. In addition, any signals set in the `sa_mask` member of the `sigaction` structure are also blocked. When the signal handler returns, the original mask is restored, even if the temporary mask was modified explicitly (by a call to `pthread_sigmask` or `sigprocmask`) while the signal handler was executing.

OK, so you've caught a signal. What do you do about it? Well, that depends on what kind of signal it is and why it was generated:

- The signal may have been generated by the kernel because it detected an error. Examples would be `SIGFPE` (arithmetic error) or `SIGPIPE` (write on a pipe with no reader). You probably want to display or log an error message and terminate the thread or process. Returning from the handler may be a bad idea because for some signals the state of the computation may be uncertain. And, for hardware-generated errors, like `SIGFPE`, the process may be terminated if you return, as I explained in Section 9.1.3.
- The signal may have been generated by something the user did, such as typing Ctrl-c, which normally generates a `SIGINT` signal. You may want to terminate the program after cleaning up, or you may want to stop a computation, such as a database search, and go back to the user prompt. These are only examples—whatever you do is highly application dependent.
- The signal may be part of your application's design. An example would be sending a process a `SIGUSR1` signal to indicate that a data file is ready for processing.
- A timer may have expired.

Whatever you do, there are always two things to think about:

1. What to do inside the signal handler to change the state of the application so it's known that the signal occurred.
2. Where to go from the signal handler. The choices are returning from the handler, terminating the program, doing a global jump to another part of the program, or generating another signal.

Inside the signal handler, you're restricted as to what system calls or standard functions you can call because the signal may have occurred in a place that can't safely be re-entered. In fact, the SUS (Version 3) defines only 116 so-called async-signal-safe functions, which are listed in Table 9.1.

Table 9.1 Async-Signal-Safe Functions

accept	getppid	sigdelset
access	getsockname	sigemptyset
aio_error	getsockopt	sigfillset
aio_return	getuid	sigismember
aio_suspend	kill	signal
alarm	link	sigpause
bind	listen	sigpending
cfgetispeed	lseek	sigprocmask
cfgetospeed	lstat	sigqueue
cfsetispeed	mkdir	sigset
cfsetospeed	mkfifo	sigsuspend
chdir	open	sleep
chmod	pathconf	socket
chown	pause	socketpair
clock_gettime	pipe	stat
close	poll	symlink
connect	posix_trace_event	sysconf
creat	pselect	tcdrain
dup	raise	tcflow
dup2	read	tcflush
execle	readlink	tcgetattr
execve	recv	tcgetpgrp
_exit/_Exit	recvfrom	tcsendbreak
fchmod	recvmsg	tcsetattr
fchown	rename	tcsetpgrp
fcntl	rmdir	time
fdatasync	select	timer_getoverrun
fork	sem_post	timer_gettime
fpathconf	send	timer_settime
fstat	sendmsg	times
fsync	sendto	umask
ftruncate	setgid	uname
getegid	setpgid	unlink
geteuid	setsid	utime
getgid	setsockopt	wait
getgroups	setuid	waitpid
getpeername	shutdown	write
getpgrp	sigaction	
getpid	sigaddset	

It's not generally safe to call a higher-level function, such as one in a library or even one in your own application, since you can't in general be sure what it's doing, especially after it's evolved over time. You can't even call `printf`, which is why in the example that started this chapter we used `write` instead.

There's a worse restriction: You can't safely refer to a global variable either unless it's of type `volatile sig_atomic_t`.

Technically, it is OK to call an unsafe (nonasync-signal-safe) function or refer to unsafe storage from within a handler if you know that an unsafe function was not interrupted, but since you would know that only under unusual circumstances, it's unwise to program that way. Better to restrict yourself to the list in the table and to modifying globals of type `volatile sig_atomic_t`.

This whole situation seems very risky and it is. Here are some recommendations to keep yourself sane and your programs reliable:

- Signal handlers that display an error (using `write` or other safe functions) and then `_exit` are OK.
- Setting a flag of type `volatile sig_atomic_t` and returning is OK if the `SA_RESTART` flag has been set for the signal.
- Avoiding signal handlers altogether is the best choice. Instead, use threads and `sigwait` (Section 9.2).

Actually, if you've read this far, you've probably already decided to adopt the last recommendation! But, even without signal handlers, signals are still useful because there are two entirely safe ways to handle them without a signal handler: With `sigsuspend` (Section 9.2.3) and with an even better choice, `sigwait` (Section 9.2.2).

To close out this section, here's an example that shows a completely legal signal handler that records what signal arrives and then returns. Even though `SA_RESTART` is specified, `sleep` is still interrupted because it's not affected by the flag.

```
static volatile sig_atomic_t gotsig = -1;

static void handler(int signum)
{
    gotsig = signum;
}
```

```
int main(void)
{
    struct sigaction act;
    time_t start, stop;

    memset(&act, 0, sizeof(act));
    act.sa_handler = handler;
    act.sa_flags = SA_RESTART;
    ec_neg1( sigaction(SIGINT, &act, NULL) )
    printf("Type Ctrl-c in the next 10 secs.\n");
    ec_neg1( start = time(NULL) )
    sleep(20);
    ec_neg1( stop = time(NULL) )
    printf("Slept for %ld secs\n", (long)(stop - start));
    if (gotsig > 0)
        printf("Got signal number %ld\n", (long)gotsig);
    else
        printf("Did not get signal\n");
    exit(EXIT_SUCCESS);

EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
EC_CLEANUP_END
}
```

Here's the output I got; I typed Ctrl-c after I saw the first line:

```
Type Ctrl-c in the next 10 secs.
Slept for 4 secs
Got signal number 2
```

9.1.8 Minimal Defensive Signal Handling

Even if you decide to avoid signal handlers, you usually still need at least a minimal level of signal handling to prevent your application from being terminated accidentally by the user. Also, it's not very professional to just have your program terminated abnormally if there's a bug that causes, say, invalid memory to be referenced (a "segmentation violation"). It's better to catch the signal, log the problem, and inform the user with something better than the simple "Program aborted – segmentation violation," or whatever the shell decides to display.

So most applications should do at least this much:

- Block all signals as soon as your program begins, like this:

```
sigset_t set;

ec_neg1( sigfillset(&set) )
ec_neg1( sigprocmask(SIG_SETMASK, &set, NULL) )
```

(Use `pthread_sigmask` if you have multiple threads.)

- Set all the keyboard-generated signals you don't want to catch to be ignored, such as `SIGINT`.
- Catch `SIGTERM` and arrange to cleanup and terminate when it arrives, as it's the standard way that system administrators shut down processes.
- Catch all the error-generated signals and arrange to display and/or log the error and terminate when one arrives.
- Ignore `SIGPIPE` so that `write` will return an error if it is writing to an empty pipe. This is more convenient than receiving a signal.
- Unblock all signals with calls to `sigemptyset` and `sigprocmask` (or `pthread_sigmask`).

Here's a function you can call at the start of your application to minimally handle signals:

```
static bool handle_signals(void)
{
    sigset_t set;
    struct sigaction act;

    ec_neg1( sigfillset(&set) )
    ec_neg1( sigprocmask(SIG_SETMASK, &set, NULL) )
    memset(&act, 0, sizeof(act));
    ec_neg1( sigfillset(&act.sa_mask) )
    act.sa_handler = SIG_IGN;
    ec_neg1( sigaction(SIGHUP, &act, NULL) )
    ec_neg1( sigaction(SIGINT, &act, NULL) )
    ec_neg1( sigaction(SIGQUIT, &act, NULL) )
    ec_neg1( sigaction(SIGPIPE, &act, NULL) )
    act.sa_handler = handler;
    ec_neg1( sigaction(SIGTERM, &act, NULL) )
    ec_neg1( sigaction(SIGBUS, &act, NULL) )
    ec_neg1( sigaction(SIGFPE, &act, NULL) )
    ec_neg1( sigaction(SIGILL, &act, NULL) )
    ec_neg1( sigaction(SIGSEGV, &act, NULL) )
    ec_neg1( sigaction(SIGSYS, &act, NULL) )
    ec_neg1( sigaction(SIGXCPU, &act, NULL) )
    ec_neg1( sigaction(SIGXFSZ, &act, NULL) )
    ec_neg1( sigemptyset(&set) )
    ec_neg1( sigprocmask(SIG_SETMASK, &set, NULL) )
    return true;
```

```
EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

Here's the actual handler and two supporting functions it calls:

```
static void handler(int signum)
{
    int i;
    struct {
        int signum;
        char *msg;
    } sigmsg[] = {
        { SIGTERM, "Termination signal" },
        { SIGBUS, "Access to undefined portion of a memory object" },
        { SIGFPE, "Erroneous arithmetic operation" },
        { SIGILL, "Illegal instruction" },
        { SIGSEGV, "Invalid memory reference" },
        { SIGSYS, "Bad system call" },
        { SIGXCPU, "CPU-time limit exceeded" },
        { SIGXFSZ, "File-size limit exceeded" },
        { 0, NULL}
    };

    clean_up();
    for (i = 0; sigmsg[i].signum > 0; i++)
        if (sigmsg[i].signum == signum) {
            (void)write(STDERR_FILENO, sigmsg[i].msg,
              strlen_safe(sigmsg[i].msg));
            (void)write(STDERR_FILENO, "\n", 1);
            break;
        }
    _exit(EXIT_FAILURE);
}

static void clean_up(void)
{
    /*
        Clean-up code goes here --
        must be async-signal-safe.
    */
}

static size_t strlen_safe(const char *s)
{
    size_t n = 0;
    while (*s++ != '\0')
        n++;
    return n;
}
```

The idea is that you replace the guts of `clean_up` with code appropriate for your application. I coded my own version of `strlen` because, silly as it sounds, the standard `strlen` isn't in the list of async-signal-safe functions. For the same reason I used `write` in the handler instead of `fprintf`. Note also the use of `_exit` instead of `exit`—as explained in Section 5.7, the underscored version skips calling `atexit` and flushing of standard C I/O buffers and is the only way to do an async-signal-safe normal exit.

9.1.9 Generating a Signal Synthetically

As I said in Section 9.1.3, each signal has a natural cause and can also be generated explicitly by a call to `kill`, `killpg`, `pthread_kill`, `abort`, `raise`, or `sigqueue` (Section 9.5.4):

kill—generate signal for processes

```
#include <signal.h>

int kill(
     pid_t pid,             /* process ID or other specification */
     int signum             /* signal */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

killpg—generate signal for process group

```
#include <signal.h>

int killpg(
     pid_t pgrp,            /* process-group ID */
     int signum             /* signal */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

pthread_kill—generate signal for thread

```
#include <signal.h>

int pthread_kill(
     pthread_t thread_id,   /* thread ID */
     int signum             /* signal */
);
/* Returns 0 on success, error number on failure (errno not set) */
```

abort—generate SIGABRT

```
#include <stdlib.h>

void abort(void);
/* Does not return */
```

raise—generate signal for thread

```
#include <signal.h>

int raise(
    int signum      /* signal */
);
/* Returns 0 on success or non-zero on error (sets errno) */
```

The misnamed `kill` system call sends any signal, not just `SIGKILL`, to one or more processes for which it has permission to send signals. It has permission if it's run by the superuser or if the real or effective user ID of the sending process matches the real or saved set-user-ID of the receiving process. Which processes the signal gets delivered to depends on the `pid` argument:

>0	the process whose process ID is `pid`
0	the process group whose process-group ID is the same as that of the sending process
<–1	the process group whose process-group ID is the same as the absolute value of `pid`
–1	all processes for which the sender has permission, except for an implementation defined set of system processes

If `signum` is 0, `kill` just tests its `pid` argument for validity. It's a way to tell whether a process or process-group is alive. If the sending process doesn't have permission to send a signal, the call will fail, but the `errno` value will indicate whether the process or process-group is alive: It will be `ESRCH` if `pid` doesn't exist, and `EPERM` if it does but the sender has no permission.

`killpg`, a totally unnecessary system call, generates a signal only for the process group whose process-group ID is `pgrp`, so it's identical to:

```
kill(-pgrp, signum);
```

`pthread_kill` is like `kill`, except it sends the signal to only thread `thread_id`, which must be in the same process as the sending thread. It doesn't have the broadcast ability that `kill` has for processes. Be careful with `pthread_kill`—remember that signals that terminate, stop, or continue always affect the whole process. So, `pthread_kill` works the way you want it to only for caught signals. If you execute

```
pthread_kill(tid, SIGKILL);
```

the process will be terminated, not just thread `tid`. (You use `pthread_cancel` to terminate just one thread.)

`abort` is almost like `kill` with an argument of `SIGABRT`, but unless that signal is caught and the signal handler does not return (e.g., calls `siglongjmp` or `_exit`), the process is terminated anyway as if `SIGABRT` had its default behavior; `abort` *never* returns. For example, if `SIGABRT` is set to `SIG_IGN` by `sigaction`, `abort` terminates a process, but

```
kill(getpid(), SIGABRT)
```

has no effect.

`raise` is actually a Standard C function. It sends a signal to the thread that executes it. That is, it's the same as:

```
pthread_kill(pthread_self(), signum);
```

However, `raise` is always present even if the POSIX Threads option is not supported, in which case it's the same as:

```
kill(getpid(), signum);
```

9.1.10 Effect of `fork`, `pthread_create`, and `exec` on Signals

There are three properties of a thread that are set for a new process, thread, or program by a `fork`, `pthread_create`, or `exec`:

1. **Signal actions:** After a `fork`, the child inherits all signal actions. After an `exec`, signals set to `SIG_DFL` remain that way; signals set to `SIG_IGN` remain that way, except for `SIGCHLD`, which may be set to `SIG_IGN` or `SIG_DFL`, as the implementation chooses; caught signals are set to `SIG_DFL`. As all actions are process-wide, `pthread_create` has no effect.
2. **Signal mask:** Inherited from the `fork`ing thread after a `fork`; stays the same as the `exec`ing thread after an `exec`; copied to the new thread from the creating thread after a `pthread_create`.
3. **Pending signals:** Cleared after a `fork`; same as the `exec`ing thread after an `exec`; cleared after a `pthread_create`.

The simple way to remember these nine rules (three properties times three system calls) is this: The bias is to copy or inherit properties unchanged, so there are only three exceptions to remember, the first two of which make perfect sense:

- A caught signal has to be changed to `SIG_DFL` if the signal handler disappears, which it does on an `exec`.
- Pending signals are per-process or per-thread, so they're cleared when there's a new process or new thread.
- The standards makers were more concerned with not breaking existing implementations than in forcing portability, so they waffled on `SIGCHLD`.

9.2 Waiting for a Signal

This section describes system calls that allow a process to wait for the delivery of a signal.

9.2.1 `pause` System Call

We've encountered lots of system calls that block waiting for an event before they complete some activity. For example, when reading a terminal, `read` normally waits for a full line to be typed. `pause` is pure wait: It doesn't do anything, and it's not waiting for anything in particular.

pause—wait for signal

```
#include <unistd.h>

int pause(void);
/* Returns -1 on error (sets errno) */
```

Since a delivered signal interrupts most system calls that are blocked, we might as well say that `pause` waits for a caught signal. If the signal-catching function returns, `pause` returns with `errno` set to `EINTR`, but since that's the only way `pause` ever returns there's no point testing for it.

Usually, you'll use a more sophisticated call like `sigwait` instead of `pause`.

9.2.2 `sigwait` System Call

Unlike `pause`, `sigwait` lets you choose what you want to wait for. You don't need a signal handler, and when it returns you're told what signal arrived:

sigwait—wait for signal

```
#include <signal.h>

int sigwait(
    const sigset_t *set,       /* signals to wait for */
    int *signum                /* signal that was accepted */
);
/* Returns 0 on success or error number on error */
```

The argument `set` is a set of signals (see Section 9.1.5) that `sigwait` is to wait for. When one becomes pending, its number is returned through the `signum` argument and `sigwait` returns. If one or more signals are already pending when `sigwait` is called, one is chosen in an undefined way and immediately returned. The technical term for a signal returned by `sigwait` is "accepted"; it is not "delivered."

When you use `sigwait`, you want a signal to stay pending until `sigwait` returns it; you don't want a signal to ever be delivered. So, you block the signals that `sigwait` is to wait for and leave them blocked. (The lifecycle of a signal—generated-to-pending-to-delivered—was described in Section 9.1.2.)

If more than one thread is in `sigwait` waiting for the same signal sent to a process, only one thread gets it, and the choice is made in an undefined way. If a signal is sent to a specific thread, only that thread's `sigwait` (if it has one) can return it.

Typically, you use `sigwait` for one of two purposes:

- When the thread can't proceed until some event occurs that's associated with a signal. For example, one thread might send `SIGUSR1` to another thread when a message has arrived. Usually, though, it's better to use a condition variable (Section 5.17.4) for this purpose. There's an example later in this section.
- When one thread is designated to handle signals. That is, instead of signal handling function, a waiting thread is used. All threads have the signals to be waited for blocked, and one thread executes a `sigwait`. But this only works for signals sent to a process—if a signal is sent to a thread, only that thread's `sigwait` can return it.

For signals sent to a process, it's much better to use `sigwait` instead of a signal handler because none of the restrictions for signal handlers apply. When `sigwait` returns, you're free to call any system call or function or do anything else that you can do in a thread.

In Section 9.1.8 we showed a function, `handle_signals`, that arranges to catch all the detected-error signals (e.g., `SIGFPE`, `SIGSEGV`) so it can call a cleanup function and display a nice message before exiting. Let's recode this function to use `sigwait` in a thread instead of a signal handler:

```
static bool handle_signals(void) /* do not use -- see below */
{
    sigset_t *set;
    struct sigaction act;
    pthread_t tid;

    ec_null( set = malloc(sizeof(*set)) )
    ec_neg1( sigfillset(set) )
    ec_rv( pthread_sigmask(SIG_SETMASK, set, NULL) )
    memset(&act, 0, sizeof(act));
    act.sa_handler = SIG_IGN;
    ec_neg1( sigaction(SIGHUP, &act, NULL) )
    ec_neg1( sigaction(SIGINT, &act, NULL) )
    ec_neg1( sigaction(SIGQUIT, &act, NULL) )
    ec_neg1( sigaction(SIGPIPE, &act, NULL) )
    ec_neg1( sigemptyset(set) )
    ec_neg1( sigaddset(set, SIGTERM) )
    ec_neg1( sigaddset(set, SIGBUS) )
    ec_neg1( sigaddset(set, SIGFPE) )
    ec_neg1( sigaddset(set, SIGILL) )
    ec_neg1( sigaddset(set, SIGSEGV) )
    ec_neg1( sigaddset(set, SIGSYS) )
    ec_neg1( sigaddset(set, SIGXCPU) )
    ec_neg1( sigaddset(set, SIGXFSZ) )
    ec_rv( pthread_sigmask(SIG_SETMASK, set, NULL) )
    ec_rv( pthread_create(&tid, NULL, sig_thread, set) )
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}

static void *sig_thread(void *arg)
{
    int signum;
    int i;
    struct {
        int signum;
        char *msg;
    } sigmsg[] = {
        { SIGTERM, "Termination signal" },
        { SIGBUS, "Access to undefined portion of a memory object" },
```

```
        { SIGFPE, "Erroneous arithmetic operation" },
        { SIGILL, "Illegal instruction" },
        { SIGSEGV, "Invalid memory reference" },
        { SIGSYS, "Bad system call" },
        { SIGXCPU, "CPU-time limit exceeded" },
        { SIGXFSZ, "File-size limit exceeded" },
        { 0, NULL}
    };

    while (true) {
        ec_rv( sigwait((sigset_t *)arg, &signum) )
        clean_up();
        for (i = 0; sigmsg[i].signum > 0; i++)
            if (sigmsg[i].signum == signum) {
                fprintf(stderr, "%s\n", sigmsg[i].msg);
                break;
            }
        _exit(EXIT_FAILURE);
    }
    return (void *)true; /* never get here */

EC_CLEANUP_BGN
    EC_FLUSH("sig_thread")
    return (void *)false;
EC_CLEANUP_END
}

static void clean_up(void)
{
    /*
        Clean-up code goes here --
        need not be async-signal-safe.
    */
}
```

The advantage of this version over the one in Section 9.1.8 is that, when a signal is returned by `sigwait`, we're in a thread, not a signal handler, and we're free to use any system calls or functions we like—we're not restricted to the async-signal-safe list. Note that the comment in the `clean_up` function has been changed accordingly.

The disadvantage of this version is that it doesn't work! There are two reasons why, both serious:

- If some other thread gets a `SIGSYS`, for example, that signal will be sent to that thread, not to the process, and the `sigwait` in the `sig_thread` function won't return with it. Indeed, since that signal is blocked in all threads, it

will just stay pending forever. So, the original version, using signal handlers, is the one you should use for the detected-error signals.

- If one of the hardware-detected signals, `SIGBUS`, `SIGFPE`, `SIGILL`, and `SIGSEGV`, occurs naturally while blocked, the result is undefined (see Section 9.1.3). Most likely the process will be immediately terminated, and `sigwait` will never get a chance to return it. This wasn't a problem with the signal-handler version because, after the initial setup, those four signals were unblocked.

This is not to say that `sigwait` isn't useful. Most signals, including all but the detected-error group in Section 9.1.3, are sent to the process when they are generated naturally (i.e., not by `pthread_kill`), so a thread waiting in `sigwait` works perfectly well and is a much better choice than a signal handler.

9.2.3 `sigsuspend` System Call

`sigsuspend` is an older, nonmultithreading system call that also waits for a signal. Before we get to its details, let's explore the problem it solves. Back in Chapter 8 when we kept running examples that forked to create processes that connected to the parent's socket, it was important for the parent to get the socket bound before the child connected, and we used the crude technique of having the children sleep for a few seconds to give the parent a chance to get ahead. Not only is sleeping unreliable, because there's no guarantee that a few seconds is enough, but it's also inefficient because a few seconds may be much too long. To recap, here's the same problem in a simpler example:

```
void try1(void)
{
    if (fork() == 0) {
        printf("child\n");
        exit(EXIT_SUCCESS);
    }
    printf("parent\n");
    return;
}
```

This function displayed

```
child
parent
```

but we want the parent to execute first and then tell the child when to proceed. Our first attempt to synchronize them has the parent sending a `SIGUSR1` signal to the child, who catches it and sets a variable:

```
static volatile sig_atomic_t got_sig;
static void handler(int signum)
{
    if (signum == SIGUSR1)
        got_sig = 1;
}

void try2(void)
{
    pid_t pid;

    got_sig = 0;
    ec_neg1( pid = fork() )
    if (pid == 0) {
        struct sigaction act;

        memset(&act, 0, sizeof(act));
        act.sa_handler = handler;
        ec_neg1( sigaction(SIGUSR1, &act, NULL) )
        while (got_sig == 0)
            if (pause() == -1 && errno != EINTR)
                EC_FAIL
        printf("child\n");
        exit(EXIT_SUCCESS);
    }
    printf("parent\n");
    ec_neg1( kill(pid, SIGUSR1) )
    return;

EC_CLEANUP_BGN
    EC_FLUSH("try2")
EC_CLEANUP_END
}
```

The handler is safe—it just sets a variable of the approved type. The child tests the variable in a loop, pausing to await the arrival of a signal. (`pause`, which we'll get to in Section 9.2.1, blocks until a signal arrives.) Now the sequence is what we want:

```
parent
child
```

But there are two problems:

- If `SIGUSR1` is delivered to the child before it has a chance to install the handler, it will terminate the child process. A potential solution is to install the handler before the `fork`, so the child will inherit it, but that only works in a parent-child situation. We'd like a solution that works for arbitrary processes that need to synchronize.
- If `SIGUSR1` is delivered between the test in the `while` statement and the call to `pause`, `pause` will wait forever since the signal that's supposed to wake it up arrived before it even got to sleep.

We can try to fix things by blocking the `SIGUSR1` signal until we're ready for it:

```
void try3(void)
{
    sigset_t set;
    pid_t pid;

    got_sig = 0;
    ec_neg1( sigemptyset(&set) )
    ec_neg1( sigaddset(&set, SIGUSR1) )
    ec_neg1( sigprocmask(SIG_SETMASK, &set, NULL) )
    ec_neg1( pid = fork() )
    if (pid == 0) {
        struct sigaction act;
        sigset_t suspendset;

        memset(&act, 0, sizeof(act));
        act.sa_handler = handler;
        ec_neg1( sigaction(SIGUSR1, &act, NULL) )
        ec_neg1( sigfillset(&suspendset) )
        ec_neg1( sigdelset(&suspendset, SIGUSR1) )
        ec_neg1( sigprocmask(SIG_SETMASK, &suspendset, NULL) )
        while (got_sig == 0)
            if (pause() == -1 && errno != EINTR)
                EC_FAIL
        printf("child\n");
        exit(EXIT_SUCCESS);
    }
    printf("parent\n");
    ec_neg1( kill(pid, SIGUSR1) )
    return;

EC_CLEANUP_BGN
    EC_FLUSH("try3")
EC_CLEANUP_END
}
```

This totally fixes problem #1. Because it's blocked, `SIGUSR1` can't arrive until after the handler is installed. But problem #2 is still with us, and there's nothing we can do to fix it. Although the gap is small, it's still possible for the signal to arrive between the test and the pause.

What we want is a way to keep the signal blocked until the `pause` begins. Or, to say it another way, we want unblocking and pausing to be atomic. That's exactly what `sigsuspend` does:

sigsuspend—change signal mask and wait for signal

```
#include <signal.h>

int sigsuspend(
    const sigset_t *sigmask       /* temporary signal mask */
);
/* Returns -1 on error, always (sets errno) */
```

`sigsuspend` temporarily replaces the thread's signal mask with `sigmask` and then waits until an unblocked signal is delivered whose action is termination or being caught. If it's termination, the process (not just the thread, remember) is terminated, and `sigsuspend` doesn't return. If it's caught and the signal handler returns, the previous signal mask is restored and `sigsuspend` returns with an error. Usually, the error is only `EINTR`; that is, a return of –1 with an `errno` or `EINTR` is normal for an interrupted system call (same as `pause`, as shown in the examples above).

In essentially all cases, the mask passed to `sigsuspend` has the effect of unblocking one or more signals that were blocked prior to the call, although that isn't actually a requirement. It's just that anything else doesn't make much sense.

OK, perfect. We're now set to fix our synchronizing problem for good by simply replacing `pause` with `sigsuspend`. We don't need the `while` loop anymore because unblocking the signal and suspending are now atomic.

```
void try4(void)
{
    sigset_t set;
    pid_t pid;

    ec_neg1( sigemptyset(&set) )
    ec_neg1( sigaddset(&set, SIGUSR1) )
    ec_neg1( sigprocmask(SIG_SETMASK, &set, NULL) )
    ec_neg1( pid = fork() )
```

```
    if (pid == 0) {
        struct sigaction act;
        sigset_t suspendset;

        memset(&act, 0, sizeof(act));
        act.sa_handler = handler;
        ec_neg1( sigaction(SIGUSR1, &act, NULL) )
        ec_neg1( sigfillset(&suspendset) )
        ec_neg1( sigdelset(&suspendset, SIGUSR1) )
        if (sigsuspend(&suspendset) == -1 && errno != EINTR)
            EC_FAIL
        printf("child\n");
        exit(EXIT_SUCCESS);
    }
    printf("parent\n");
    ec_neg1( kill(pid, SIGUSR1) )
    return;

EC_CLEANUP_BGN
    EC_FLUSH("try4")
EC_CLEANUP_END
}
```

We could still use the same handler function, which sets the `got_sig` variable; however, we don't need the variable anymore, and an empty handler will work fine:

```
static void handler(int signum)
{
}
```

Also, the test of the return code from `sigsuspend` against –1 isn't really needed since it always returns –1. But doing that is weird and would confuse readers who don't happen to know or remember all the details of `sigsuspend`.

Note that we are blocking `SIGUSR1` before the `fork` so that the child inherits the signal mask. In a situation where the two processes to be synchronized are unrelated, the process that calls `sigsuspend` simply does its own blocking. Actually, this is just a special case of the general recommendation to start every application with all signals blocked, as I said in Section 9.1.8.

Note that, unlike `sigwait`, you almost always use `sigsuspend` along with a signal handler, but usually the handler doesn't do anything. It's only present so that the delivered signal will interrupt `sigsuspend`.

Another big difference between `sigsuspend` and `sigwait` is that with `sigwait` the signal you're waiting for stays blocked. In fact, it is never delivered—rather, `sigwait` accepts it: removes it from the collection of pending signals and returns

it. So the race condition between unblocking the signal and waiting for it to be delivered, which `sigsuspend` eliminates, doesn't exist when you're using `sigwait` because you never unblock the signal. Our synchronization code thus becomes even simpler:

```
void try5(void)
{
    sigset_t set;
    pid_t pid;

    ec_neg1( sigemptyset(&set) )
    ec_neg1( sigaddset(&set, SIGUSR1) )
    ec_neg1( sigprocmask(SIG_SETMASK, &set, NULL) )
    ec_neg1( pid = fork() )
    if (pid == 0) {
        int signum;

        ec_rv( sigwait(&set, &signum) )
        printf("child\n");
        exit(EXIT_SUCCESS);
    }
    printf("parent\n");
    ec_neg1( kill(pid, SIGUSR1) )
    return;

EC_CLEANUP_BGN
    EC_FLUSH("try5")
EC_CLEANUP_END
}
```

We should mention that synchronizing the parent and child can also be done with a pipe, skipping the complexities of signals altogether:

```
void try6(void)
{
    int pfd[2];
    pid_t pid;

    ec_neg1( pipe(pfd) )
    ec_neg1( pid = fork() )
    if (pid == 0) {
        char c;

        ec_neg1( close(pfd[1]) )
        ec_neg1( read(pfd[0], &c, 1) )
        ec_neg1( close(pfd[0]) )
        printf("child\n");
        exit(EXIT_SUCCESS);
    }
```

```
    printf("parent\n");
    ec_neg1( close(pfd[0]) )
    ec_neg1( close(pfd[1]) )
    return;

EC_CLEANUP_BGN
    EC_FLUSH("try6")
EC_CLEANUP_END
}
```

Here the child blocks in `read` until the parent closes the writing end of the pipe, resulting in the child getting a zero return.

9.3 Miscellaneous Signal System Calls

You can find out whether a signal is pending with `sigpending`, which returns a set of the pending signals:

sigpending—examine pending signals

```
#include <signal.h>

int sigpending(
    sigset_t *set              /* returned set of pending signals */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

You can test which signals are pending by using `sigismember` (Section 9.1.5) on the returned set. Of course, it may not still be pending by the time you test it unless it's blocked.

Recall from Section 9.1.6 that the `SA_ONSTACK` flag arranges for a signal's handler to execute on an alternate stack. You manage the stack with `sigaltstack`:

sigaltstack—set and get alternate stack context

```
#include <signal.h>

int sigaltstack(
    const stack_t *stack,      /* new stack */
    stack_t *ostack            /* old stack */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

See your system's documentation or the SUS for details of this system call.

Also, recall that the `SA_RESTART` flag prevents a signal from interrupting a function. You can turn the flag on or off, without calling `sigaction`, with `siginterrupt`:

siginterrupt—set or clear SA_RESTART flag

```
#include <signal.h>

int siginterrupt(
    int signum,           /* signal */
    int flag              /* non-zero to clear, zero to set */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

9.4 Deprecated Signal System Calls

The system calls in this section are standardized, but they don't add any functionality to the system calls already presented and aren't worth spending your time on, except for your time on Exercises 9.4 and 9.5.

The classic way to set the action for a signal was with the `signal` system call:

signal—set signal action

```
#include <signal.h>

void (*signal(
    int signum,           /* signal */
    void (*act)(int)      /* new action */
))(int);
/* Returns old action or SIG_ERR on error (sets errno) */
```

The strange declaration means that `signal` returns an action that could be a pointer to a void function taking an integer argument (the signal number).

There are two problems when you catch a signal with `signal`:

- Upon delivery, the action is set to its default. You need to call `signal` again if you still want to catch it.
- The delivered signal isn't blocked, so a second arrival can terminate the process.

The way to get around these problems is to use `sigaction` (Section 9.1.6) and forget about `signal`.

There is a group of five system calls that provide for simplified signal handling, but you should avoid using them, as they don't do anything that the primary functions (e.g., `sigaction`) don't do better. I'll briefly describe them anyway.

You can set a `sigaction`-style action for a signal without using a structure with `sigset`:

sigset—set signal action

```
#include <signal.h>

void (*sigset(
     int signum,            /* signal */
     void (*act)(int)       /* new action */
))(int);
/* Returns old action or SIG_ERR on error (sets errno) */
```

`sigset` is as simple to call as `signal`, but it has the behavior of `sigaction`, in that a delivered signal is masked while the handler is executing, and the action is not changed. In addition, it takes a new action, `SIG_HOLD`, which just adds the signal to the signal mask. Unlike `sigaction`, if you call `sigset` without the `SIG_HOLD` action, it also unblocks it (removes it from the signal mask) as a byproduct.

The chief reason for avoiding `sigset` and the other related functions that follow is that it's hard enough mastering what `sigaction` does without also trying to learn a somewhat different combination of features. *Less is more!* Another reason for avoiding them is that they're not defined in a multithreading environment. Imagine the problems if you've used them in a single-threading program and then multithreading is added later!

To unblock a signal directly (remove it from the signal mask), you can call `sigrelse`:

sigrelse—unblock signal

```
#include <signal.h>

int sigrelse(
     int signum,            /* signal */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

There's a simplified version of `sigsuspend`, `sigpause`, which removes a signal from the signal mask, pauses until a signal arrives (any unblocked signal), and then restores the mask:

sigpause—change signal mask and wait for signal

```
#include <signal.h>

int sigpause(
    int signum,            /* signal */
);
/* Returns -1 on error, always (sets errno) */
```

Finally, here are two redundant system calls, as they do exactly what `sigset` does with the `SIG_HOLD` and `SIG_IGN` actions:

sighold—block signal

```
#include <signal.h>

int sighold(
    int signum,            /* signal */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

sigignore—ignore signal

```
#include <signal.h>

int sigignore(
    int signum,            /* signal */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

9.5 Realtime Signals Extension (RTS)

The so-called POSIX.4 (real-time) standard includes a new signal mechanism, RTS, that improves on the signal-related system calls described earlier in this chapter by:

- Increasing the number of signals for application use beyond just `SIGUSR1` and `SIGUSR2`
- Allowing for queuing of signals, so that a signal that's generated when a signal of the same type is pending isn't lost
- Specifying the order of delivery of signals
- Including additional information with a signal, such as who it came from and why, and possibly some application data

The new features add on to the traditional features so you can still use the classic 28 signals (Section 9.1.3), with the same handlers and other actions, and with the same synthesizing calls, such as `kill`. The standard doesn't say that you can use

the new RTS features (e.g., queuing, passing a value) with the old signals, although on many systems you can, so avoid doing that if you want to be portable.

There are several feature-test macros (Section 1.5.4) that indicate whether these features are available. The principal one is `_POSIX_REALTIME_SIGNALS`.

The subsections in this section discuss most of the RTS features in detail. At various times, it will be convenient to refer to a group of functions that use the RTS signal mechanism to send signals. They're spelled out in the next section as those associated with codes `SI_QUEUE`, `SI_TIMER`, `SI_ASYNCIO`, and `SI_MESGQ`. I call the group the *RTS-generation* functions (my name, not one used elsewhere).

There are a few more RTS features described in Sections 9.7.5 and 9.7.6.

9.5.1 RTS Signal Handlers

You may want to review the discussion of `sigaction` in Section 9.1.6 before continuing.

If you set the `SA_SIGINFO` flag in the structure passed to `sigaction`, you use the `sa_sigaction` member instead of the `sa_handler` member to hold the pointer to the signal-handling function.[6] Instead of just a signal-number argument, it has this more elaborate prototype:

```
void rts_handler(int signum, siginfo_t *info, void *context);
```

The `info` argument points to information about the signal in a `siginfo_t` structure, which we first used with `waitid` in Section 5.8:

siginfo_t—structure for sigaction

```
typedef struct {
    int si_signo;              /* signal number */
    int si_errno;              /* errno value associated with signal */
    int si_code;               /* signal code (see below) */
    pid_t si_pid;              /* sending process ID */
    uid_t si_uid;              /* real user ID of sending process */
    void *si_addr;             /* address of faulting instruction */
    int si_status;             /* exit value or signal */
    long si_band;              /* band event for SIGPOLL */
    union sigval si_value;     /* signal value */
} siginfo_t;
```

6. The SUS is unclear about whether you can use `SIG_IGN` or `SIG_DFL` with the `sa_sigaction` member; to be safe, use them only with `sa_handler`.

union sigval—union for signal values[7]

```
union sigval {
    int sival_int;          /* integer signal value */
    void *sival_ptr;        /* pointer signal value */
};
```

Member `si_signo` is just a repeat of the `signum` argument to the handler.

The use of `si_errno` is up to the implementation; it may contain an error code that indicates the cause of the signal.

Member `si_code` indicates the reason for the signal. Usually, if it's 0 or negative, the signal is from a process; the process ID is in `si_pid` and the real user ID is in `si_uid`. The `si_code` member is one of:

`SI_USER`	sent by `kill` or, at the discretion of the implementation, by one of the other system calls for synthesizing a signal (e.g., `raise`)
`SI_QUEUE`	sent by `sigqueue` (Section 9.5.4)
`SI_TIMER`	expiration of timer set by `timer_settime` (Section 9.7.6)
`SI_ASYNCIO`	completion of asynchronous I/O (Section 3.9)
`SI_MESGQ`	arrival of message (Section 7.7)

For the last four, the RTS-generation functions, there's also a value that can be sent with the signal that's accessible through the `si_value` member.

If `si_code` is positive, the signal came from the kernel and the code depends on the signal. For example, if the signal is `SIGFPE`, the code is `FPE_INTDIV` for integer divide by zero, `FPE_INTOVF` for integer overflow, and `FPE_FLTDIV` for floating-point divide by zero. For the full list of the `SIGFPE` codes and the codes for other signals, see [SUS2002] or your system's documentation. Also, Section 5.8 lists the codes for `SIGCHLD`.

How the other members are used depends on the signal. Those for `SIGCHLD` were described in Section 5.8. For `SIGILL` and `SIGSEGV`, the `si_addr` member gives the actual machine address that caused the problem. For `SIGPOLL` (used with STREAMS), member `si_band` indicates the band event.

7. FreeBSD (and perhaps other systems) defines the members as `sigval_int` and `sigval_ptr`, but it doesn't claim to support RTS. Perhaps that's OK.

Here's a program that displays some of the additional information that's available in a signal handler when you set the `SA_SIGINFO` flag:

```
int main(void)
{
    struct sigaction act;
    union sigval val;

    memset(&act, 0, sizeof(act));
    act.sa_flags = SA_SIGINFO;
    act.sa_sigaction = handler;
    ec_neg1( sigaction(SIGUSR1, &act, NULL) )
    ec_neg1( sigaction(SIGRTMIN, &act, NULL) )
    ec_neg1( kill(getpid(), SIGUSR1) )
    val.sival_int = 1234;
    ec_neg1( sigqueue(getpid(), SIGRTMIN, val) )
    exit(EXIT_SUCCESS);

EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
EC_CLEANUP_END
}

static void handler(int signum, siginfo_t *info, void *context)
{
    printf("signal number: %d\n", info->si_signo);
    printf("sending process ID: %ld\n", (long)info->si_pid);
    printf("real user ID of sending process: %ld\n", (long)info->si_uid);
    switch (info->si_code) {
    case SI_USER:
        printf("Signal from user\n");
        break;
    case SI_QUEUE:
        printf("Signal from sigqueue; value = %d\n",
          info->si_value.sival_int);
        break;
    case SI_TIMER:
        printf("Signal from timer expiration; value = %d\n",
          info->si_value.sival_int);
        break;
    case SI_ASYNCIO:
        printf("Signal from asynchronous I/O completion; value = %d\n",
          info->si_value.sival_int);
        break;
    case SI_MESGQ:
        printf("Signal from message arrival; value = %d\n",
          info->si_value.sival_int);
        break;
```

```
    default:
        printf("Other signal\n");
    }
}
```

Compare this code to, say, the first example in this chapter and you'll see the differences in setting up for the call to `sigaction`: the `SA_SIGINFO` flag is set, and the member for the handler is `sa_sigaction` instead of `sa_handler`. I'll introduce the signal `SIGRTMIN` in the next section; for now just pretend it's `SIGUSR2` if that helps you understand the code.

It's legal for us to call `printf` in the handler, even though that function is not async-signal-safe, because the signal was generated by a system call that is async-signal-safe: `kill` or `sigqueue` (which we'll get to shortly, in Section 9.5.4).

Here's the output I got:

```
signal number: 10
sending process ID: 29501
real user ID of sending process: 500
Signal from user
signal number: 32
sending process ID: 29501
real user ID of sending process: 500
Signal from sigqueue; value = 1234
```

The third argument to the signal handler, `context`, is a pointer to an object of type `ucontext_t` that describes the receiving process's context at the time it was interrupted. It's a pointer to a `void` rather than a `ucontext_t` because at the time this prototype was introduced the new type wasn't standardized. This pointer is not always implemented and rarely is used. For more information, see [SUS2002] or your system's documentation.

9.5.2 RTS Signals

RTS introduced a group of new signals whose numbers range from `SIGRTMIN` to `SIGRTMAX`, with at least `RTSIG_MAX` signals available. You can get the actual number at runtime with `sysconf` (Section 1.5.5), but it's always at least 8. (That's in fact what it was on my version of Solaris; on Linux it was 32.)

For the RTS-generation functions (defined at the start of Section 9.5), you get to specify what signal you want, and you should choose one of the RTS signals; whether you can use one of the classic signals is implementation-dependent.

There aren't any symbols for the RTS signals other than for the first and last, so you refer to them as SIGRTMIN, SIGRTMIN + 1, and so on, up to SIGRTMAX. Of course, you'll want to define your own macros, like this:

```
#define DB_IO_DONE SIGRTMIN + 4
```

Alas, the problem of two libraries both using the same signals wasn't solved by the POSIX.4 group, so be careful.

SIGRTMIN and SIGRTMAX aren't necessarily constant integer expressions, so you can't portably use them as case labels or in preprocessor expressions (e.g., in #ifs).

If more than one RTS signal (SIGRTMIN through SIGRTMAX) is pending, the lowest-numbered signal is delivered first. So, you can think of them as being in priority order, and you may want to take that into account as you assign them to different application purposes. The default action for all RTS signals is termination.

9.5.3 Queued Signals

Normally, you can't count on what happens when a signal is generated while a signal of the same type is pending. If the implementation isn't equipped to queue signals, keeping only a flag for each signal, say, then it will just forget about the second signal. You must never use signals for counting purposes—to keep track of the number of times SIGUSR1 was sent, for example.

With RTS, however, an RTS signal (SIGRTMIN through SIGRTMAX) sent by an RTS-generation function (defined at the start of Section 9.5) is queued if the SA_SIGINFO flag was set for that signal by sigaction. Whether queuing also works for classic signals is implementation dependent, so don't count on it. Many systems queue the RTS signals even if you don't set the SA_SIGINFO flag, which is perfectly legal, but you should set the flag anyway to be portable.

If you want queuing but don't want a signal handler, because you're going to use, say, sigwait (Section 9.2.2), it would seem that you could set the sa_sigaction member of the sigaction structure to SIG_DFL, but the SUS isn't clear on this point. To be safe, code an empty handler and set the action to point to it.

The maximum queue length for a process is at least 32; you can find out the actual number with sysconf (Section 1.5.5).

9.5.4 `sigqueue`

sigqueue—generate signal for process

```
#include <signal.h>

int sigqueue(
     pid_t pid,                 /* process ID */
     int signum,                /* signal */
     const union sigval value   /* value */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

We've already seen `sigqueue` in use, in the example in Section 9.5.1. The first two arguments are like those for `kill`, except `pid` can only be a process ID—the broadcast capabilities of `kill` (e.g., sending to all processes) aren't there. In addition, you can pass a value that the signal handler can receive if the `SA_SIGINFO` flag was set and it uses the three-argument form described in Section 9.5.1. You decide whether you want to use the `int` or pointer members of the union; the handler has to know which to use because no information about which you used is passed along.

If you're sending the signal to another process, you probably can't use a pointer because it won't mean anything to the other process. The only way it would be valid would be if the two processes shared memory that was located at the same address and the pointer pointed into that shared memory.

As I said in the previous section, if the `SA_SIGINFO` flag was set and there's already a `signum` signal pending for the receiving process, the new signal is queued.

You can count on queuing and passing a value only for RTS Signals (Section 9.5.2).

You can only queue so many signals, as explained in the previous section. If the new signal can't be queued, `sigqueue` returns –1 with `errno` set to `EAGAIN`. There's another example using `sigqueue` in the next section.

9.5.5 `sigwaitinfo` and `sigtimedwait`

The `sigwait` system call in Section 9.2.2 isn't part of RTS, but it works just fine on RTS signals, even with queuing. It accepts a pending signal and returns it; if there are still signals of that type pending, they stayed queued. As with signal han-

dlers, if more than one RTS signal (SIGRTMIN through SIGRTMAX) is pending, the highest priority (lowest numbered) signal is accepted first.

But the problem with sigwait is that you can't get the siginfo_t stuff, including the value, which is perhaps the most important RTS feature. So, there's a slightly enhanced system call named sigwaitinfo:

sigwaitinfo—wait for signal

```
#include <signal.h>

int sigwaitinfo(
    const sigset_t *set,       /* signals to wait for */
    siginfo_t *info            /* returned info */
);
/* Returns signal number or -1 on error (sets errno) */
```

If info is NULL, you don't get the information back, and sigwaitinfo is exactly like sigwait except that the signal number is returned as the function value instead of through an argument. If info is non-NULL, it receives a structure just as for an SA_SIGINFO-style signal handler, as described in Section 9.5.1. As with sigwait, make sure the signals in set are blocked; otherwise it's unpredictable what will happen when one is delivered.

Here's an example using queued signals. First, these are definitions for the two RTS signals we'll use:

```
#define MYSIG_COUNT SIGRTMIN
#define MYSIG_STOP SIGRTMIN + 1
```

Here's the function for the thread that waits for a signal, displays its value as a string if it's MYSIG_COUNT, and returns (terminating the thread) if it's MYSIG_STOP:

```
static void *sig_thread(void *arg)
{
    int signum;
    siginfo_t info;

    do {
        signum = sigwaitinfo((sigset_t *)arg, &info);
        if (signum == MYSIG_COUNT)
            printf("Got MYSIG_COUNT; value: %s\n",
              (char *)info.si_value.sival_ptr);
```

```
        else if (signum == MYSIG_STOP) {
            printf("Got MYSIG_STOP; terminating thread\n");
            return (void *)true;
        }
        else
            printf("Got %d\n", signum);
    } while (signum != -1 || errno == EINTR);
    EC_FAIL

EC_CLEANUP_BGN
    EC_FLUSH("sig_thread")
    return (void *)false;
EC_CLEANUP_END
}
```

Note that we couldn't use a `switch` statement because `MYSIG_COUNT` and `MYSIG_STOP` are potentially nonconstant integer expressions, as noted in Section 9.5.2.

Now here's the `main` function:

```
int main(void)
{
    sigset_t set;
    struct sigaction act;
    union sigval value;
    pthread_t tid;

    ec_neg1( sigemptyset(&set) )
    ec_neg1( sigaddset(&set, MYSIG_COUNT) )
    ec_neg1( sigaddset(&set, MYSIG_STOP) )
    ec_rv( pthread_sigmask(SIG_SETMASK, &set, NULL) )
    memset(&act, 0, sizeof(act));
    act.sa_flags = SA_SIGINFO;
    act.sa_sigaction = dummy_handler;
    ec_neg1( sigaction(MYSIG_COUNT, &act, NULL) )
    ec_neg1( sigaction(MYSIG_STOP, &act, NULL) )
    value.sival_ptr = "One";
    ec_neg1( sigqueue(getpid(), MYSIG_COUNT, value) )
    value.sival_ptr = "Two";
    ec_neg1( sigqueue(getpid(), MYSIG_COUNT, value) )
    value.sival_ptr = "Three";
    ec_neg1( sigqueue(getpid(), MYSIG_COUNT, value) )
    value.sival_ptr = NULL;
    ec_neg1( sigqueue(getpid(), MYSIG_STOP, value) )
    ec_rv( pthread_create(&tid, NULL, sig_thread, &set) )
    ec_rv( pthread_join(tid, NULL) )
    exit(EXIT_SUCCESS);
```

```
EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
EC_CLEANUP_END
}

static void dummy_handler(int signum, siginfo_t *info, void *context)
{
}
```

Note that four signals are generated before the thread is even created, to demonstrate that they will be queued. It's guaranteed that the `MYSIG_COUNT` signals will be accepted before `MYSIG_STOP` because they have a lower number. Also note that the signals are blocked and stay blocked. The dummy handler is needed because we must set `SA_SIGINFO` to turn on queuing. It so happens that Solaris queues the RTS signals no matter what, but you can't count on that in a portable program.

Here's the output, proving that the signals were queued:

```
Got MYSIG_COUNT; value: One
Got MYSIG_COUNT; value: Two
Got MYSIG_COUNT; value: Three
Got MYSIG_STOP; terminating thread
```

If you only want to wait for a limited time, you can use `sigtimedwait`, which adds a time-out argument to `sigwaitinfo`:

sigtimedwait—wait for signal

```
#include <signal.h>

int sigtimedwait(
    const sigset_t *set,          /* signals to wait for */
    siginfo_t *info,              /* returned info */
    const struct timespec *ts     /* max. time to wait */
);
/* Returns signal number or -1 on error (sets errno) */
```

`sigtimedwait` waits for at most the number of nanoseconds specified by `ts` for a signal in `set` to become pending and be returned. If the time period expires, it returns –1 with `errno` set to `EAGAIN`. You can't have a `timeout` of `NULL`. If both `timespec` members are zero and no signal in `set` is pending, `sigtimedwait` returns immediately, but this is not really a special case—it has timed-out.

9.5.6 `sigevent` Structure and `SIGEV_THREAD`

The RTS-generation functions other than `sigqueue` are described elsewhere, as listed in Section 9.5.1, but they all use a `sigevent` structure to specify what signal and value to send. I'll explain just the `sigevent` structure here.

`struct sigevent`—structure for RTS-generation functions

```
struct sigevent {
     int sigev_notify;              /* notification type (see below) */
     int sigev_signo;               /* signal number */
     union sigval sigev_value;      /* signal value */
     void (*sigev_notify_function)(union sigval); /* thread function */
     pthread_attr_t *sigev_notify_attributes;     /* thread attributes */
};
```

The signal to be sent (`SIGRTMIN`, say) goes into `sigev_signo`, and the value to go along with it goes into `sigev_value`. As with the other RTS features, it's guaranteed to work only for RTS signals. The `sigev_notify` member specifies how the signal is to be sent:

`SIGEV_SIGNAL` Generate a signal, as though `sigqueue` had been called

`SIGEV_THREAD` Start a thread, as though `pthread_create` had been called

`SIGEV_NONE` Don't provide any notification, which usually doesn't make much sense

`SIGEV_THREAD` is the most interesting. It causes a new thread to be started when the event occurs, every time it occurs, and the function pointed to by `sigev_notify_function` becomes the start function for the thread. Normally, start functions have a pointer-to-`void` argument; in this case it's a pointer-to-`union-sigval`, which is more or less the same, since one of the union members is a `void` pointer. You can set the attributes for the thread with the `sigev_notify_attributes` member, but you can't make the thread joinable. If the member is `NULL`, the thread is detached by default, which is the opposite of what `pthread_create` does.

There's a lot of overhead in creating a thread, so clearly you don't want to specify `SIGEV_THREAD` unless the event is fairly rare and/or its generation represents a large amount of work relative to the cost of starting a new thread. For example, starting a new thread every time input arrives is probably overdoing it.

In case you're confused, creating a thread when a signal would otherwise be generated is completely different from having a thread waiting in `sigwait` (Section 9.2.2). All they have in common is that they both use threads.

9.6 Global Jumps

Normally, a function returns to its caller only by executing a return statement or, if it's a `void` function, by flowing off the end of its outer block, which is the same thing. But it's possible to jump to an arbitrary, but preplanned, point in the program with the Standard C functions `setjmp` and `longjmp`:

setjmp—set jump point

```
#include <setjmp.h>

int setjmp(
    jmp_buf loc_info       /* saved location information */
);
/* Returns 0 if called directly, non-zero if from longjmp (no error
   return) */
```

longjmp—jump to jump point

```
#include <setjmp.h>

void longjmp(
    jmp_buf loc_info,      /* saved location information */
    int val                /* value for setjmp to return */
);
```

You mark the location you want to jump to—the label, so to speak—with a call to `setjmp` that saves whatever location information it needs (e.g., current stack pointer, machine address) in its argument, which doesn't look like a pointer that can receive information, but is. Then, from within any function, no matter how deeply nested, you execute `longjmp` with that same argument and the effect is as if `setjmp` had returned `val`, which can't be zero; if it is, `setjmp` returns 1.

As calling the functions in which `longjmp` appears may have pushed data onto the stack, `longjmp` automatically pops it all off. The function containing the `setjmp` is not "called," and there is no recursion upon the return, although there may have been on the way to calling `longjmp`.

Here's an example. If you've never seen `setjmp` and `longjmp` before, you'll think it's pretty weird:

```
static jmp_buf loc_info;

static void fcn2(void)
{
    printf("In fcn2\n");
    longjmp(loc_info, 1234);
    printf("Leaving fcn2\n");
}

static void fcn1(void)
{
    printf("In fcn1\n");
    fcn2();
    printf("Leaving fcn1\n");
}

int main(void)
{
    int rtn;

    rtn = setjmp(loc_info);
    printf("setjmp returned %d\n", rtn);
    if (rtn == 0)
        fcn1();
    printf("Exiting\n");
    exit(EXIT_SUCCESS);
}
```

And here's the output:

```
setjmp returned 0
In fcn1
In fcn2
setjmp returned 1234
Exiting
```

So you can see that `setjmp` was called once but returned twice and that the jump went straight from the middle of `fcn2` back to the middle of `main`; the two "Leaving" lines weren't printed.

If you don't completely understand, don't worry about it because I'm now going to say this: *Don't ever use `longjmp`*. Here's why:

- While the stack is cleaned up, little else (e.g., allocated memory, open files) is.
- There are some restrictions about where you can use `setjmp` and `longjmp` that will get you into trouble if you forget them.

- Some people (including me) don't even like `goto`s *within* a function. They like `longjmp` much less. It makes programs very difficult to understand and maintain.
- The main use for `longjmp` is from within a signal handler, but it's not async-signal-safe, so you can't use it there unless you can guarantee that the signal handler was invoked by an async-signal-safe function, such as a `kill` executed from within the same process (Section 9.1.7).

If you think you need `longjmp`, it's probably because you haven't designed your functions well enough to allow them to return to their caller, perhaps with an indication of an error or other exceptional condition. Work harder and you'll come up with something.

`longjmp` may or may not restore the signal mask. If you want to force it to be restored, there are variants of `setjmp`/`longjmp` that are identical, except that they can save and restore the signal mask:

sigsetjmp—set jump point

```
#include <setjmp.h>

int sigsetjmp(
    sigjmp_buf loc_info,      /* saved location information */
    int savemask              /* non-zero to save signal mask */
);
/* Returns 0 if called directly, non-zero if from siglongjmp (no error
return) */
```

siglongjmp—jump to jump point, restore signal mask if saved

```
#include <setjmp.h>

void siglongjmp(
    sigjmp_buf loc_info,      /* saved location information */
    int val                   /* value for sigsetjmp to return */
);
```

The signal mask that gets restored is the one that was in effect when `sigsetjmp` was called. If `siglongjmp` is called from within a signal handler, which is what it was designed for, this may be different from what the signal handler would restore were it allowed to return normally.

Since `siglongjmp` is no more async-signal-safe than is `longjmp`, it can't usually be used in a signal handler, and its added ability to restore the signal mask is therefore almost useless.

If you don't want the signal mask to be saved, and for some reason you don't want to just set the `savemask` argument of `sigsetjmp` to zero, you can use yet another pair:

_setjmp—set jump point

```
#include <setjmp.h>

int _setjmp(
    jmp_buf loc_info       /* saved location information */
);
/* Returns 0 if called directly, non-zero if from longjmp (no error
   return) */
```

_longjmp—jump to jump point without restoring signal mask

```
#include <setjmp.h>

void _longjmp(
    jmp_buf loc_info,      /* saved location information */
    int val                /* value for setjmp to return */
);
```

Of course, `_longjmp` isn't async-signal-safe either. Actually, these two underscored functions are just leftovers from an early standard.

9.7 Clocks and Timers

This section describes various system clocks and timers that use those clocks to generate a signal, after a preset interval.

9.7.1 `alarm` System Call

alarm—schedule an alarm signal

```
#include <unistd.h>

unsigned alarm(
    unsigned secs      /* seconds until signal */
);
/* Returns seconds left on previous alarm or zero if none (no error
   return) */
```

Every process has one alarm clock set aside for the `alarm` system call. When the alarm goes off, a `SIGALRM` is sent. A child inherits its parent's alarm clock value, but the actual clock isn't shared. The alarm clock remains set across an `exec`.

`alarm` sets the clock to the number of seconds given by `secs`. The previous setting is returned; it will be 0 if no time remained on the clock previously or if there was no previous alarm. The previous setting is used to restore the clock to the way it was before `alarm` was called.

If `secs` is 0, the alarm clock is turned off. It's important to remember to do this. For example, let's say you have a blocking system call like `read` and you only want to block for a limited time. You catch `SIGALRM` (with an empty handler), call `alarm` for, say, 5 seconds, and then issue the `read`, which blocks. When the alarm goes off, the `SIGALRM` interrupts the blocking system call, and `read` returns –1 with `errno` set to `EINTR`. But if `read` returns sooner than 5 seconds, and you forget to turn off the alarm, it will go off later—*much* later, since 5 seconds is a very long time in computer terms—and maybe mysteriously interrupt something else! You'll know something is amiss if you rigorously check error returns, as we do in this book, but, alas, not everyone does.

Here's an example:

```
int main(void)
{
    struct sigaction act;
    char buf[100];
    ssize_t rtn;

    memset(&act, 0, sizeof(act));
    act.sa_handler = handler;
    ec_neg1( sigaction(SIGALRM, &act, NULL) )
    alarm(5);
    if ((rtn = read(STDIN_FILENO, buf, sizeof(buf) - 1)) == -1) {
        if (errno == EINTR)
            printf("Timed out... type faster next time!\n");
        else
            EC_FAIL
    }
    alarm(0);
    if (rtn == 0)
        printf("Got EOF\n");
    else if (rtn > 0) {
        buf[rtn] = '\0';
        printf("Got %s", buf);
    }
    exit(EXIT_SUCCESS);

EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
EC_CLEANUP_END
}
```

```
static void handler(int signum)
{
}
```

In the following sample interaction, I didn't type anything after the first execution of `alarm_test`; I typed "faster" after the second:

```
$ alarm_test
Timed out... type faster next time!
$ alarm_test
faster
Got faster
$
```

Using `alarm` to interrupt a blocking system call is fine in simple cases, like the one I showed, but in more complicated situations you'll probably be blocking in `select` or `poll`, rather than in `read` directly. If you want to time out, use `pselect` if it's available instead of setting an alarm.

A limitation with `alarm` is that there's only one such alarm clock per process. Sections 9.7.4 and 9.7.6 describe timer system calls with more flexibility.

9.7.2 `sleep` System Call

`sleep` is a familiar function that we've used throughout this book. It blocks a thread for a specified number of seconds:

sleep—suspend execution for seconds or until signal

```
#include <unistd.h>

unsigned sleep(
    unsigned secs          /* seconds to sleep */
);
/* Returns unslept amount */
```

The rules for `sleep` in the SUS contain lots of verbiage that allows for a `SIGALRM` to interfere with it, specifically so `sleep` can be implemented as a function in terms of `alarm` and `pause`, or better, `alarm` and `sigsuspend`. Here's a simple way to implement `sleep`:

```
unsigned aup_sleep(unsigned secs)
{
    struct sigaction act;
    unsigned unslept;
```

```
    memset(&act, 0, sizeof(act));
    act.sa_handler = slp_handler;
    ec_neg1( sigaction(SIGALRM, &act, NULL) )
    alarm(secs);
    pause();
    unslept = alarm(0);
    return unslept;

EC_CLEANUP_BGN
    EC_FLUSH("aup_sleep")
    return 0;
EC_CLEANUP_END
}

static void slp_handler(int signum)
{
}
```

This version works but has some problems:

- If `SIGALRM`s aren't blocked, the arrival of one before the call to `sigaction` might terminate the process.
- If they are blocked, the function won't even work.
- If the alarm goes off between the calls to `alarm` and `pause`, the pause may last forever.
- The old action for `SIGALRM` isn't restored.
- `alarm` and `sleep` are supposed to be compatible.

The last point means that in a sequence like

```
alarm(10);
...
sleep(20);
```

the alarm going off should interrupt the sleep and execute the handler for `SIGALRM`, and in a sequence like

```
alarm(20);
...
sleep(10);
```

the alarm should be set for 10 more seconds after `sleep` returns.

Handling the interaction of `alarm` and `sleep`, preventing an errant `SIGALRM` from terminating the process, preventing an infinite pause, and restoring the old action and mask require a lot of tricky code that has to handle the three possible cases:

- No `alarm` was set when `sleep` was called.
- The remaining `alarm` time is less than or equal to the requested `sleep` time.
- The remaining `alarm` time is greater than the requested `sleep` time.

This version works:[8]

```
unsigned aup_sleep(unsigned secs)
{
    sigset_t set, oset;
    struct sigaction act, oact;
    unsigned prev_alarm, slept, unslept, effective_secs;

    ec_neg1( sigemptyset(&set) )
    ec_neg1( sigaddset(&set, SIGALRM) )
    ec_neg1( sigprocmask(SIG_BLOCK, &set, &oset) )
    prev_alarm = alarm(0);
    if (prev_alarm != 0 && prev_alarm <= secs)
        effective_secs = prev_alarm;
    else {
        memset(&act, 0, sizeof(act));
        act.sa_handler = slp_handler;
        ec_neg1( sigaction(SIGALRM, &act, &oact) )
        effective_secs = secs;
    }
    alarm(effective_secs);
    set = oset;
    ec_neg1( sigdelset(&set, SIGALRM) )
    if (sigsuspend(&set) == -1 && errno != EINTR)
        EC_FAIL
    unslept = alarm(0);
    slept = effective_secs - unslept;
    ec_neg1( sigaction(SIGALRM, &oact, NULL) )
    if (prev_alarm > slept)
        alarm(prev_alarm - slept);
    ec_neg1( sigprocmask(SIG_SETMASK, &oset, NULL) )
    return unslept;

EC_CLEANUP_BGN
    EC_FLUSH("aup_sleep")
    return 0;
EC_CLEANUP_END
}

static void slp_handler(int signum)
{
}
```

8. Geoff Clare contributed to this version by finding bugs in my original attempt.

Here's what's going on, step-by-step:

1. We block `SIGALRM` so we can proceed without worrying about one being delivered. We save the old signal mask in `oset`.
2. We turn off the alarm, in case it's set, and save the remaining time.
3. We calculate how long to sleep (`effective_secs`), reducing the amount requested if the remaining alarm time was less, and we set an alarm for that time.
4. If no alarm was already set or the sleep time is less than the remaining alarm time, we install the empty handler and save the old action in `oact`.
5. We call `sigsuspend` instead of `pause` so that we can atomically unblock `SIGALRM`. We leave the other bits in the mask the way they were on entry to `aup_sleep`.
6. When `sigsuspend` returns, it is because the alarm went off or because some other signal interrupted it. We don't especially care which it was. We do need the amount remaining on the alarm, though, which we get when we turn it off.
7. We calculate the amount of time actually slept.
8. We reset the old action for `SIGALRM`.
9. If the time slept is less than the remaining alarm time on entry, we reset the alarm for the new time remaining (some was slept off).
10. We reset the signal mask.
11. We return the unslept time.

If you can understand this function, you've mastered 90% of what's in this chapter and you can give yourself an A. If you find a mistake and mail it to *aup@basepath.com,* score yourself an A+.

9.7.3 Higher-Resolution Sleeping

`sleep` sleeps for some number of seconds, but that's way too long for many purposes. There are two other calls that sleep for more precise intervals:

usleep—suspend execution for microseconds or until signal

```
#include <unistd.h>

int usleep(
    useconds_t usecs              /* microseconds to sleep */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

`usleep` is almost like `sleep`, except that it's an error to use it to sleep for a full second or longer. That is, `usecs` must be less than a million. If you know you have the Timers option (`_POSIX_TIMERS`), `nanosleep` is even more precise and doesn't have the restriction:

nanosleep—suspend execution for nanoseconds or until signal

```
#include <time.h>

int nanosleep(
    const struct timespec *nsecs, /* nanoseconds to sleep */
    struct timespec *remain       /* remaining time or NULL */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

`nanosleep` takes a `timespec` structure, which was defined in Section 1.7.2. To recap, it has a `tv_sec` member for seconds and a `tv_nsec` member for nanoseconds.

If it was interrupted, it returns –1 with `errno` set to `EINTR`, as usual, and also sets what `remain` points to, to the remaining time. If it returns zero or if `remain` is `NULL`, it doesn't return anything through the argument.

There's another sleeping function in Section 9.7.5 named `clock_nanosleep`.

9.7.4 Basic Interval-Timer System Calls

The `alarm` system call (Section 9.7.1) uses one process-wide interval timer and is in all versions of UNIX. All SUS-compatible systems, and some others, too, also have three more interval timers that you can use independently:

`ITIMER_REAL` Decrements in real time; generates a `SIGALRM` when it expires.

`ITIMER_VIRTUAL` Decrements in process virtual time; generates a `SIGVTALRM` when it expires.

`ITIMER_PROF` Decrements both in process virtual time and when the system is running on behalf of the process; generates a `SIGPROF` when it expires. It is intended for use by profilers, which help tune programs by indicating where they spend their time.

In the next two system calls, the `which` argument must be one of the three listed macros:

getitimer—get value of interval timer

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

int getitimer(
    int which,                    /* timer to get */
    struct itimerval *val         /* returned value */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

setitimer—set value of interval timer

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

int setitimer(
    int which,                    /* timer to set */
    const struct itimerval *val,  /* value to set */
    struct itimerval *oval        /* returned old value */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

struct itimerval—structure for getitimer and setitimer

```
struct itimerval {
    struct timeval it_interval;   /* reset value */
    struct timeval it_value;      /* current value */
};
```

In an `itimerval` structure, `it_interval` is the time to reset the timer to when it expires, and `it_value` is the value of the current interval. That is, unlike `alarm`, which goes off only once, these timers can automatically reset themselves when they go off. If you call `setitimer` with an `it_value` member of zero, it stops the timer immediately. If you call `setitimer` with an `it_interval` member of zero, it stops the timer after the current interval expires. A `timeval` structure, defined in Section 1.7.1, has two members: `tv_sec` for seconds, and `tv_usec` for microseconds.

If the `oval` argument to `setitimer` is non-`NULL`, it gets the old value. Some examples:

```
/* 3.5 sec. timer, one time only */
itv.it_interval.tv_sec = 0;
itv.it_interval.tv_usec = 0;
itv.it_value.tv_sec = 3;
itv.it_value.tv_usec = 500000;
ec_neg1( setitimer(ITIMER_REAL, &itv, NULL) )
```

```
/* 3.5 sec. timer, then repeating 2.25 sec. timers */
itv.it_interval.tv_sec = 2;
itv.it_interval.tv_usec = 250000;
itv.it_value.tv_sec = 3;
itv.it_value.tv_usec = 500000;
ec_neg1( setitimer(ITIMER_REAL, &itv, NULL) )

/* stop timer immediately; interval doesn't matter */
itv.it_value.tv_sec = 0;
itv.it_value.tv_usec = 0;
ec_neg1( setitimer(ITIMER_REAL, &itv, NULL) )
```

In the middle example, the timer would fire at 3.5 seconds, again at 5.75, again at 8, and so on, every 2.25 seconds thereafter until stopped.

Here's an example that displays an X every 2 seconds of real ("wall clock") time. Note that once the timer is set, the rest of the program is free to go about its business, which in the example is just to read from the terminal and echo back what was read:

```
void timer_try1(void)
{
    struct sigaction act;
    struct itimerval itv;
    char buf[100];
    ssize_t nread;

    memset(&act, 0, sizeof(act));
    act.sa_handler = handler;
    ec_neg1( sigaction(SIGALRM, &act, NULL) )
    memset(&itv, 0, sizeof(itv));
    itv.it_interval.tv_sec = 2;
    itv.it_value.tv_sec = 2;
    ec_neg1( setitimer(ITIMER_REAL, &itv, NULL) )
    while (true) {
        switch( nread = read(STDIN_FILENO, buf, sizeof(buf) - 1) ) {
        case -1:
            EC_FAIL
        case 0:
            printf("EOF\n");
            break;
        default:
            if (nread > 0)
                buf[nread] = '\0';
            ec_neg1( write(STDOUT_FILENO, buf, strlen(buf)) )
            continue;
        }
        break;
```

```
    }
    return;

EC_CLEANUP_BGN
    EC_FLUSH("timer_try1")
EC_CLEANUP_END
}

void handler(int signum)
{
    write(STDOUT_FILENO, "\nX\n", 3);
}
```

And this was the output:

```
X
ERROR:  0: tm1 [/aup/c9/tmr.c:26] 0
                *** EINTR (4: "Interrupted system call") ***
```

What happened was that the first X came out OK, after 2 seconds, but the `SIGALRM` signal interrupted the `read`, which was blocked. The fix is to set the `SA_RESTART` flag so system calls won't be interrupted:

```
    memset(&act, 0, sizeof(act));
    act.sa_handler = handler;
    act.sa_flags = SA_RESTART;
    ec_neg1( sigaction(SIGALRM, &act, NULL) )
```

Now the output is this, showing that "hello" was typed and then echoed, along with the Xs every 2 seconds:

```
X

X
hello
X

hello

X
...
```

There's another interval timer system call named `ualarm`, but it's obsolete.

9.7.5 Realtime Clocks

All UNIX systems have a basic clock whose value you can read with the `time` and `gettimeofday` system calls (Section 1.7.1), but the same Timers option

(_POSIX_TIMERS) that brought us nanosleep (Section 9.7.3) includes one or more additional clocks, depending on how many the software and hardware support. To access one of these, you refer to it by its clock ID.

All systems with the Timers option support one ID, CLOCK_REALTIME, which keeps track of the time of day. Whether there are others, what their macros are, and whether they're system-wide or per-process is implementation dependent. For example, Solaris also supports a system-wide CLOCK_HIGHRES clock that counts off from some point in the past. Reading it doesn't give you the time of day since you don't know how it was set, but it's fine for comparing readings at two different times.

You call clock_gettime to get the time in nanoseconds in a returned timespec structure (Sections 1.7.2 and 9.7.3):

clock_gettime—get time from clock

```
#include <time.h>

int clock_gettime(
     clockid_t clock_id,        /* clock ID (CLOCK_REALTIME, etc.) */
     struct timespec *tp        /* time */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

You get the resolution of a clock in nanoseconds with clock_getres:

clock_getres—get clock resolution

```
#include <time.h>

int clock_getres(
     clockid_t clock_id,        /* clock ID */
     struct timespec *res       /* resolution */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

And, with appropriate privileges (superuser for CLOCK_REALTIME), you can set a clock:

clock_settime—set clock

```
#include <time.h>

int clock_settime(
     clockid_t clock_id,        /* clock ID */
     const struct timespec *tp  /* time */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

Here is a function that gets the time and resolution from a clock and then displays it, along with the time in seconds from the `time` system call (Section 1.7.1):

```
void clocks(void)
{
    struct timespec ts;
    time_t tm;

    ec_neg1( time(&tm) )
    printf("time() Time: %ld secs.\n", (long)tm);
    printf("CLOCK_REALTIME:\n");
    ec_neg1( clock_gettime(CLOCK_REALTIME, &ts) )
    printf("Time: %ld.%09ld secs.\n", (long)ts.tv_sec, (long)ts.tv_nsec);
    ec_neg1( clock_getres(CLOCK_REALTIME, &ts) )
    printf("Res.: %ld.%09ld secs.\n", (long)ts.tv_sec, (long)ts.tv_nsec);
    return;

EC_CLEANUP_BGN
    EC_FLUSH("clocks")
EC_CLEANUP_END
}
```

The output on a FreeBSD system was:

```
time() Time: 1051646878 secs.
CLOCK_REALTIME:
Time: 1051646878.568628061 secs.
Res.: 0.000000838 secs.
```

This indicates that the resolution is just under a microsecond. On different hardware, the output on Solaris was:

```
time() Time: 1051646409 secs.
CLOCK_REALTIME:
Time: 1051646409.686869683 secs.
Res.: 0.010000000 secs.
```

There the resolution was only a hundredth of a second. Displaying a time with 9 digits to the right of the decimal point is misleading—it would have been better to limit the display to what the resolution is capable of. However, these clocks aren't for display purposes. They're for timing things.

There's a version of `nanosleep` (Section 9.7.3) that uses a specific clock:

clock_nanosleep—suspend execution for nanoseconds or until signal

```
#include <time.h>

int clock_nanosleep(
     clockid_t clock_id,           /* clock ID */
     int flags,                    /* TIMER_ABSTIME or zero */
     const struct timespec *nsecs, /* nanoseconds to sleep */
     struct timespec *remain       /* remaining time or NULL */
);
/* Returns 0 on success or error number on error */
```

The first argument is the clock ID; the last two are identical to `nanosleep`. If `flags` is zero, you sleep for the specified number of nanoseconds, just as with `nanosleep`. But if it's `TIMER_ABSTIME`, you sleep until the absolute time specified by `nsecs`. The last argument, which returns the time remaining, is used only for relative sleeping; that is, when `flags` is zero.

Finally, you can get a clock ID for a process's CPU-time clock on systems with the Process CPU-Time Clocks option (`_POSIX_CPUTIME`):

clock_getcpuclockid—get process CPU-time clock

```
#include <time.h>

int clock_getcpuclockid(
     pid_t pid,                    /* process ID */
     clockid_t *clock_id           /* returned clock ID */
);
/* Returns 0 on success or error number on error */
```

9.7.6 Advanced Interval-Timer System Calls

Just as the realtime clocks go beyond the basic clocks, so do the advanced interval timers go beyond the interval timers discussed in Section 9.7.4. Instead of just a few system-wide timers, with these calls you can have several interval timers per process, all based on the same clock; recall from the previous section that the only guaranteed clock is `CLOCK_REALTIME`.

You start by creating a timer:

timer_create—create per-process timer

```
#include <signal.h>
#include <time.h>

int timer_create(
     clockid_t clockid,            /* clock ID */
     struct sigevent *sig,         /* NULL or signal to generate */
     timer_t *timer_id             /* returned timer ID */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

If it succeeds, `timer_create` returns a new timer ID through the `timer_id` argument. If non-`NULL`, `sig` specifies a signal to be generated when the timer expires, a value, and how it's to be delivered, as explained in Section 9.5.6. If `sig` is `NULL`, you get a `SIGALRM` generated and the value is set to the timer ID.

You delete a timer with `timer_delete`:

timer_delete—delete per-process timer

```
#include <time.h>

int timer_delete(
     timer_t timer_id              /* timer ID */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

Next come two calls like `getitimer` and `setitimer` (Section 9.7.4), except the resolution in the `itimerspec` structure is in nanoseconds, instead of microseconds:

timer_gettime—get value of per-process timer

```
#include <time.h>

int timer_gettime(
     timer_t timer_id,             /* timer ID */
     struct itimerspec *val        /* returned value */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

timer_settime—set value of per-process timer

```
#include <time.h>

int timer_settime(
     timer_t timer_id,             /* timer ID */
     int flags,                    /* flags */
     const struct itimerspec *val, /* value to set */
     struct itimerspec *oval       /* returned old value or NULL */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

struct itimerspec—structure for timer_ functions

```
struct itimerspec {
     struct timespec it_interval;  /* reset value */
     struct timespec it_value;     /* current value */
};
```

If the `flags` argument to `timer_settime` is zero, the function behaves just like `setitimer`. But if the `TIMER_ABSTIME` flag is set, the `it_value` time is inter-

preted like the `nsecs` argument to `clock_nanosleep`: as an absolute time when the timer should go off. (That is, like the alarm clock next to your bed.) The `it_interval` member is still used to reset the timer when it goes off.

Only one signal will be queued when the timer expires, and if the `it_interval` member is small enough, it's possible for it to expire several times between the time the signal is generated and when it is delivered or accepted. In this case, you can get a count of the number of extra expirations—overruns—with this system call:

timer_getoverrun—get per-process timer overrun count

```
#include <time.h>

int timer_getoverrun(
     timer_t timer_id           /* timer ID */
);
/* Returns overrun count or -1 on error (sets errno) */
```

Exercises

9.1. Investigate what signals are implemented on your system other than the standard 28 listed in Section 9.1.3.

9.2. Write a program that generates a `SIGPIPE` signal (hint: use a pipe) when `write` is called. Then change it to ignore `SIGPIPE`s and verify that `write` returns a write error instead. What is the `errno` value and what does it mean?

9.3. Implement `pause` in terms of `sigwait` and any other system calls you need.

9.4. Try to write `signal` (Section 9.4) in terms of `sigaction`. It's tricky to deal with the handler—see if you can figure out a way.

9.5. Write `sigset` (Section 9.4) in terms of `sigaction`.

9.6. Write a signal handler that does a global jump when a signal arrives (Section 9.6) and demonstrates that execution continues normally from the `setjmp` or `sigsetjmp`. Why are `longjmp` and `siglongjmp` not on the list of async-signal-safe functions (Section 9.1.7)?

9.7. `longjmp` and `siglongjmp` only clean up the stack; they don't deal with memory allocated dynamically by `malloc` and `realloc`. Discuss the ramifications of this. Is an automatic solution possible? Is it desirable? Suggest a possible design change to

(at least) `setjmp`, `sigsetjmp`, `longjmp`, `siglongjmp`, `malloc`, `realloc`, and `free` that handles the problem.

9.8. Write `sleep` in terms of `usleep`, obeying the restriction that `usleep` must sleep for under a second. That is, you must figure out a way to sleep for, say, 3 seconds.

9.9. Using whatever system calls you want from this chapter (e.g., interval-timer calls) and earlier chapters, write a command named `alarmclock` that takes a time and a message as arguments and then, at that time, displays that message on the standard output and rings a bell. Your command should use kernel facilities to wait for the time; other than to set things up, it should not execute until the time arrives. Design it to run in the background even if the user forgets the trailing ampersand.

9.10. Explain (without writing the code) how you would enhance your `alarmclock` command from the previous Exercise to remember its setting even if the system crashes. You also have to figure out how to get the background process restarted. Is there already a UNIX command that does something like this?

9.11. Extend the program you wrote in Exercise 5.14 to include process attributes from Appendix A that are explained in this chapter.

A

Process Attributes

The table in this appendix lists the process attributes that are affected by a `fork` or `exec` and indicates where each is discussed in the book.

Note that the following are not process attributes because they exist independent of any process, as explained in Chapter 7. However, unlike files, their duration is normally only that of the live system—they're lost on a reboot.

POSIX message queues (Section 7.7)
POSIX named semaphores (Section 7.10)
POSIX shared memory segments (Section 7.14)
System V message queue IDs (Section 7.5)
System V message queues (Section 7.5)
System V semaphore set IDs (Section 7.9)
System V semaphore sets (Section 7.9)
System V shared memory segment IDs (Section 7.13)
System V shared memory segments (Section 7.13)

Table A.1 Process Attributes

Attribute	fork	exec	Section
asynchronous I/O operations	not copied	cancelled or completed without notification	3.9
atexit-registered functions	copied	unregistered	1.3.4
controlling terminal	copied	preserved	4.3
directory stream positioning	shared	inaccessible	3.6.1
directory streams	copied	closed	3.6.1

Table A.1 Process Attributes (cont.)

Attribute	fork	exec	Section
directory, current	copied	preserved	3.6.2
directory, root	copied	preserved	5.14
exit status[1]	-	-	5.7
file descriptions	shared	unchanged	2.2
file descriptors	copied	preserved unless FD_CLOEXEC set	2.2
file locks	not copied	preserved unless fd was closed	7.11
file-mode creation mask	copied	preserved	2.4.2
floating-point environment[2]	copied	default	-
ID, effective group	copied	depends on set-group-ID bit	5.12
ID, effective user	copied	depends on set-user-ID bit	5.12
ID, parent process	changed	preserved	5.13
ID, process	new	preserved	5.13
ID, process-group	copied	preserved	4.3
ID, real group	copied	preserved	5.12
ID, real user	copied	preserved	5.12
ID, saved set-group-ID	copied	depends on set-group-ID bit	5.12
ID, saved set-user-ID	copied	depends on set-user-ID bit	5.12
ID, thread (thread)	copied	lost	5.17.1
IDs, supplementary group	copied	preserved	1.1.8
limit, resources (incl. file size)	copied	preserved	5.16

Table A.1 Process Attributes (cont.)

Attribute	fork	exec	Section
memory, data segment	copied	new	1.1.5
memory, data segment, arguments	copied	new	5.3
memory, data segment, environment	copied	new	5.2
memory, instruction segment	copied	new	1.1.5
memory, locks[3]	not copied	removed	-
memory, mappings (POSIX and System V)	copied	unmapped	7.12
memory, stack (thread)	copied	new	5.17
message-queue descriptions (POSIX)	shared	unaffected	7.5
message-queue descriptors (POSIX)	copied	closed	7.5
pipes (FIFO)[4]	copied	preserved	7.2
pipes (unnamed)	copied	preserved	6.2
scheduling, nice value	copied	preserved	5.15
scheduling, SCHED_FIFO and SCHED_RR policies[5]	copied	preserved	-
semaphore, pointer to named (POSIX)	copied	closed	7.10
semaphores, memory (POSIX)	copied	closed	7.10
semaphores, semadj values (System V)	zeroed	preserved	7.9

Table A.1 Process Attributes (cont.)

Attribute	fork	exec	Section
session membership	copied	preserved	4.3
signal actions	copied	preserved, except caught changed to default (`SIGCHLD` is exception)	9.1.6
signal alternate stack	copied	lost; all `SA_ONSTACK` flags cleared	9.3
signal mask (thread)	copied	preserved	9.1.5
signals pending	not copied	preserved	9.1.2
sockets	copied	preserved	8.1
thread attributes, priority, etc.	copied	lost	5.17
thread barriers (thread)	copied	lost	5.17.3
thread condition variables (thread)	copied	lost	5.17.4
thread mutexes (thread)	copied	lost	5.17.3
thread read-write locks (thread)	copied	lost	5.17.3
thread spin lock (thread)	copied	lost	5.17.3
threads (thread)	one copied	terminated	5.17
thread-specific data (thread)	copied	lost	5.17.1
time, CPU	zeroed	preserved	1.7.2
time, CPU-time clock	zeroed	preserved	9.7.5
time, CPU-time clock of thread[6]	zeroed	zeroed	5.17.1

Table A.1 Process Attributes (cont.)

Attribute	fork	exec	Section
timer, alarm	canceled and zeroed	preserved	9.7.1
timers, interval	reset	preserved	9.7.4
timers, per-process	not copied	deleted	9.7.6
tracing attributes	depends[7]	preserved (mostly)	1.1.10

[1]Not set until process exits.
[2]Floating-point environment refers to floating-point status flags and control modes.
[3]Locked and unlocked with `mlock` and `munlock`; part of the Range Memory Locking option.
[4]FIFO itself has same persistence as other files but data in it goes away if no process has it opened for reading.
[5]Set with `sched_setscheduler`; part of the Process Scheduling option.
[6]Set with `pthread_getcpuclockid`; part of the Thread CPU-Time Clocks option.
[7]Depends on trace stream's inheritance policy.

This page intentionally left blank

B

Ux: A C++ Wrapper for Standard UNIX Functions

I've been complaining throughout this book that the standard UNIX API suffers from inconsistent error handling, confusing naming, and redundancies. As I suggested in Section 1.4.3, C++ exceptions provide a much better way to handle errors. As long as I'm going to do that, I might as well rename the functions and organize them into classes, mostly following the categories in Appendix D.

My attempt at a C++ wrapper is called Ux. It was designed to meet these goals:

1. 100% uniform error handling for all functions
2. 100% functionality for the functions that are wrapped
3. Organization into objects that reflect the UNIX architecture
4. Elimination of redundant, obsolete, or defective functions (e.g., `readdir`, `signal`, `mktemp`) where there is an alternative
5. As close to native-C-interface speed as possible

To achieve these goals, these rules were followed in designing Ux:

1. At the level of Standard UNIX Specification v3—no higher, and with identical semantics (other than error handling). That is, not like iostream or Boost[1] threads, or even more elaborate packages like ACE.[2] However, additional functions (e.g., `setblock`, `find_and_open_master`) not in the standard can be included.

1. A C++ threading library (*www.boost.org*).
2. The ADAPTIVE Communication Environment (*www.cs.wustl.edu/~schmidt/ACE.html*).

2. 100% implementation of 290 or so interfaces out of the 1108 that are in the SUS. Excluded are Standard C functions, POSIX threads, user-level functions such as `bsearch` and accounting, spawning, tracing, obsolete functions, and a few others.
3. All errors thrown, never returned. Thrown object is a type that includes `errno` and other error codes (e.g., from `getaddrinfo`).
4. No memory dynamically allocated unless the word "alloc" appears in the member-function name, and very few instances of that.
5. Space and time efficiency very close to native C SUS, except for member-function-call overhead.
6. Objects reflect organization of UNIX and SUS objects (e.g., File, Dir, DirStream, Process). That is, Ux implements UNIX abstractions rather than introducing different ones.
7. Where re-entrant and non-re-entrant functions are defined (e.g., `readdir` and `readdir_r`), usually the re-entrant version is used and the buffer is in the object. If necessary, an "alloc" member function is provided to allocate storage for it (see #4, above).
8. No additional error checking over what the SUS function provides. For example, if a `NULL` pointer is passed to `File::open`, it is passed directly to the underlying `open`. This is because of #5, above.
9. Automatically chooses, for example, `fchown` over `chown` if the object is `open`. Single member function `File::chown` for both.
10. Automatically detects whether an error occurred in some cases. For example, for `pathconf`, a –1 return is an error only if `errno` changed.
11. Some functions coalesced with use of optional arguments. For example, `read`/`pread`, `write`/`pwrite`, `fsync`/`fdatasync`.
12. No copy constructors or assignment operators other than default. Follows from #4, in part.

Here's the Ux class hierarchy:

```
Ux::Base
      Ux::Aio
      Ux::Clock
      Ux::DirStream
```

```
      Ux::ExitStatus
      Ux::File
         Ux::Dir
         Ux::PosixShm
         Ux::Socket
         Ux::Terminal
            Ux::Pty
      Ux::Netdb
      Ux::PosixMsg
      Ux::PosixSem
      Ux::Process
      Ux::Sigset
      Ux::SockAddr
         Ux::SockAddrIn
         Ux::SockAddrUn
      Ux::SockIPv4
      Ux::SockIPv6
      Ux::System
      Ux::SysVIPC
         Ux::SysVMsg
         Ux::SysVSem
         Ux::SysVShm
      Ux::Termios
      Ux::TimeMsec
      Ux::TimeNsec
      Ux::TimeParts
      Ux::Timer
         Ux::IntervalTimer
         Ux::RealtimeTimer
      Ux::TimeSec
      Ux::Timestr
      Ux::TimeString
      Ux::Timet
      Ux::Timetm
      Ux::Timeval
Ux::Error
mq_attr
Ux::semun
```

More details, including complete documentation and the source files, are at *www.basepath.com/aup*. While Ux looks promising, it's only been used for trivial examples, so the jury, of which readers of this book are members, is still out. Let me know what you think (aup@basepath.com).

This page intentionally left blank

C

Jtux: A Java/Jython Interface to Standard UNIX Functions

Thirty years ago, when UNIX was new, almost all UNIX application programming was in C, and C is still the "UNIX language," in that the principal UNIX standards use C, UNIX kernels are written in C, and most UNIX programming books, such as this one, show all their examples in C. Today, however, much, if not most, UNIX programming is in anything but C: C++, Java, Perl, Python, Jython,[1] PHP, and so on.

C++ is a special case because you can use the C interface directly, or a very thin C++ class library such as Ux, described in Appendix B. The other languages typically provide interfaces to some classic UNIX system calls that were in POSIX1990 but not to newer calls like `getaddrinfo` or `msgsnd`. So, serious students who want to learn UNIX are forced to learn C or C++, even if they use Java for all their other course work, and developers don't have ready access to system calls they may need.

To solve the problem for Java, I created Java-to-Unix (Jtux). The goals of Jtux are to provide:

- An educational tool for learning UNIX programming with Java or Jython.
- Access to UNIX features that are currently unsupported by Java packages.
- Consistent error handling. An error from a system call, no matter how it's indicated (–1 return, `NULL` return, error-code return, etc.) always results in throwing a `UErrorException`.

Jtux does *not*:

1. Jython is an implementation of Python, written in Java, that runs in the Java environment, so any Java class can be accessed directly. For example, in Jython you write your user interface using AWT or Swing, rather than with Tkinter. That's why Jtux works with Jython just as well as with Java. For more on Jython, go to *www.jython.org*.

- Provide an object-oriented view of the UNIX API.
- Fix any of the inconsistencies in UNIX system-call naming or arguments, or clean up most of the redundancies, as Ux (Appendix B) does. In fact, nearly every Jtux method has the same name and the same arguments as the corresponding POSIX/SUS call because otherwise Jtux wouldn't work to help students learn to program UNIX. (Jtux does fix the error-handling inconsistencies.)
- Provide portability other than that already provided by POSIX/SUS. Jtux does not work in Java/Jython environments that aren't hosted on a POSIX/SUS system, such as Windows.
- Work in applets because the Java Native Interface (JNI) that was used to connect Java with the C API does not.

If you want an object-oriented approach to UNIX facilities that's more in the spirit of Java, you're better off using other Java packages. For example, use `java.net.Socket`, instead of `jtux.UNetwork`. However, Jtux is the most complete UNIX package for Java or Jython, by a very wide margin.

Jtux covers these 186 system calls:

abort	getppid	poll	setpgid
accept	getrlimit	pread	setrlimit
access	getrusage	pselect	setsid
alarm	getsid	pthread_sigmask	setsockopt
bind	getsockopt	putenv	setuid
chdir	getuid	pwrite	shm_open
chmod	htonl	read	shm_unlink
chown	htons	readdir	shmat
chroot	inet_ntop	readlink	shmctl
clock	inet_pton	readv	shmdt
close	kill	recv	shmget

closedir	lchown	recvfrom	sigaction
connect	link	recvmsg	sigaddset
creat	listen	rename	sigaltstack
dup	lockf	rewinddir	sigdelset
dup2	lseek	rmdir	sigemptyset
execvp	lstat	S_ISBLK	sigfillset
_exit	mkdir	S_ISCHR	siginterrupt
exit	mkfifo	S_ISDIR	sigismember
fchdir	mknod	S_ISFIFO	sigpending
fchmod	mkstemp	S_ISLNK	sigprocmask
fchown	mmap	S_ISREG	sigqueue
fcntl	mq_close	S_ISSOCK	sigsuspend
FD_CLR	mq_getattr	seekdir	sigtimedwait
FD_ISSET	mq_notify	select	sigwait
FD_SET	mq_open	sem_close	sigwaitinfo
FD_ZERO	mq_open	sem_destroy	sleep
fdatasync	mq_receive	sem_getvalue	sockatmark
fork	mq_send	sem_init	socket
freeaddrinfo	mq_setattr	sem_open	stat
fstat	mq_timedreceive	sem_open	statvfs
fstatvfs	mq_timedsend	sem_post	symlink
fsync	mq_unlink	sem_timedwait	sync

```
ftok            msgctl          sem_trywait     system
ftruncate       msgget          sem_unlink      telldir
gai_strerror    msgrcv          sem_wait        times
getaddrinfo     msgsnd          semctl          truncate
getcwd          munmap          semget          umask
getegid         nanosleep       semop           unlink
getenv          nice            send            unsetenv
geteuid         ntohl           sendmsg         usleep
getgid          ntohs           sendto          utime
gethostid       open            setegid         wait
gethostname     open            setenv          waitpid
getnameinfo     opendir         seteuid         write
getpgid         pause           setgid          writev
getpid          pipe
```

What's here is all POSIX/SUS calls dealing with files, file systems, directories, sockets, pipes, processes, System V messages/semaphores/shared memory, POSIX messages/semaphores/shared memory, signals, environment, and more. I've tried very hard to be complete; for example, there's not only `write`, but also `pwrite` and `writev`; there's not only `send`, but also `sendmsg` and `sendto`. Every argument of every included function and every field of every structure is supported, except where technically impossible. Functions for terminal I/O, pseudo terminals, interval timers (other than `alarm`), and realtime clocks weren't included, although they could have been. You can add these if you like.

Jython is a terrific system in which to try out UNIX system calls. To show you what Jython and Jtux are like, here's a program that has two processes communicating using datagrams (see Section 8.6):

```
# Jython example showing use of AF_INET sockets with SOCK_DGRAMs.
# System calls used: bind, close, _exit, fork, getaddrinfo,
# recvfrom, sendto, setsockopt, sleep, socket, waitpid
import jtux.UClock as UClock
import jtux.UConstant as UConstant
import jtux.UErrorException as UErrorException
import jtux.UFile as UFile
import jtux.UNetwork as UNetwork
import jtux.UProcess as UProcess
import jtux.UUtil as UUtil
import java.lang
import jarray
def b_to_s(ba): # convert byte array to string
    s = ""
    for b in ba:
        if b == 0:
            break
        s += chr(b)
    return s
nodename = "localhost"
servname1 = "5431" # unused port for peer 1
servname2 = "5432" # unused port for peer 2
msgsize = 300
hint = UNetwork.s_addrinfo()
hint.ai_family = UConstant.AF_INET
hint.ai_socktype = UConstant.SOCK_DGRAM
infop = UNetwork.AddrInfoListHead()
UNetwork.getaddrinfo(nodename, servname2, hint, infop)
sa_peer2 = infop.ai_next.ai_addr; # both peers need this socket addr
int_opt = UNetwork.SockOptValue_int() # Jtux's way of handling setsockopt values
int_opt.value = 1
pid = UProcess.fork()
if pid == 0: # Peer 1
    msg = jarray.zeros(msgsize, 'b') # jarray is a standard Jython module
    UNetwork.getaddrinfo(nodename, servname1, hint, infop)
    sa_peer1 = infop.ai_next.ai_addr
    UClock.sleep(1); # let peer 2 startup first -- not the best approach
    fd_skt = UNetwork.socket(UConstant.AF_INET, UConstant.SOCK_DGRAM, 0)
    UNetwork.setsockopt(fd_skt, UConstant.SOL_SOCKET, UConstant.SO_REUSEADDR,
      int_opt, 0)
    UNetwork.bind(fd_skt, sa_peer1, 0)
    sa_sender = UNetwork.s_sockaddr_in()
    sa_len = UUtil.IntHolder(); # Jtux class for passing ints by reference
    maxmsgs = 4
    for i in xrange(maxmsgs + 1):
        if i == maxmsgs:
            m = "Stop"
        else:
            m = "Message #" + str(i)
```

```
        UNetwork.sendto(fd_skt, m, len(m), 0, sa_peer2, 0)
        if i == maxmsgs:
            break
        nrcv = UNetwork.recvfrom(fd_skt, msg, len(msg), 0, sa_sender, sa_len)
        print "Peer 1 got \"" + b_to_s(msg) + "\" from " + str(sa_sender)
    UFile.close(fd_skt)
    print "Peer 1 exiting"
    UProcess._exit(UConstant.EXIT_SUCCESS)
else: # Peer 2
    msg = jarray.zeros(msgsize, 'b') # jarray is a standard Jython module
    fd_skt = UNetwork.socket(UConstant.AF_INET, UConstant.SOCK_DGRAM, 0)
    UNetwork.setsockopt(fd_skt, UConstant.SOL_SOCKET, UConstant.SO_REUSEADDR,
      int_opt, 0)
    UNetwork.bind(fd_skt, sa_peer2, 0)
    sa_sender = UNetwork.s_sockaddr_in()
    sa_len = UUtil.IntHolder()
    while 1:
        nrcv = UNetwork.recvfrom(fd_skt, msg, len(msg), 0, sa_sender, sa_len)
        if b_to_s(msg[:4]) == "Stop":
            break
        print "Peer 2 got \"" + b_to_s(msg[:nrcv]) + "\" from " + str(sa_sender)
        msg[0] = ord('m')
        UNetwork.sendto(fd_skt, msg, len(msg), 0, sa_sender, sa_len.value)
    UFile.close(fd_skt)
    UProcess.waitpid(pid, None, 0)
    print "Peer 2 exiting"
```

And here's a Java example:

```
// Java example that recursively descends the directory tree.
import jtux.*;
class TreeList {
    static void list(String entry, int level) {
        int dir_fd = -1;
        try {
            String tabs = "";
            for (int i = 0; i < level; i++)
                tabs += "\t";
            long dir = -1;
            try {
                dir = UDir.opendir(entry);
            }
            catch (UErrorException e) {
                System.out.println(tabs + entry + " - " + e);
                return;
            }
            dir_fd = UFile.open(".", UConstant.O_RDONLY);
            UProcess.chdir(entry);
            UDir.s_dirent dirent = new UDir.s_dirent();
            UFile.s_stat sbuf = new UFile.s_stat();
```

```
            while ((dirent = UDir.readdir(dir)) != null) {
                UFile.lstat(dirent.d_name, sbuf);
                if (UFile.S_ISDIR(sbuf.st_mode) && !dirent.d_name.equals(".") &&
                  !dirent.d_name.equals("..")) {
                    System.out.println(tabs + dirent.d_name + ":");
                    if (UFile.S_ISLNK(sbuf.st_mode)) {
                        System.out.println(tabs + "[symbolic link -- skipping]");
                    }
                    else
                        list(dirent.d_name, level + 1);
                }
                else
                    System.out.println(tabs + dirent.d_name);
            }
            UDir.closedir(dir);
            UProcess.fchdir(dir_fd);
            UFile.close(dir_fd);
            dir_fd = -1;
        }
        catch (UErrorException e) {
            try {
                if (dir_fd != -1)
                    UProcess.fchdir(dir_fd);
            }
            catch (Exception edummy) {
            }
            System.out.println(e);
        }
    }
    public static void main(String args[]) {
        list("/", 0);
    }
}
```

Blending the UNIX C API with Java isn't entirely smooth. Java doesn't have things like pointers. Implementation-defined constants like `ENOSYS` or `O_CREAT` aren't visible to Java programs. Go to *www.basepath.com/aup* for details about how these difficulties and others were overcome, how to build and install Jtux, and, of course, to download the source code itself.

This page intentionally left blank

D

Alphabetical and Categorical Function Lists

This appendix lists the 307 functions covered in this book alphabetically and by category, along with the section where the function is explained.

Functions Alphabetically

FD_CLR—clear fd_set bit (4.2.3)

FD_ISSET—test fd_set bit (4.2.3)

FD_SET—set fd_set bit (4.2.3)

FD_ZERO—clear entire fd_set (4.2.3)

_Exit—terminate process without cleanup (5.7)

_exit—terminate process without cleanup (5.7)

_longjmp—jump to jump point without restoring signal mask (9.6)

_setjmp—set jump point (9.6)

abort—generate SIGABRT (9.1.9)

accept—accept new connection on socket and create new socket (8.1.2)

access—determine accessibility of file (3.8.1)

aio_cancel—cancel asynchronous I/O request (3.9.5)

aio_error—retrieve error status for asynchronous I/O operation (3.9.4)

aio_fsync—initiate buffer-cache flushing for one file (3.9.6)

aio_read—asynchronous read from file (3.9.3)

aio_return—retrieve return status of asynchronous I/O operation (3.9.4)

aio_suspend—wait for asynchronous I/O request (3.9.7)

aio_write—asynchronous write to file (3.9.3)

alarm—schedule an alarm signal (9.7.1)

asctime—convert broken-down time to local-time string (1.7.1)

atexit—register function to be called when process exits (1.3.4)

bind—bind name to socket (8.1.2)

cfgetispeed—get input speed from termios structure (4.5.3)

cfgetospeed—get output speed from termios structure (4.5.3)

cfsetispeed—set input speed in termios structure (4.5.3)

cfsetospeed—set output speed in termios structure (4.5.3)

chdir—change current directory by path (3.6.2)

chmod—change mode of file by path (3.7.1)

chown—change owner and group of file by path (3.7.2)

chroot—change root directory (5.14)

clock—get execution time (1.7.2)

clock_getcpuclockid—get process CPU-time clock (9.7.5)

clock_getres—get clock resolution (9.7.5)

clock_gettime—get time from clock (9.7.5)

clock_nanosleep—suspend execution for nanoseconds or until signal (9.7.5)

clock_settime—set clock (9.7.5)

close—close file descriptor (2.11)

closedir—close directory (3.6.1)

confstr—get configuration string (1.5.7)

connect—connect socket (8.1.2)

creat—create or truncate file for writing (2.4.2)

ctermid—get pathname for controlling terminal (4.7)

ctime—convert time_t to local-time string (1.7.1)

difftime—subtract two time_t values (1.7.1)

dup—duplicate file descriptor (6.3)

dup2—duplicate file descriptor (6.3)

endhostent—end host-database scan (8.8.1)

endnetent—end network-database scan (8.8.2)

endprotoent—end protocol-database scan (8.8.3)

endservent—end service-database scan (8.8.4)

execl—execute file with argument list (5.3)

execle—execute file with argument list and environment (5.3)

execlp—execute file with argument list and PATH search (5.3)

execv—execute file with argument vector (5.3)

execve—execute file with argument vector and environment (5.3)

execvp—execute file with argument vector and PATH search (5.3)

exit—terminate process with cleanup (5.7)

fchdir—change current directory by file descriptor (3.6.2)

fchmod—change mode of file by file descriptor (3.7.1)

fchown—change owner and group of file by file descriptor (3.7.2)

fcntl—control open file (3.8.3)

fdatasync—force buffer-cache flushing for one file's data (2.16.2)

fork—create new process (5.5)

fpathconf—get system option or limit by file descriptor (1.5.6)

freeaddrinfo—free socket-address information (8.2.6)

fstat—get file info by file descriptor (3.5.1)

fstatvfs—get file system information by file descriptor (3.2.3)

fsync—schedule or force buffer-cache flushing for one file (2.16.2)

ftok—generate System V IPC key (7.4.2)

ftruncate—truncate or stretch file by file descriptor (2.17)

gai_strerror—get error-code description (8.2.6)

getaddrinfo—get socket-address information (8.2.6)

getcwd—get current directory pathname (3.4.2)

getdate—convert string to broken-down time with rules (1.7.1)

getegid—get effective group ID (5.11)

getenv—get value of environment variable (5.2)

geteuid—get effective user ID (5.11)

getgid—get real group ID (5.11)

getgrgid—get group-file entry (3.5.2)

gethostbyaddr—look up host by address (8.8.1)

gethostbyname—look up host by name (8.8.1)

gethostent—get next host-database entry (8.8.1)

gethostid—get identifier for local host (8.8.1)

gethostname—get name of host (8.2.7)

getitimer—get value of interval timer (9.7.4)

getlogin—get login name (3.5.2)

getnameinfo—get name information (8.8.1)

getnetbyaddr—look up network by number (8.8.2)

getnetbyname—look up network by name (8.8.2)

getnetent—get network-database entry (8.8.2)

getpeername—get socket address of connected socket (8.9.2)

getpgid—get process-group ID (4.3.3)

getpid—get process ID (5.13)

getppid—get parent process ID (5.13)

getprotobyname—look up protocol by name (8.8.3)

getprotobynumber—look up protocol by number (8.8.3)

getprotoent—get protocol-database entry (8.8.3)

getpwuid—get password-file entry (3.5.2)

getrlimit—get resource limits (5.16)

getrusage—get resource usage (5.16)

getservbyname—look up service by name (8.8.4)

getservbyport—look up service by port (8.8.4)

getservent—get service-database entry (8.8.4)

getsid—get session ID (4.3.2)

getsockname—get socket address (8.9.2)

getsockopt—get socket options (8.3)

gettimeofday—get current date and time as timeval (1.7.1)

getuid—get real user ID (5.11)

gmtime—convert time_t to UTC broken-down time (1.7.1)

grantpt—get access to slave side of pty (4.10.1)

htonl—convert 32-bit value from host to network byte order (8.1.4)

htons—convert 16-bit value from host to network byte order (8.1.4)

if_freenameindex—free array allocated by if_nameindex (8.8.5)

if_indextoname—map network interface index to name (8.8.5)

if_nameindex—get all network interface names and indexes (8.8.5)

if_nametoindex—map network interface name to index (8.8.5)

inet_addr—convert dotted IPv4 address string to integer (8.2.3)

inet_ntoa—convert integer IPv4 address to dotted string (8.2.3)

inet_ntop—convert IPv4 or IPv6 binary addresses to string (8.9.5)

inet_pton—convert IPv4 or IPv6 string addresses to binary (8.9.5)

ioctl—control character device (4.4)

isatty—test for terminal (4.7)

kill—generate signal for processes (9.1.9)

killpg—generate signal for process group (9.1.9)

lchown—change owner and group of symbolic link by path (3.7.2)

link—create hard link (3.3.1)

lio_listio—list-directed I/O (3.9.9)

listen—mark socket for accepting and set queue limit (8.1.2)

localtime—convert time_t to local broken-down time (1.7.1)

lockf—lock section of file (7.11.3)

longjmp—jump to jump point (9.6)

lseek—set and get file offset (2.13)

lstat—get file info by path without following symbolic link (3.5.1)

mkdir—make directory (3.6.3)

mkfifo—make FIFO (7.2.1)

mknod—make file (3.8.2)

mkstemp—create and open file with unique name (2.7)

mktime—convert local broken-down time to time_t (1.7.1)

mmap—map pages of memory (7.14.1)

mount—mount file system (nonstandard) (3.2.4)

mq_close—close message queue (7.7.1)

mq_getattr—get message-queue attributes (7.7.1)

mq_notify—register or unregister for message notification (7.7.1)

mq_open—open message queue (7.7.1)

mq_receive—receive message (7.7.1)

mq_send—send message (7.7.1)

mq_setattr—set message queue attributes (7.7.1)

mq_timedreceive—receive message with timeout (7.7.1)

mq_timedsend—send message with timeout (7.7.1)

mq_unlink—remove message queue (7.7.1)

msgctl—control message queue (7.5.1)

msgget—get message-queue identifier (7.5.1)

msgrcv—receive message (7.5.1)

msgsnd—send message (7.5.1)

munmap—unmap pages of memory (7.14.1)

nanosleep—suspend execution for nanoseconds or until signal (9.7.3)

nice—change nice value (5.15)

ntohl—convert 32-bit value from network to host byte order (8.1.4)

ntohs—convert 16-bit value from network to host byte order (8.1.4)

open—open or create file (2.4)

opendir—open directory (3.6.1)

pathconf—get system option or limit by path (1.5.6)

pause—wait for signal (9.2.1)

pipe—create pipe (6.2.1)

poll—wait for I/O to be ready (4.2.4)

posix_openpt—open pty (4.10.1)

pread—read from file descriptor at offset (2.14)

pselect—wait for I/O to be ready (4.2.3)

pthread_cancel—cancel thread (5.17.5)

pthread_cleanup_pop—uninstall cleanup handler (5.17.5)

pthread_cleanup_push—install cleanup handler (5.17.5)

pthread_cond_signal—signal condition (5.17.4)

pthread_cond_wait—wait for condition (5.17.4)

pthread_create—create thread (5.17.1)

pthread_join—wait for thread to terminate (5.17.2)

pthread_kill—generate signal for thread (9.1.9)

pthread_mutex_lock—lock mutex (5.17.3)

pthread_mutex_unlock—unlock mutex (5.17.3)

pthread_sigmask—change thread's signal mask (9.1.5)

pthread_testcancel—test for cancellation (5.17.5)

ptsname—get name of slave side of pty (4.10.1)

putenv—change or add to environment (5.2)

pwrite—write to file descriptor at offset (2.14)

raise—generate signal for thread (9.1.9)

read—read from file descriptor (2.10)

readdir—read directory (3.6.1)

readdir_r—read directory (3.6.1)

readlink—read symbolic link (3.3.3)

readv—scatter read (2.15)

recv—receive data from socket (8.9.1)

recvfrom—receive message from socket (8.6.2)

recvmsg—receive message from socket (8.6.3)

rename—rename file (3.3.2)

rewinddir—rewind directory (3.6.1)

rmdir—remove directory (3.6.3)

seekdir—seek directory (3.6.1)

select—wait for I/O to be ready (4.2.3)

sem_close—close named semaphore (7.10.1)

sem_destroy—destroy unnamed semaphore (7.10.2)

sem_getvalue—get value of semaphore (7.10.1)

sem_init—initialize unnamed semaphore (7.10.2)

sem_open—open named semaphore (7.10.1)

sem_post—increment semaphore (7.10.1)

sem_timedwait—decrement semaphore (7.10.1)

sem_trywait—decrement semaphore if possible (7.10.1)

sem_unlink—remove named semaphore (7.10.1)

sem_wait—decrement semaphore (7.10.1)

semctl—control semaphore set (7.9.1)

semget—get semaphore-set identifier (7.9.1)

semop—operate on semaphore set (7.9.2)

send—send data to socket (8.9.1)

sendmsg—send message to socket using message structure (8.6.3)

sendto—send message to socket (8.6.2)

setegid—set effective group ID (5.12)

setenv—change or add to environment (5.2)

seteuid—set effective user ID (5.12)

setgid—set real, effective, and saved group ID (5.12)

sethostent—start host-database scan (8.8.1)

setitimer—set value of interval timer (9.7.4)

setjmp—set jump point (9.6)

setnetent—start network-database scan (8.8.2)

setpgid—set or create process-group (4.3.3)

setprotoent—start protocol-database scan (8.8.3)

setrlimit—set resource limits (5.16)

setservent—start service-database scan (8.8.4)

setsid—create session and process group (4.3.2)

setsockopt—set socket options (8.3)

setuid—set real, effective, and saved user ID (5.12)

shm_open—open shared-memory object (7.14.1)

shm_unlink—remove shared-memory object (7.14.1)

shmat—attach shared memory segment (7.13.1)

shmctl—control shared memory segment (7.13.1)

shmdt—detach shared memory segment (7.13.1)

shmget—get shared memory segment (7.13.1)

shutdown—shut down socket send and/or receive operations (8.9.4)

sigaction—set signal action (9.1.6)

sigaddset—add signal to signal set (9.1.5)

sigaltstack—set and get alternate stack context (9.3)

sigdelset—delete signal from signal set (9.1.5)

sigemptyset—initialize empty signal set (9.1.5)

sigfillset—initialize full signal set (9.1.5)

sighold—block signal (9.4)

sigignore—ignore signal (9.4)

siginterrupt—set or clear SA_RESTART flag (9.3)

sigismember—test for signal in signal set (9.1.5)

siglongjmp—jump to jump point, restore signal mask if saved (9.6)

signal—set signal action (9.4)

sigpause—change signal mask and wait for signal (9.4)

sigpending—examine pending signals (9.3)

sigprocmask—change thread's signal mask (single thread only) (9.1.5)

sigqueue—generate signal for process (9.5.4)

sigrelse—unblock signal (9.4)

sigset—signal management (9.4)

sigsetjmp—set jump point (9.6)

sigsuspend—change signal mask and wait for signal (9.2.3)

sigtimedwait—wait for signal (9.5.5)

sigwait—wait for signal (9.2.2)

sigwaitinfo—wait for signal (9.5.5)

sleep—suspend execution for seconds or until signal (9.7.2)

sockatmark—test presence of out-of-band mark (8.7)

socket—create endpoint for communication (8.1.2)

socketpair—create pair of sockets (8.9.3)

stat—get file info by path (3.5.1)

statvfs—get file system information by path (3.2.3)

strftime—convert broken-down time to string with format (1.7.1)

strptime—convert string to broken-down time with format (1.7.1)

symlink—create symbolic link (3.3.3)

sync—schedule buffer-cache flushing (2.16.2)

sysconf—get system option or limit (1.5.5)

system—run command (5.5)

tcdrain—drain (wait for) terminal output (4.6)

tcflow—suspend or restart flow of terminal input or output (4.6)

tcflush—flush (throw away) terminal output, input, or both (4.6)

tcgetattr—get terminal attributes (4.5.1)

tcgetpgrp—get foreground process-group ID (4.3.4)

tcgetsid—get session ID (4.3.4)

tcsendbreak—send break to terminal (4.6)

tcsetattr—set terminal attributes (4.5.1)

tcsetpgrp—set foreground process-group ID (4.3.4)

telldir—get directory location (3.6.1)

time—get current date and time as time_t (1.7.1)

timer_create—create per-process timer (9.7.6)

timer_delete—delete per-process timer (9.7.6)

timer_getoverrun—get per-process timer overrun count (9.7.6)

timer_gettime—get value of per-process timer (9.7.6)

timer_settime—set value of per-process timer (9.7.6)

times—get process and child-process execution times (1.7.2)

truncate—truncate or stretch file by path (2.17)

ttyname—find pathname of terminal (4.7)

ttyname_r—find pathname of terminal (4.7)

tzset—set time zone information (1.7.1)

ulimit—get and set process limits (5.16)

umask—set and get file mode creation mask (2.5)

umount—unmount file system (non-standard) (3.2.4)

uname—get info about current system (8.8.1)

unlink—remove directory entry (2.6)

unlockpt—unlock pty (4.10.1)

unsetenv—remove environment variable (5.2)

usleep—suspend execution for microseconds or until signal (9.7.3)

utime—set file access and modification times (3.7.3)

vfork—create new process; share memory (obsolete) (5.5)

wait—wait for child process to terminate (5.8)

waitid—wait for child process to change state [X/Open] (5.8)

waitpid—wait for child process to change state (5.8)

wcsftime—convert broken-down time to wide-character string with format (1.7.1)

write—write to file descriptor (2.9)

writev—scatter write (2.15)

Functions Categorically

Configuration

confstr—get configuration string (1.5.7)

fpathconf—get system option or limit by file descriptor (1.5.6)

gethostid—get identifier for local host (8.8.1)

gethostname—get name of host (8.2.7)

pathconf—get system option or limit by path (1.5.6)

sysconf—get system option or limit (1.5.5)

uname—get info about current system (8.8.1)

Directory I/O

closedir—close directory (3.6.1)

opendir—open directory (3.6.1)

readdir—read directory (3.6.1)

readdir_r—read directory (3.6.1)

rewinddir—rewind directory (3.6.1)

seekdir—seek directory (3.6.1)

telldir—get directory location (3.6.1)

Directory Management

link—create hard link (3.3.1)

mkdir—make directory (3.6.3)

readlink—read symbolic link (3.3.3)

rename—rename file (3.3.2)

rmdir—remove directory (3.6.3)

symlink—create symbolic link (3.3.3)

unlink—remove directory entry (2.6)

File Attributes

access—determine accessibility of file (3.8.1)

chmod—change mode of file by path (3.7.1)

chown—change owner and group of file by path (3.7.2)

fchmod—change mode of file by file descriptor (3.7.1)

fchown—change owner and group of file by file descriptor (3.7.2)

fstat—get file info by file descriptor (3.5.1)

lchown—change owner and group of symbolic link by path (3.7.2)

lstat—get file info by path without following symbolic link (3.5.1)

stat—get file info by path (3.5.1)

utime—set file access and modification times (3.7.3)

File I/O

close—close file descriptor (2.11)

creat—create or truncate file for writing (2.4.2)

dup—duplicate file descriptor (6.3)

dup2—duplicate file descriptor (6.3)

fcntl—control open file (3.8.3)

fdatasync—force buffer-cache flushing for one file's data (2.16.2)

fsync—schedule or force buffer-cache flushing for one file (2.16.2)

ftruncate—truncate or stretch file by file descriptor (2.17)

lseek—set and get file offset (2.13)

mkstemp—create and open file with unique name (2.7)

open—open or create file (2.4)

pipe—create pipe (6.2.1)

poll—wait for I/O to be ready (4.2.4)

pread—read from file descriptor at offset (2.14)

pselect—wait for I/O to be ready (4.2.3)

pwrite—write to file descriptor at offset (2.14)

read—read from file descriptor (2.10)

readv—scatter read (2.15)

select—wait for I/O to be ready (4.2.3)

sync—schedule buffer-cache flushing (2.16.2)

truncate—truncate or stretch file by path (2.17)

write—write to file descriptor (2.9)

writev—scatter write (2.15)

File I/O (Asynchronous)

aio_cancel—cancel asynchronous I/O request (3.9.5)

aio_error—retrieve error status for asynchronous I/O operation (3.9.4)

aio_fsync—initiate buffer-cache flushing for one file (3.9.6)

aio_read—asynchronous read from file (3.9.3)

aio_return—retrieve return status of asynchronous I/O operation (3.9.4)

aio_suspend—wait for asynchronous I/O request (3.9.7)

aio_write—asynchronous write to file (3.9.3)

lio_listio—list-directed I/O (3.9.9)

File Management

lockf—lock section of file (7.11.3)

mkfifo—make FIFO (7.2.1)

mknod—make file (3.8.2)

File System

fstatvfs—get file system information by file descriptor (3.2.3)

mount—mount file system (nonstandard) (3.2.4)

statvfs—get file system information by path (3.2.3)

umount—unmount file system (nonstandard) (3.2.4)

File-Descriptor Set

FD_CLR—clear fd_set bit (4.2.3)

FD_ISSET—test fd_set bit (4.2.3)

FD_SET—set fd_set bit (4.2.3)

FD_ZERO—clear entire fd_set (4.2.3)

IPC—POSIX Message Queue

mq_close—close message queue (7.7.1)

mq_getattr—get message-queue attributes (7.7.1)

mq_notify—register or unregister for message notification (7.7.1)

mq_open—open message queue (7.7.1)

mq_receive—receive message (7.7.1)

mq_send—send message (7.7.1)

mq_setattr—set message queue attributes (7.7.1)

mq_timedreceive—receive message with timeout (7.7.1)

mq_timedsend—send message with timeout (7.7.1)

mq_unlink—remove message queue (7.7.1)

IPC—POSIX Semaphore

sem_close—close named semaphore (7.10.1)

sem_destroy—destroy unnamed semaphore (7.10.2)

sem_getvalue—get value of semaphore (7.10.1)

sem_init—initialize unnamed semaphore (7.10.2)

sem_open—open named semaphore (7.10.1)

sem_post—increment semaphore (7.10.1)

sem_timedwait—decrement semaphore (7.10.1)

sem_trywait—decrement semaphore if possible (7.10.1)

sem_unlink—remove named semaphore (7.10.1)

sem_wait—decrement semaphore (7.10.1)

IPC—POSIX Shared Memory

mmap—map pages of memory (7.14.1)

munmap—unmap pages of memory (7.14.1)

shm_open—open shared-memory object (7.14.1)

shm_unlink—remove shared-memory object (7.14.1)

IPC—System V Message Queue

ftok—generate System V IPC key (7.4.2)

msgctl—control message queue (7.5.1)

msgget—get message-queue identifier (7.5.1)

msgrcv—receive message (7.5.1)

msgsnd—send message (7.5.1)

IPC—System V Semaphore

semctl—control semaphore set (7.9.1)

semget—get semaphore-set identifier (7.9.1)

semop—operate on semaphore set (7.9.2)

IPC—System V Shared Memory

shmat—attach shared memory segment (7.13.1)

shmctl—control shared memory segment (7.13.1)

shmdt—detach shared memory segment (7.13.1)

shmget—get shared memory segment (7.13.1)

Network Database

endhostent—end host-database scan (8.8.1)

endnetent—end network-database scan (8.8.2)

endprotoent—end protocol-database scan (8.8.3)

endservent—end service-database scan (8.8.4)

gethostent—get next host-database entry (8.8.1)

getnetbyaddr—look up network by number (8.8.2)

getnetbyname—look up network by name (8.8.2)

getnetent—get network-database entry (8.8.2)

getprotobyname—look up protocol by name (8.8.3)

getprotobynumber—look up protocol by number (8.8.3)

getprotoent—get protocol-database entry (8.8.3)

getservbyname—look up service by name (8.8.4)

getservbyport—look up service by port (8.8.4)

getservent—get service-database entry (8.8.4)

htonl—convert 32-bit value from host to network byte order (8.1.4)

htons—convert 16-bit value from host to network byte order (8.1.4)

if_freenameindex—free array allocated by if_nameindex (8.8.5)

if_indextoname—map network interface index to name (8.8.5)

if_nameindex—get all network interface names and indexes (8.8.5)

if_nametoindex—map network interface name to index (8.8.5)

ntohl—convert 32-bit value from network to host byte order (8.1.4)

ntohs—convert 16-bit value from network to host byte order (8.1.4)

sethostent—start host-database scan (8.8.1)

setnetent—start network-database scan (8.8.2)

setprotoent—start protocol-database scan (8.8.3)

setservent—start service-database scan (8.8.4)

Process Attribute

chdir—change current directory by path (3.6.2)

chroot—change root directory (5.14)

fchdir—change current directory by file descriptor (3.6.2)

getcwd—get current directory pathname (3.4.2)

getrusage—get resource usage (5.16)

nice—change nice value (5.15)

Process Control Flow

_longjmp—jump to jump point without restoring signal mask (9.6)

_setjmp—set jump point (9.6)

longjmp—jump to jump point (9.6)

pause—wait for signal (9.2.1)

setjmp—set jump point (9.6)

siglongjmp—jump to jump point, restore signal mask if saved (9.6)

sigsetjmp—set jump point (9.6)

Process Environment

getenv—get value of environment variable (5.2)

putenv—change or add to environment (5.2)

setenv—change or add to environment (5.2)

unsetenv—remove environment variable (5.2)

Process Limit

getrlimit—get resource limits (5.16)

setrlimit—set resource limits (5.16)

ulimit—get and set process limits (5.16)

Process Management

_Exit—terminate process without cleanup (5.7)

_exit—terminate process without cleanup (5.7)

abort—generate SIGABRT (9.1.9)

atexit—register function to be called when process exits (1.3.4)

execl—execute file with argument list (5.3)

execle—execute file with argument list and environment (5.3)

execlp—execute file with argument list and PATH search (5.3)

execv—execute file with argument vector (5.3)

execve—execute file with argument vector and environment (5.3)

execvp—execute file with argument vector and PATH search (5.3)

exit—terminate process with cleanup (5.7)

fork—create new process (5.5)

system—run command (5.5)

vfork—create new process; share memory (obsolete) (5.5)

wait—wait for child process to terminate (5.8)

waitid—wait for child process to change state [X/Open] (5.8)

waitpid—wait for child process to change state (5.8)

Process Permission

getegid—get effective group ID (5.11)

geteuid—get effective user ID (5.11)

getgid—get real group ID (5.11)

getuid—get real user ID (5.11)

setegid—set effective group ID (5.12)

seteuid—set effective user ID (5.12)

setgid—set real, effective, and saved group ID (5.12)

setuid—set real, effective, and saved user ID (5.12)

umask—set and get file mode creation mask (2.5)

Process Resource

getpgid—get process-group ID (4.3.3)

getpid—get process ID (5.13)

getppid—get parent process ID (5.13)

getsid—get session ID (4.3.2)

setpgid—set or create process group (4.3.3)

setsid—create session and process group (4.3.2)

Signal

kill—generate signal for processes (9.1.9)

killpg—generate signal for process group (9.1.9)

raise—generate signal for thread (9.1.9)

sigaction—set signal action (9.1.6)

sigaltstack—set and get alternate stack context (9.3)

sighold—block signal (9.4)

sigignore—ignore signal (9.4)

siginterrupt—set or clear SA_RESTART flag (9.3)

sigismember—test for signal in signal set (9.1.5)

signal—set signal action (9.4)

sigpause—change signal mask and wait for signal (9.4)

sigpending—examine pending signals (9.3)

sigqueue—generate signal for process (9.5.4)

sigrelse—unblock signal (9.4)

sigset—signal management (9.4)

sigsuspend—change signal mask and wait for signal (9.2.3)

sigtimedwait—wait for signal (9.5.5)

sigwait—wait for signal (9.2.2)

sigwaitinfo—wait for signal (9.5.5)

Signal Mask

sigaddset—add signal to signal set (9.1.5)

sigdelset—delete signal from signal set (9.1.5)

sigemptyset—initialize empty signal set (9.1.5)

sigfillset—initialize full signal set (9.1.5)

sigprocmask—change thread's signal mask (single thread only) (9.1.5)

Socket

accept—accept new connection on socket and create new socket (8.1.2)

bind—bind name to socket (8.1.2)

connect—connect socket (8.1.2)

getpeername—get socket address of connected socket (8.9.2)

getsockname—get socket address (8.9.2)

getsockopt—get socket options (8.3)

listen—mark socket for accepting and set queue limit (8.1.2)

recv—receive data from socket (8.9.1)

recvfrom—receive message from socket (8.6.2)

recvmsg—receive message from socket (8.6.3)

send—send data to socket (8.9.1)

sendmsg—send message to socket using message structure (8.6.3)

sendto—send message to socket (8.6.2)

setsockopt—set socket options (8.3)

shutdown—shut down socket send and/or receive operations (8.9.4)

sockatmark—test presence of out-of-band mark (8.7)

socket—create endpoint for communication (8.1.2)

socketpair—create pair of sockets (8.9.3)

Socket Address

freeaddrinfo—free socket-address information (8.2.6)

gai_strerror—get error-code description (8.2.6)

getaddrinfo—get socket-address information (8.2.6)

gethostbyaddr—look up host by address (8.8.1)

gethostbyname—look up host by name (8.8.1)

getnameinfo—get name information (8.8.1)

inet_addr—convert dotted IPv4 address string to integer (8.2.3)

inet_ntoa—convert integer IPv4 address to dotted string (8.2.3)

inet_ntop—convert IPv4 or IPv6 binary addresses to string (8.9.5)

inet_pton—convert IPv4 or IPv6 string addresses to binary (8.9.5)

Terminal

cfgetispeed—get input speed from termios structure (4.5.3)

cfgetospeed—get output speed from termios structure (4.5.3)

cfsetispeed—set input speed in termios structure (4.5.3)

cfsetospeed—set output speed in termios structure (4.5.3)

ctermid—get pathname for controlling terminal (4.7)

ioctl—control character device (4.4)

isatty—test for terminal (4.7)

tcdrain—drain (wait for) terminal output (4.6)

tcflow—suspend or restart flow of terminal input or output (4.6)

tcflush—flush (throw away) terminal output, input, or both (4.6)

tcgetattr—get terminal attributes (4.5.1)

tcgetpgrp—get foreground process-group ID (4.3.4)

tcgetsid—get session ID (4.3.4)

tcsendbreak—send break to terminal (4.6)

tcsetattr—set terminal attributes (4.5.1)

tcsetpgrp—set foreground process-group ID (4.3.4)

ttyname—find pathname of terminal (4.7)

ttyname_r—find pathname of terminal (4.7)

Terminal (Pty)

grantpt—get access to slave side of pty (4.10.1)

posix_openpt—open pty (4.10.1)

ptsname—get name of slave side of pty (4.10.1)

unlockpt—unlock pty (4.10.1)

Thread

pthread_cancel—cancel thread (5.17.5)

pthread_cleanup_pop—uninstall cleanup handler (5.17.5)

pthread_cleanup_push—install cleanup handler (5.17.5)

pthread_cond_signal—signal condition (5.17.4)

pthread_cond_wait—wait for condition (5.17.4)

pthread_create—create thread (5.17.1)

pthread_join—wait for thread to terminate (5.17.2)

pthread_kill—generate signal for thread (9.1.9)

pthread_mutex_lock—lock mutex (5.17.3)

pthread_mutex_unlock—unlock mutex (5.17.3)

pthread_sigmask—change thread's signal mask (9.1.5)

pthread_testcancel—test for cancellation (5.17.5)

Time

asctime—convert broken-down time to local-time string (1.7.1)

clock—get execution time (1.7.2)

clock_getcpuclockid—get process CPU-time clock (9.7.5)

clock_getres—get clock resolution (9.7.5)

clock_gettime—get time from clock (9.7.5)

clock_nanosleep—suspend execution for nanoseconds or until signal (9.7.5)

clock_settime—set clock (9.7.5)

ctime—convert time_t to local-time string (1.7.1)

difftime—subtract two time_t values (1.7.1)

getdate—convert string to broken-down time with rules (1.7.1)

gettimeofday—get current date and time as timeval (1.7.1)

gmtime—convert time_t to UTC broken-down time (1.7.1)

localtime—convert time_t to local broken-down time (1.7.1)

mktime—convert local broken-down time to time_t (1.7.1)

nanosleep—suspend execution for nanoseconds or until signal (9.7.3)

sleep—suspend execution for seconds or until signal (9.7.2)

strftime—convert broken-down time to string with format (1.7.1)

strptime—convert string to broken-down time with format (1.7.1)

time—get current date and time as time_t (1.7.1)

times—get process and child-process execution times (1.7.2)

tzset—set time zone information (1.7.1)

usleep—suspend execution for microseconds or until signal (9.7.3)

wcsftime—convert broken-down time to wide-character string with format (1.7.1)

Timer

alarm—schedule an alarm signal (9.7.1)

getitimer—get value of interval timer (9.7.4)

setitimer—set value of interval timer (9.7.4)

timer_create—create per-process timer (9.7.6)

timer_delete—delete per-process timer (9.7.6)

timer_getoverrun—get per-process timer overrun count (9.7.6)

timer_gettime—get value of per-process timer (9.7.6)

timer_settime—set value of per-process timer (9.7.6)

User Info

getgrgid—get group-file entry (3.5.2)

getlogin—get login name (3.5.2)

getpwuid—get password-file entry (3.5.2)

References

[Abr1996] Abrahams, Paul W. and Bruce R. Larson, *UNIX for the Impatient, 2nd Ed.*, Addison-Wesley Longman, 1996.

[AUP2003] Rochkind, Marc, Advanced UNIX Programming Web Site, *www.basepath.com/aup*

[Bac1986] Bach, Maurice J., *The Design of the UNIX Operating System*, Prentice Hall, 1986.

[Bov2001] Bovet, Daniel P. and Marco Cesati, *Understanding the Linux Kernel*, O'Reilly & Associates, 2001.

[But1997] Butenhof, David R., *Programming with POSIX Threads*, Addison-Wesley Longman, 1997.

[Dre2003] Drepper, Ulrich, "POSIX Option Groups," *http://people.redhat.com/~drepper/posix-option-groups.html*

[Har2002] Harbison III, Samuel P. and Guy L. Steele Jr., *C: A Reference Manual, 5th Ed.*, Prentice Hall, 2002.

[Keg2003] Kegel, Dan, "The C10K Problem," *www.kegel.com/c10k.html*

[Ker1984] Kernighan, Brian W. and Rob Pike, *The UNIX Programming Environment*, Prentice Hall Computer Books, 1984.

[Mau2001] Mauro, Jim and Richard McDougall, *Solaris Internals*, Prentice Hall PTR, 2001.

[McK1996] McKusick, Marshall Kirk, Keith Bostic, Michael J. Karels, and John S. Quarterman, *The Design and Implementation of the 4.4BSD Operating System*, Addison-Wesley, 1996.

[Nor1997] Norton, Scott J. and Mark D. DiPasquale, *Thread Time: The Multithreaded Programming Guide*, Prentice-Hall PTR, 1997.

[RFC] IETF RFC Page, *www.ietf.org/rfc*

[RFC854] Postel, J. and J. Reynolds, "Telnet Protocol Specification," *www.ietf.org/rfc/rfc0854.txt?number=854*

[RFC1288] Zimmerman, D., "The Finger User Information Protocol," *www.ietf.org/rfc/rfc1288.txt?number=1288*

[Ste1999] Stevens, Richard P., *UNIX Network Programming, Vol. 2, 2nd Ed.*, Prentice Hall PTR, 1999.

[Ste2003] Stevens, Richard P., Bill Fenner, and Andrew M. Rudoff, *UNIX Network Programming, Vol. 1, 3rd Ed.*, Addison-Wesley, 2004.

[Str2000] Stroustrup, Bjarne, *The C++ Programming Language (Special 3rd Edition)*, Addison-Wesley, 2000.

[SUS2002] The Open Group, "The Single UNIX Specification, Version 3," *www.unix.org/version3*

Other Recommendations

Bell Labs, "The Creation of the UNIX* Operating System," *www.bell-labs.com/history/unix/*. An interesting site sponsored by Bell Labs, where UNIX originated.

Gallmeister, Bill O., *POSIX.4: Programming for the Real World*, O'Reilly & Associates, 1995. The only book on the POSIX realtime extensions, written by the vice-chair of the group that wrote the standard.

Garfinkel, Simson, Daniel Weise, and Steven Strassmann, *The UNIX-Haters Handbook*, IDG Books Worldwide, 1994. UNIX-lovers hate this book, but its criticisms are right on. The book is hilarious and deserves to be recommended for that reason alone. Out of print, but you can read it for free at *http://web.mit.edu/~simsong/www/ugh.pdf*. (If the link doesn't work, google the book title.)

Google Groups, *www.google.com/grphp*. Where you'll find newsgroups such as comp.unix.programmer.

Libes, Don and Sandy Ressler, *Life With UNIX*, Prentice Hall, 1989. UNIX history and lots of other interesting information, both trivial and important. Most of its technical and market information is dated, but there's plenty left that isn't.

Nemeth, Evi, Garth Snyder, Scott Seebass, and Trent R. Hein, *UNIX System Administration Handbook, 3rd Ed.*, Prentice Hall PTR, 2001. The best UNIX administration book. Covers Solaris, HP-UX, Red Hat Linux, and FreeBSD specifically, but most of the book applies to any UNIX system. From its title you might not think this book is a pleasure to read, but it is.

Salus, Peter H., *A Quarter Century of UNIX*, Addison-Wesley, 1994. Exceptionally complete history of UNIX.

Taylor, Christopher C., "Unix Is a Four Letter Word," *http://unix.t-a-y-l-o-r.com*. Online introduction to UNIX and vi.

Unix Heritage Society, *www.tuhs.org*. Outstanding collection of links to historical UNIX materials.

Index

A
abort signals, 7
absolute path, directories, 3
`accept`, 526
  file descriptors, 73
`access`, 185–186
  file systems, low-level access, 127–131
addresses, sockets, 523
  AF_INET6, 538–539
  AF_UNIX, 534–535
  functions, 700
  structures, 533–534
advisory locks, 81, 487–488
AF_INET6 socket addresses, 538–539
AF_UNIX socket addresses, 534–535
AIO (asynchronous I/O), 189–190
  control block, 191–192
  synchronous comparison, 196–198
`aio_cancel`, 194
`aio_error`, 193
`aio_fsync`, 194–195
`aio_read`, 192
`aio_return`, 193
`aio_suspend`, 195–196
`aio_write`, 192
`alarm`, 651–653
API standards, 39–40
applications, full-screen, 250–255
`asctime`, 59
async-signal-safe functions, 616
AT&T Bell Laboratories, 16

B
bidirectional pipes, 399–401
binary semaphores, 81
`bind`, 525
block special files, 4–5
blocks, 14
  processes and, 350–351
  threads and, 350–351
  Unified Event Manager, 351–357
BSD (Berkeley Software Distribution), 16
buffer-flushing system calls, 115–117
buffers, 4
byte order, 530–533

C
C programming language
  bindings, 20–21
  error checking, macros, 29–36
  macros, error checking, 29–36
C++ programming language
  bindings, 20–21
  exceptions, 36–38
calendar time, 56–63
  `asctime`, 59
  `ctime`, 59
  `difftime`, 60
  `get date`, 60
  `gettimeofday`, 58
  `gmtime`, 58
  `localtime`, 58
  `mktime`, 59
  `strftime`, 59
  `struct timeval`, 58
  `struct tm`, 58
  `time`, 58
  `wcsftime`, 59
canceling threads, 344–348
canonical I/O, 239–242

character mapping, 236–237
character special files, 4
characters
  parity, 234–235
  size, 234–235
`chdir`, 166–168
checking for errors, 24–29
child processes, 6
  SIGCHLD signal, 313–314
  `wait`, 309–310
  `waitid`, 310–313
  `waitpid`, 304
`chmod`, i-nodes and, 182
`chown`, 182–184
`chroot`, 319–320
clients
  SMI, 419–420
  sockets, 527–530
`clock`, 64
`clock_getres`, 661
`clock_gettime`, 661
`clock_settime`, 661
clocks/timers
  basic interval-timer system calls, 657–660
  signals
  `alarm`, 651–653
  realtime clocks, 660–663
  `sleep`, 653–656
`close`, 97–98
`closedir`, 160–162
command line, arguments, 293
common header file, 55–56
condition variables, threads, 338–344
configuration functions, 693
`confstr`, 53
`connect`, 526–527, 576
connect command, file descriptors, 73
connected sockets, system calls, 524–527
connectionless
  `recvfrom`, 568–574
connectionless sockets
  `connect`, 576
  datagrams, 567–568
  `sendto`, 568–574
control characters, terminals, 238–239
controlling terminal, 8
`creat`, 77–80
`ctermid`, 248
`ctime`, 59
Curses functions, 254–255

D

daemons, 8
data transfer, DMA and, 4
databases, file locking, 488
datagrams, 567–568
date/time
  calendar time, 56–63
  execution time, 63–67
    `clock`, 64
    `struct timespec`, 64
    `struct tms`, 64
    `times`, 64
delays, terminals, 237
device drivers, 4
device number, i-node, 4
`difftime`, 60
directories, 2–4
  `chdir`, 166–168
  creating, 168–170
  `fchdir`, 166–168
  `ftw` and, 174–180
  `getcwd` and, 170–174
  links
    multiple, 3–4
    removing, 86–87
  pathnames, `getcwd`, 145–147
  paths, 2–3
    absolute, 3
    relative, 3
  reading, 158–166
  removing, 168–170
  renaming, 139–141
  rewinding, 161–162
  root, `chroot`, 319–320
directory I/O functions, 693
directory management functions, 693
disk special files, I/O, 123–127
disks, physical, 124
DMA (direct memory access), 4
DNS (domain name system), 539–540
drivers, device drivers, 4
`dup`, 371–375
`dup2`, 371–375
duplicates, 371–375

E

`e_cai macro`, 35
`EC_CLEANUP` macro, 36
`EC_CLEANUP_BGN` macro, 35
`EC_CLEANUP_END` macro, 35
`ec_cmp` macro, 35, 36

`ec_eof` macro, 35
`EC_FAIL` macro, 36
`ec_false` macro, 35
`EC_FLUSH` macro, 36
echo on/off, 239
`ec_neg1` macro, 35
`ec_null` macro, 35
`ec_print` macro, 36
`ec_push` macro, 36
`ec_reinit` macro, 36
`ec_rv` macro, 35
`ec_warn` macro, 36
effective group-ID, 10
effective user-ID, 9
`endhostent`, 578
`endnetent`, 586
`endprotoent`, 588
`endservent`, 590
environment, 277–283
`errno`, 24–29
error checking, 24–29
  C, macros, 29–36
error handling, error checking, 24–29
event flags, `poll` and, 219
events, `ptys`, 268
example code, 67–68
exceptions, C++, 36–38
`exec` system calls, 284–292
  `execl`, 285
  `execle`, 289
  `execlp`, 288
  `execv`, 288
  `execve`, 289
  `execvp`, 288
  signals and, 623–624
execute permissions, 10
  symbols, 75–76
execution time, 63
  `clock`, 64
  `struct timespec`, 64
  `struct tms`, 64
  `times`, 64
`exit` system calls, 301–304

F

`fchdir`, 166–168
`fchmod`, i-nodes and, 182
`fchown`, 182–184
`fcntl`, 187–189
  file locking and, 485–486
`fdatasync`, 116–117
FIFOs, 15. *See also* named pipes
  advantages/disadvantages, 414
  creating, 407–408
  examples, 408–413
  introduction, 406–407
  performance comparisons, 515–517
  pipes and, 363–371
  SMI implementation, 420–428
  sockets, 520
file attribute functions, 693–694
file descriptors, 12
  duplicates, 371–375
  metadata and, `fstat`, 147–154
  numbering, 72
  opening, 74–75
  sharing, 74–75
  standard, 73
  uses, 73–74
file I/O functions, 694
file I/O functions (asynchronous), 694
file locks, 13, 15, 477–481
  databases, 488
  `fcntl`, 485–486
  semaphores as, 481–483
file management functions, 695
file objects
  locks, 13, 15
  messages, 13, 15
  semaphores, 13
  shared memory, 13, 15–16
  sockets, 13
file offset, 2, 90–92
file permission bits, symbols, 75–76
file size limit, 13
file system functions, 695
file systems, 124
  access, low-level, 127–131
  mounting, 135–137
  unmounting, 135–137
file tree
  walking down (`ftw`), 174–180
  walking up (`getcwd`), 170–174
file-descriptor set functions, 695
files
  as locks, 80–85
  access times, 184–185
  buffers, 4
  bytes, inserting in middle, 2
  creating, 77–80, 186–187
  directories, 2–4
  header files, common header file, 55–56

files (*cont.*)
  i-numbers, 2
  metadata
    accessing, 147–158
    displaying, 147–158
  mknod facility, 186–187
  opening, 76–77
  owners, changing, 182–184
  regular, 1–2
  renaming, 139–141
  semaphores, 2, 458
  shared file offsets, 13
  special, 4–5
    block, 4
    character, 4
  symbolic links, 2–4
  temporary, creating, 88–90
  text files, 5
flags, open flags, 85
  synchronized I/O and, 117–119
`fork`, 284, 296–300
  signals and, 623–624
`fpathconf`, 52
`fstat`, 147–154
`fstatvfs`, 131–135
`ftok`, 429–431
`ftruncate`, 119–120
`ftw`, implementation, 174–180
full-screen applications, 250–255
function calls, SSI, 550–551
functions, 23
  `aio_cancel`, 194
  `aio_error`, 193
  `aio_fsync`, 194–195
  `aio_read`, 192
  `aio_return`, 193
  `aio_suspend`, 195–196
  `aio_write`, 192
  alphabetic listing, 685–693
  async-signal-safe, 616
  calendar time
    `asctime`, 59
    `ctime`, 59
    `difftime`, 60
    `get date`, 60
    `gettimeofday`, 58
    `gmtime`, 58
    `localtime`, 58
    `mktime`, 59
    `strftime`, 59
    `struct timeval`, 58
    `struct tm`, 58
    `time`, 58
    `wcsftime`, 59
  categorical listing, 693–702
  `clock_getres`, 661
  `clock_gettime`, 661
  `clock_settime`, 661
  configuration functions, 693
  Curses functions, 254–255
  directory I/O function, 693
  directory management, 693
  `endhostent`, 578
  `endnetent`, 586
  `endprotoent`, 587
  `endservent`, 590
  execution time
    `clock`, 64
    `struct timespec`, 64
    `struct tms`, 64
    `times`, 64
  file attributes, 693–694
  file I/O, 694
  file I/O (asynchronous), 694
  file management, 695
  file system, 695
  file-descriptor set, 695
  `gai_strerror`, 541
  `getenv`, 279
  `gethostbyaddr`, 581–582
  `gethostbyname`, 580–581
  `gethostent`, 578
  `gethostid`, 583–584
  `getnameinfo`, 582–583
  `getnetent`, 586
  `getprotobyname`, 589
  `getprotobynumber`, 589
  `getprotoent`, 587
  `getservbyname`, 590
  `getservbyport`, 590
  `getservent`, 590
  `if_freenameindex`, 591
  `if_indextoname`, 593
  `if_nameindex`, 591
  `if_nametoindex`, 592–593
  `inet_addr`, 536–537
  `inet_ntoa`, 536–537
  `inet_ntop`, 596–597

`inet_pton`, 596–597
IPC-POSIX message queue, 695
IPC-POSIX semaphore, 695
IPC-POSIX shared memory, 696
IPC-System V message queue, 696
IPC-System V semaphore, 696
IPC-System V shared memory, 696
library functions, calling, 22–23
`lio_listio`, 199–200
`lock`, 81
`lockpath`, 82
`longjump`, 648–651
`mkstemp`, 88–90
network databases, 696–697
    host functions, 578–585
    network functions, 585–587
process attributes, 697
process control flow, 697
process environment, 697
process limit, 697
process management, 698
process permission, 698
process resources, 698
`putenv`, 280
`recv`, 593–594
`send`, 593–594
`setenv`, 280–281
`sethostent`, 578
`setjump`, 648–651
`setnetent`, 586
`setprotoent`, 586
`setservent`, 589
`sigaddset`, 608
`sigdelset`, 608
`sigemptyset`, 608
`sigfillset`, 608
`sigismemberset`, 609
signal masks, 698–699
signals, 698–699
sockets, 699, 700
`tcgetattr`, 233
`tcsetattr`, 233
terminal, 700
terminal (Pty), 700
thread, 701
`time`, 701–702
timers, 702
`tmpnam`, 88–90
`uname`, 584
`unlock`, 81
`unsetenv`, 280–281
user buffering and, 100–105
user info, 702

G

`gai_strerror`, 541
gather write, 110
generating signals, synthetically, 621–623
`get date`, 60
`getaddrinfo`, 540–543
`getcwd`, 145–147
    implementation, 170–174
`getegid`, 315–316
`getenv`, 279
`geteuid`, 315–316
`getgid`, 315–316
`getgrgid`, 154–155
`gethostbyaddr`, 581–582
`gethostbyname`, 580–581
`gethostent`, 578
`gethostid`, 583–584
`gethostname`, 543–544
`getlogin`, 154–155
`getnameinfo`, 582–583
`getnetent`, 586
`getpeername`, 594
`getpgid`, 228–229
`getpid`, 319
`getppid`, 319
`getprotobyname`, 589
`getprotobynumber`, 589
`getprotoent`, 587
`getpwuid`, 154–155
`getrlimit`, 323
`getrusage`, 328
`getservbyname`, 590
`getservbyport`, 590
`getservent`, 590
`getsid`, 227
`getsockname`, 594
`gettimeofday`, 58
`gettimer`, 658
`getuid`, 315–316
global jumps, 648–651
`gmtime`, 58
group files, 10
group IDs
    setting, 317–318
    system calls, 315–316

group-ID, 9
groups, users, 9
GUIs (Graphical User Interfaces), 250

H
hard links
  creating, 138–139
  introduction, 137–138
  renaming and, 139–141
header files
  common header file, 55–56
  system calls, 20
host functions, network databases, 578–585
HTTP (Hypertext Transfer Protocol), SSI and, 551–552

I
identifiers, System V IPC, 429–431
IEEE (Institute of Electrical and Electronics Engineers), 17
`if_freenameindex`, 591
`if_indextoname`, 593
`if_nameindex`, 591
`if_nametoindex`, 592–593
implementation
  shells
    `exec`, 292–296
    `fork`, 300–301
    `waitpid`, 314–315
  SMI, sockets and, 563–566
  SSI, 559–563
`inet_ntoa`, 536–537
`inet_addr`, 536–537
`inet_ntop`, 596–597
`inet_pton`, 596–597
i-nodes
  changing, 181
  `chmod` and, 182
  `chown` and, 182–184
  device number, 4
  `fchmod` and, 182
  `fchown` and, 182–184
  `lchown` and, 182–184
  System V IPC, 429
instructions, 5
interfaces, semaphores, 463–469
interprocess communication, introduction, 361–362
interrupted system calls, signals and, 607–608
i-numbers, files, 2
  links, 3
I/O (input/output)
  asynchronous, 189–200
  buffered *versus* unbuffered, 104
  disk special files, 123–127
  overview, 71–72
  pipes and, 363–371
  STREAMS I/O, 255
  synchronized, *versus* synchronous, 114
  terminal I/O (*See* terminal I/O)
  user buffering
    functions, 100–105
    kernel buffering and, 98–100
`ioctl`, 232–233
IPC (interprocess communication), 405. *See also* System V IPC
IPC-POSIX
  message queue functions, 695
  semaphore functions, 695
  shared memory functions, 696
IPC-System V
  message queue functions, 696
  semaphore functions, 696
  shared memory functions, 696
`isatty`, 249

J
Java, 21
job control, 8, 225
jobs, 8
jumps, global, 648–651

K
kernel buffered I/O, user buffering and, 98–100
keys, System V IPC, 429–431

L
`lchown`, 182–184
lexical analyzer, shells, 377–381
library functions, calling, 22–23
limiting processes, 322–329
links
  directories
    multiple, 3–4
    removing, 86–87
  hard links, 137–138
    creating, 138–139
    renaming and, 139–141
  i-numbers and, 3
  symbolic links, 2–4, 3–4, 137–138
    creating, 141–144

`links`, 138–139
`lio_listio`, 199–200
`listen`, 525
`localtime`, 58
`lock`, 81
`lockf`, 483–485
locking files, 477–481
  advisory locks, 487–488
  databases, 488
  `fcntl`, 485–486
  mandatory locks, 487–488
  semaphores as file locks, 481–483
`lockpath`, 82
locks
  advisory, 81
  files as, 80–85
  mandatory, 81
`longjump`, 648–651
low-level file system access, 127–131
`lseek`, 105–108
`lstat`, 147–154

M

macros
  error checking, C, 29–36
  POSIX IPC, 444–445
mandatory locks, 81, 487–488
mapping
  character, 236–237
  newlines, 236–237
memory, shared, 488–489
  POSIX, 504
    system calls, 505–507
  POSIX implementation of SMI, 508–515
  semaphores and, 492–496
  System V implementation of SMI, 496–504
  System V system calls, 489–492
messages, 13, 15
  semaphores, 458
  System V queue, limits, 437–438
metadata
  accessing, 147–158
  displaying, 147–158
`mkdir`, 168–170
`mkfifo`, 407
`mknod`, 186–187
`mkstemp`, 88–90
`mktime`, 59
`mount`, 135–136
mounting file systems, 135–137
`mq_close`, 447
`mq_getattr`, 450
`mq_notify`, 450
`mq_open`, 446
`mq_receive`, 448
`mq_send`, 448
`mq_setattr`, 451
`mq_timedreceive`, 449
`mq_timedsend`, 449
`mq_unlink`, 447
`msgctl`, 435
`msgget`, 434
`msgrev`, 436
`msgsnd`, 436
mutexes, 81, 333–338
  process-shared, 476

N

named pipes, 13, 15. *See also* FIFOs
named POSIX semaphores, 470–474
name/i-node pairs, 2–4
`nanosleep`, 657
network database functions, 696–697
  host functions, 578–585
  network functions, 585–587
  protocol functions, 587–589
  service functions, 589–591
network interface functions
  `if_freenameindex`, 591
  `if_nameindex`, 591
networking. *See also* sockets
  byte order, 530–533
newlines, mapping, 236–237
`nice`, 320–321
  values, 13
nonblocking input, terminal I/O and, 208–213

O

`O_APPEND` flag, file offsets and, 90–92
object files, 5
`O_DSYNC` flag, 117
`open`, 76–77
open command, file descriptors, 73
open files, controlling, 187
open flags, 85
  synchronization and, 117–119
`opendir`, 160–162
options, POSIX, 48–50
`O_RSYNC` flag, 117
OS, checking for, 53–54

O_SYNC flag, 117
out-of-band data, sockets, 577
ownership, System V IPC, 431–432

P
parent processes, 6
parent-process-ID, 7
partitions, 124
passwd, 12
pathconf, 52
pathnames
  directories, getcwd, 145–147
  length, 144–145
  System V IPC, 429
paths, 2–3
  absolute, 3
  relative, 3
Perl, 21
permissions
  bits, 10–11
  changing, 12
  effective group-ID, 10
  effective user-ID, 9
  execute, 10
  file permission bits, symbols, 75–76
  group-ID, 9
  passwd, 12
  read, 10
  real group-ID, 9
  real user-ID, 9
  restricting, 78
  supplementary group-ID, 10
  System V IPC, 431–432
  user-ID, 9
  write, 10
physical disks, 124
pipelines, shells, 376
pipes, 13, 14–15
  bidirectional, 399–401
  examples, 366–371
  FIFOs and, 363–371
  file descriptors, 73
  I/O behavior, 363–371
  named pipes, 13
  pipe, 362–363
  reading from, 365–366
  unidirectional, 390–399
  writing to, 364–365
poll, 218–221
POSIX
  message queues
    advantages/disadvantages, 456–457
    SMI implementation, 451–456
    system calls, 445–451
  options, 48–50
  performance comparisons, 515–517
  POSIX 1990, 18
  POSIX IPC
    history, 443
    macros, 444–445
    naming conventions, 443–444
    utilities, 445
  semaphores, 469
    named, 470–474
    process-shared mutexes, 476
    read-write locks, 476
    System V comparison, 476
    unnamed, 474–476
  shared memory, 504
    disadvantages/advantages, 515
    SMI implementation, 508–515
    system calls, 505–507
  standards, 39–40
pread, 108–110
priorities
  getting, 320–322
  nice value, 13
  processes, 13
  setting, 320–322
process control flow functions, 697
process environment functions, 697
process groups, 7, 224. *See also* jobs
  leaders, 8
  process-group leader, 224
  sessions, 7
  system calls, 228–229
process IDs, system calls, 319
process limit functions, 697
process management functions, 698
process permission functions, 698
process resources functions, 698
process tracing, 13, 14
processes, 5
  attributes, 667–671
    functions, 697
  blocking and, 350–351
  child
    SIGCHLD signal, 313–314

wait, 309–310
waitid, 310–313
waitpid, 304
file descriptor, 12
file size limit, 13
fork, 296–297
limits, 322–329
parent-process-ID, 7
priorities, 13
getting, 320–322
setting, 320–322
signals, 13
processes (*cont.*)
terminating, exit system calls, 301–304
threads comparison, 348–349
process-ID, 7
process-shared mutexes, POSIX semaphores and, 476
programs, 5–6
protocol functions, network databases, 587–589
pseudo terminals. *See* ptys (pseudo terminals)
pthread_cancel, 344–345
pthread_cleanup_pop, 347–348
pthread_cleanup_push, 347–348
pthread_cond_signal, 341–344
pthread_cond_wait, 341–344
pthread_create, 331, 623–624
pthread_join, 332–333
pthread_mutex_lock, 334–338
pthread_mutex_unlock, 334–338
pthread_testcancel, 345–347
ptys (pseudo terminals), 256–257
library, 258–267
record/playback, 267–276
punctual I/O, 239–242
putenv, 280
pwrite, 108–110

R

raw terminal I/O, 242–245
read, 96–97
pread and, 108
synchronous I/O and, 114
read permissions, 10
symbols, 75–76
readdir, 161–162
reading directories, 158–166
readv, 110–113
read-write locks, POSIX semaphores and, 476
real group-ID, 9
real user-ID, 9
realtime clocks, signals, 660–663
record/playback, ptys and, 267–276
recv, 593–594
recvfrom, 568–574
recvmsg, 574–576
regular files, 1–2
relative paths, directories, 3
rename, 139–141
renaming
directories, 139–141
files, 139–141
restrictions, permissions, 78
rewinddir, 161–162
Ritchie, Dennis, 16
rmdir, 168–170
root directory, chroot, 319–320
RTS (Realtime Signals Extension), 637–638
queued signals, 642
RTS signals, 641–642
signal handlers, 638–641

S

scatter read, 110
seekdir, 161–162
segments, 5
select, 213–218, 528
semaphores, 13, 15, 458–459
binary semaphores, 81
as file locks, 481–483
files, 2
interface, 463–469
POSIX, 469
named, 470–474
System V comparison, 476
unnamed, 474–476
shared memory and, 492–496
System V
POSIX comparison, 476
system calls, 460–463
semctl, 461
semget, 460
send, 593–594
sendmsg, 574–576
sendto, 568–574
servers
SMI, 419–420
Web, SSI, 554–558
service functions, network databases, 589–591
sessions, 7
controlling processing, 224

sessions (*cont.*)
controlling terminal, 224
leaders, 8–9
process groups, 224
process-group leaders, 224
session leader, 224
system calls, 227–228, 230–232
`setegid`, 318
`setenv`, 280–281
`setgid`, 318
set-group-ID, 12
`sethostent`, 578
`setjump`, 648–651
`setnetent`, 586
`setpgid`, 228–229
`setprotoent`, 587
`setrlimit`, 323
`setservent`, 589
`setsid`, 227
`setsockopt`, 544–545
`settimer`, 658
`setuid`, 318
set-user-ID, 12
shared file offsets, 13
shared memory, 13, 15–16, 488–489
POSIX, 504
disadvantages/advantages, 515
SMI implementation, 508–515
system calls, 505–507
semaphores and, 492–496
System V
SMI implementation, 496–504
system calls, 489–492
sharing, file descriptors, 74–75
shells
implementation
`exec`, 292–296
`fork`, 300–301
`waitpid`, 314–315
lexical analyzer, 377–381
pipelines, 376
simple commands, 376
tokens, 376–377
`shmat`, 490
`shmctl`, 489
`shmdt`, 490
`shmget`, 489
`shm_open`, 505
`shm_unlink`, 505
`shutdown`, 595
`sigaction`, 611–615
`sigaddset`, 608
`sigaltstack`, 634
`SIGCHLD`, 313–314
`sigdelset`, 608
`sigemptyset`, 608
`sigfillset`, 608
`sighold`, 636
`sigignore`, 636
`sigismember`, 609
`siglongjmp`, 650–651
`signal`, 635–636
signal handlers, 615–618
minimal defensive signal handling, 618–621
signal masks
functions, 698–699
`sigaddset`, 608
`sigdelset`, 608
`sigemptyset`, 608
`sigfillset`, 608
`sigismember`, 609
signals, 6–7, 13
abort signals, 7
advanced interval-timer system calls, 663–665
async-signal-safe functions, 616
clocks/timers, 651–665
basic interval-timer system calls, 657–660
realtime clocks, 660–663
default actions, 605
definition, 601
detected errors, 604–605
functions, 698–699
generating, synthetically, 621–623
job control, 605
lifecycle, 603–604
miscellaneous, 605
overview, 601–603
queued signals (RTS), 642
RTS (Realtime Signals Extension), 637–642
SIGCHLD, 313–314
`sigevent`, 647–648
system calls
advanced interval-timers, 663–665
alarm, 651–653
basic interval-timers, 657–660
deprecated calls, 635–637
`exec`, 623–624
`fork`, 623–624

interrupted, 607–608
nanosleep, 657
pause, 624
`pthread_create`, 623–624
`sigaction`, 611–615
`sigaltstack`, 634
`siglengjmp`, 650–651
`sigpending`, 634
`sigqueue`, 643
`sigsetjmp`, 650–651
`sigsuspend`, 628–634
`sigtimedwait`, 643–646
`sigwait`, 624–628
`sigwaitinfo`, 643–646
`sleep`, 653–656
`usleep`, 657
timer, 605
user/application generated, 605
`sigpause`, 636
`sigpending`, 634
`sigqueue`, 643
`sigrelse`, 636
`sigset`, 636
`sigsetjmp`, 650–651
`sigsuspend`, 628–634
`sigtimedwait`, 643–646
`sigwait`, 624–628
`sigwaitinfo`, 643–646
simple commands, shells, 376
`sleep`, 653–657
SMI (Simple Messaging Interface), 405
client example, 420
functions, 415–418
implementation
FIFO and, 420–428
sockets and, 563–566
System V shared memory, 496–504
introduction, 414
POSIX shared memory implementation, 508–515
POSIX message-queue implementation, 451–456
server example, 419–420
System V Message-Queue implementation, 438–442
types, 415–418
`socket`, 524–525
`socketpair`, 595
sockets, 13, 519
addresses, 523
`AF_INET6`, 538–539
`AF_UNIX`, 534–535
functions, 700
structures, 533–534
clients, multiple, 527–530
connected, system calls, 524–527
connectionless, datagrams, 567–568
FIFOs, 520
file descriptors, 73
functions, 699–700
options, 544–549
out-of-band data, 577
overview, 520–524
SMI implementation, 563–566
special files, 4–5
block special files, 4–5
character special files, 4
disk special files, I/O on, 123–127
SSI (Simple Socket Interface)
function calls, 550–551
HTTP and, 551–552
implementation, 559–563
introduction, 549
Web browser, 552–554
Web server, 554–558
`ssi_close`, 550
`ssi_close_fd`, 551
`ssi_get_server_fd`, 551
`ssi_open`, 550
`ssi_wait_server`, 550
standard file descriptors, 73
standard interfaces, library function calls and, 22
`stat`, 147–154
`statvfs`, 131–135
STREAMS I/O, 255
`strftime`, 59
`struct dirent`, 161–162
`struct iovecs`, 111
`struct timespec`, 64
`struct timeval`, 58
`struct tm`, 58
`struct tms`, 64
super user, 11
supplementary group-ID, 10
symbolic links, 2–4, 3–4, 137–138
creating, 141–144
metadata and, 147–154
`symlink`, 141–144

`sync`, 115–117
synchronization, threads, 333–338
synchronized I/O
  definition, 189–190
  open flags and, 117–119
  versus synchronous, 114
synchronized writes, 94
synchronous I/O
  asynchronous I/O comparison, 196–198
  defined, 189–190
  versus synchronized, 114
synthetically generated signals, 621–623
`sysconf`, 51–52
system calls, 19, 661. *See also* functions
  `accept`, 526
  access, 185–186
  `bind`, 525
  buffer-flushing, 115–117
  C bindings, 20–21
  C++ bindings, 20–21
  `chdir`, 166–168
  `chmod`, 182
  `chown`, 182–184
  `chroot`, 319–320
  `close`, 97–98
  `closedir`, 160–162
  `confstr`, 53
  `connect`, 526–527, 576
  connect sockets and, 524–527
  `creat`, 77–80
  `ctermid`, 248
  `dup`, 371–375
  `dup2`, 371–375
  `exec`, 284–292
    `execl`, 285
    `execle`, 289
    `execlp`, 288
    `execv`, 288
    `execve`, 289
    `execvp`, 288
  `exit`, 301–304
  `fchdir`, 166–168
  `fchmod`, 182
  `fchown`, 182–184
  `fcntl`, 187–189
  `fdatasync`, 116–117
  `fork`, 284, 296–300
  `fpathconf`, 52
  `fstat`, 147–154
  `fstatvfs`, 131–135
  `ftok`, 429–431
  `ftruncate`, 119–120
  `getaddrinfo`, 540–543
  `getcwd`, 145–147, 170–174
  `getgrgid`, 154–155
  `gethostname`, 543–544
  `getlogin`, 154–155
  `getpeername`, 594
  `getpgid`, 228–229
  `getpwuid`, 154–155
  `getrlimit`, 323
  `getrusage`, 328
  `getsid`, 227
  `getsockname`, 594
  `getsockopt`, 545–549
  `gettimer`, 658
  group IDs, 315–316
  interrupted, signals and, 607–608
  `ioctl`, 232–233
  `isatty`, 249
  `lchown`, 182–184
  links, 138–139
  `listen`, 525
  `lockf`, 483–485
  `lseek`, 105–108
  `lstat`, 147–154
  `mkdir`, 168–170
  `mkfifo`, 407
  `mknod`, 186–187
  `mount`, 135–136
  `mq_close`, 447
  `mq_getattr`, 450
  `mq_notify`, 450
  `mq_open`, 446
  `mq_receive`, 448
  `mq_send`, 448
  `mq_setattr`, 451
  `mq_timedreceive`, 449
  `mq_timedsend`, 449
  `mq_unlink`, 447
  `msgctl`, 435
  `msgget`, 434
  `msgrcv`, 436
  `msgsnd`, 436
  `nanosleep`, 657
  `nice`, 320–321
  `open`, 76–77
  `opendir`, 160–162

pathconf, 52
pipe, 362–363
poll, 218–221
POSIX message queues, 445–451
POSIX shared memory, 505–507
pread, 108–110
priority setting/getting, 320–321
process groups, 228–229
process IDs, 319
pthread_cancel, 344–345
pthread_cleanup_pop, 347–348
pthread_cleanup_push, 347–348
pthread_cond_signal, 341–344
pthread_cond_wait, 341–344
pthread_create, 331
pthread_join, 332–333
pthread_mutex_lock, 334–338
pthread_mutex_unlock, 334–338
pthread_testcancel, 345–347
pwrite, 108–110
read, 96–97
readdir, 161–162
readv, 110–113
recvfrom, 568–574
recvmsg, 574–576
rename, 139–141
rewinddir, 161–162
rmdir, 168–170
seekdir, 161–162
select, 213–218, 528
semctl, 461
semget, 460
sendmsg, 574–576
sendto, 568–574
sessions, 227–228, 230–232
setegid, 318
seteuid, 318
setgid, 318
setpgid, 228–229
setrlimit, 323
setsid, 227
setsockopt, 544–545
settimer, 658
setuid, 318
shmat, 490
shmctl, 489
shmdt, 490
shmget, 489
shm_open, 505
shm_unlink, 505
shutdown, 595
sigaction, 611–615
sigaltstack, 634
sighold, 636
sigignore, 636
siglongjmp, 650–651
signal, 635–636
sigpause, 636
sigpending, 634
sigqueue, 643
sigrelse, 636
sigset, 636
sigsetjmp, 650–651
sigsuspend, 628–634
sigtimedwait, 643–646
sigwait, 624–628
sigwaitinfo, 643–646
sleep, 653–656
socket, 524–525
socketpair, 595
stat, 147–154
statvfs, 131–135
symlink, 141–144
sync, 115–117
sysconf, 51–52
System V message queues, 434–437
System V semaphores, 460–463
System V shared memory, 489–492
tcdrain, 245–247
tcflow, 246
tcflush, 246
tcgetpgrp, 229–230
tcgetsid, 229–230
tcsendbreak, 247
tcsetpgrp, 229–230
telldir, 161–162
terminal control, 229–230, 245–247
terminal identification, 248–250
truncate, 119–120
ttyname, 248
ttyname_r, 248
umask, 86
umount, 135–136
union semun, 461
unlink, 86–87
user IDs, 315–316
usleep, 657
utime, 184–185

system calls (*cont.*)
`vfork`, 299
`wait`, 304–313
`waitid`, 304–313
`waitpid`, 304–313
`write`, 92–96
`writev`, 110–113
system data, 5, 6
System V
IPC, 428 (*See also* IPC [interprocess communication])
identifiers, 429–431
objects, 428–429
ownerships, 431–432
permissions, 431–432
utilities, 432–433
message queues
disadvantages/advantages, 442
limits, 437–438
SMI implementation, 438–442
system calls, 434–437
performance comparisons, 515–517
semaphores
POSIX comparison, 476
system calls, 460–463
shared memory
advantanges/disadvantages, 504
SMI implementation, 496–504
system calls, 489–492

T
tabs, terminals, 237
`tcflow`, 246
`tcflush`, 246
`tcgetattr`, 233
`tcgetpgrp`, 229–230
`tcgetsid`, 229–230
`tcsendbreak`, 247
`tcsetattr`, 233
`tcsetpgrp`, 229–230
`telldir`, 161–162
temporary files, creating, 88–90
terminal functions, 700
terminal I/O
canonical, 239–242
introduction, 203
`poll`, 218–221
punctual, 239–242
raw terminal I/O, 242–245
reading from terminal
`gtln`, 206
nonblocking input, 208–213
`select`, 213–218
testing input, 221–224
terminal (Pty) functions, 700
terminals
attributes
`tcgetattr`, 233
`tcsetattr`, 233
breaks, 247
character mapping, 236–237
characters, 234–235
control characters, 238–239
controlling terminal, 8
delays, 237
echo on/off, 239
flow control, 237
flushing, 246
I/O flow control, 246
pathname, 248
pseudo terminals, 256–257
reading from, terminal I/O and, 204–224
speed, 235–236
system calls, 229–230
tabs, 237
testing for, 249
termination
processes, 301–304
threads, waiting for, 332–333
termios structure, 233–234
testing, input, terminal I/O, 221–224
text, object files, 5
text files, 5
Thompson, Ken, 16
thread functions, 701
threads, 6
blocking and, 350–351
canceling, 344–348
creating, 329–331
library functions and, 22
processes comparison, 348–349
synchronization, 333–338
termination, waiting for, 332–333
variables, condition variables, 338–344
time. *See* date/time
`time`, 58
time functions, 701–702
timers, functions, 702

`times`, 64
`tmpnam`, 88–90
tokens, shells, 376–377
transferring data. *See* data transfer
`truncate`, 119–120
`ttyname`, 248
`ttyname_r`, 248

U

`umask`, 86
`umount`, 135–136
`uname`, 584
unidirectional pipes, 390–399
Unified Event Manager, blocking and, 351–357
`union semun`, 461
UNIX, history, 16–19
`unlink`, 86–87
`unlock`, 81
unmounting file systems, 135–137
`unsetenv`, 280–281
updating environment, 279
user buffered I/O
  functions, 100–105
  kernel buffering and, 98–100
user data, 5
user IDs
  setting, 317–318
  system calls, 315–316
user info functions, 702
user-ID, 9
users
  groups, 9
  super user, 11
`usleep`, 657
utilities
  POSIX IPC, 445
  System V IPC, 432–433
`utime`, 184–185

V

variables, condition variables, threads, 338–344
versions of UNIX, 16–19
`vfork`, 299
vi text editor, 250
volumes, 124

W

`wait`, 304–313
`waitid`, 304–313
`waitpid`, 304–315
`wcsftime`, 59
Web browsers, minibr (SSI), 552–554
Web servers, SSI, 554–558
`write`, 92–96
  `pread` and, 108
  `pwrite` and, 108
  synchronized I/O and, 114
  synchronized writes, 94
  `writev` comparison, 113
write permissions, 10
  symbols, 75–76
`writev`, 110–113
writing, to pipes, 364–365

This page intentionally left blank

UNIX Programming

The classic guide to UNIX® programming—completely updated!

UNIX application programming requires a mastery of system-level services. Making sense of the many functions—more than 1,100 functions in the current UNIX specification—is a daunting task, so for years programmers have turned to *Advanced UNIX Programming* for its clear, expert advice on how to reliably use the key functions.

An enormous number of changes have taken place in the UNIX environment since the landmark first edition. In *Advanced UNIX Programming, Second Edition,* UNIX pioneer Marc J. Rochkind brings the book fully up to date with all-new, comprehensive coverage including:

- ▲ POSIX
- ▲ Solaris™
- ▲ Linux®
- ▲ FreeBSD
- ▲ Darwin, the Mac™ OS X kernel
- ▲ And more than 200 new system calls

Rochkind's fully updated classic explains all the UNIX system calls you're likely to need, all in a single volume!

- ▲ Interprocess communication, networking (sockets), pseudo terminals, asynchronous I/O, advanced signals, realtime, and threads
- ▲ Covers the system calls you'll actually use—no need to plow through hundreds of improperly implemented, obsolete, and otherwise unnecessary system calls!
- ▲ Thousands of lines of example code include a Web browser and server, a keystroke recorder/player, and a shell complete with pipelines, redirection, and background processes
- ▲ Emphasis on the practical—ensuring portability, avoiding pitfalls, and much more!

Since 1985, the one book to have for mastering UNIX application programming has been Rochkind's *Advanced UNIX Programming.* Now completely updated, the second edition remains the choice for up-to-the-minute, in-depth coverage of the essential system-level services of the UNIX family of operating systems.

About the Author

MARC J. ROCHKIND was fortunate enough to have worked at Bell Laboratories in the 1970s, when UNIX was still in its infancy. It was there that Rochkind made several key contributions to UNIX, notably the Source Code Control System. He wrote the first edition of *Advanced UNIX Programming* in 1984. This complete revision benefits from his years of post-Bell application systems development experience.

Cover Design by Joyce C. Weston

Addison-Wesley
Pearson Education
informit.com/series/professionalcomputing

ISBN 978-0-13-141154-8
90000
9 780131 411548