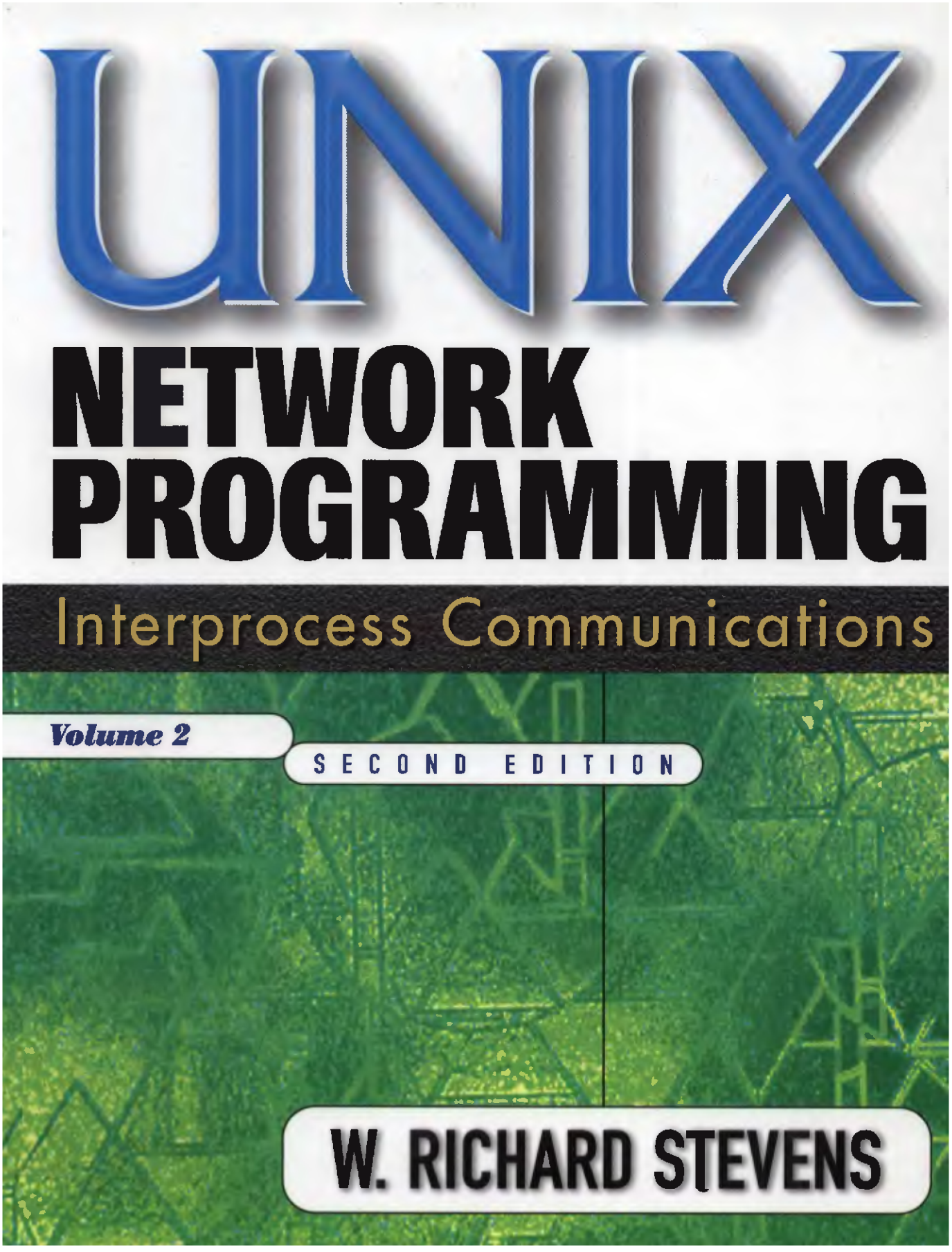
UNIX
NETWORK PROGRAMMING
Interprocess Communications
Volume 2
SECOND EDITION
W. RICHARD STEVENS

Communications
UNIX

W. RICHARD STEVENS

UNIX NETWORK PROGRAMMING

Interprocess Communications

The only guide to UNIX® interprocess communications you'll ever need!

Well-implemented interprocess communications (IPC) are key to the performance of virtually every non-trivial UNIX program. In **UNIX Network Programming, Volume 2, Second Edition,** legendary UNIX expert W. Richard Stevens presents a comprehensive guide to every form of IPC, including message passing, synchronization, shared memory, and Remote Procedure Calls (RPC).

Stevens begins with a basic introduction to IPC and the problems it is intended to solve. Step-by-step you'll learn how to maximize both System V IPC and the new Posix standards, which offer dramatic improvements in convenience and performance. You'll find extensive coverage of Pthreads, with many examples reflecting multiple threads instead of multiple processes. Along the way, you'll master every current IPC technique and technology, including:

- Pipes and FIFOs
- Posix and System V Message Queues
- Mutexes and Condition Variables
- Read-Write Locks
- Record Locking
- Posix and System V Semaphores
- Posix and System V Shared Memory
- Solaris Doors and Sun RPC
- Performance Measurements of IPC Techniques

If you've read Stevens' best-selling first edition of **UNIX Network Programming**, this book expands its IPC coverage by a factor of five! You won't just learn about IPC "from the outside." You'll actually create implementations of Posix message queues, read-write locks, and semaphores, gaining an in-depth understanding of these capabilities you simply can't get anywhere else.

The book contains extensive new source code—all carefully optimized and available on the Web. You'll even find a complete guide to measuring IPC performance with message passing bandwidth and latency programs, and thread and process synchronization programs.

The better you understand IPC, the better your UNIX software will run. This book contains all you need to know.

ABOUT THE AUTHOR

W. RICHARD STEVENS is author of *UNIX Network Programming, First Edition,* widely recognized as the classic text in UNIX networking and *UNIX Network Programming, Volume 1, Second Edition.* He is also author of *Advanced Programming in the UNIX Environment* and the *TCP/IP Illustrated* Series. Stevens is an acknowledged UNIX and networking expert, sought-after instructor, and occasional consultant.

PRENTICE HALL
Upper Saddle River, NJ 07458
http://www.phptr.com

Function prototype	page
`bool_t clnt_control(CLIENT *cl, unsigned int request, char *ptr);`	418
`CLIENT *clnt_create(const char *host, unsigned long prognum, unsigned long versnum, const char *protocol);`	401
`void clnt_destroy(CLIENT *cl);`	420
`int door_bind(int fd);`	390
`int door_call(int fd, door_arg_t *argp);`	361
`int door_create(Door_server_proc *proc, void *cookie, u_int attr);`	363
`int door_cred(door_cred_t *cred);`	365
`int door_info(int fd, door_info_t *info);`	365
`int door_return(char *dataptr, size_t datasize, door_desc_t *descptr, size_t ndesc);`	365
`int door_revoke(int fd);`	390
`Door_create_proc *door_server_create(Door_create_proc *proc);`	384
`int door_unbind(void);`	390
`void err_dump(const char *fmt, ...);`	512
`void err_msg(const char *fmt, ...);`	512
`void err_quit(const char *fmt, ...);`	512
`void err_ret(const char *fmt, ...);`	511
`void err_sys(const char *fmt, ...);`	511
`int fcntl(int fd, int cmd, ... /* struct flock *arg */ );`	199
`int fstat(int fd, struct stat *buf);`	328
`key_t ftok(const char *pathname, int id);`	28
`int ftruncate(int fd, off_t length);`	327
`int mq_close(mqd_t mqdes);`	77
`int mq_getattr(mqd_t mqdes, struct mq_attr *attr);`	79
`int mq_notify(mqd_t mqdes, const struct sigevent *notification);`	87
`mqd_t mq_open(const char *name, int oflag, ... /* mode_t mode, struct mq_attr *attr */ );`	76
`int mq_unlink(const char *name);`	77
`int msgctl(int msqid, int cmd, struct msqid_ds *buff);`	134
`int msgget(key_t key, int oflag);`	130
`FILE *popen(const char *command, const char *type);`	52

Function prototype	page
`int pthread_cancel(pthread_t tid);`	187
`void pthread_cleanup_pop(int execute);`	187
`void pthread_cleanup_push(void (*function)(void *), void *arg);`	187
`int pthread_create(pthread_t *tid, const pthread_attr_t *attr, void *(*func)(void *), void *arg);`	502
`int pthread_detach(pthread_t tid);`	504
`void pthread_exit(void *status);`	504
`int pthread_join(pthread_t tid, void **status);`	503
`pthread_t pthread_self(void);`	503
`int pthread_condattr_destroy(pthread_condattr_t *attr);`	172
`int pthread_condattr_getpshared(const pthread_condattr_t *attr, int *valptr);`	173
`int pthread_condattr_init(pthread_condattr_t *attr);`	172
`int pthread_condattr_setpshared(pthread_condattr_t *attr, int value);`	173
`int pthread_cond_broadcast(pthread_cond_t *cptr);`	171
`int pthread_cond_destroy(pthread_cond_t *cptr);`	172
`int pthread_cond_init(pthread_cond_t *cptr, const pthread_condattr_t *attr);`	172
`int pthread_cond_signal(pthread_cond_t *cptr);`	167
`int pthread_cond_timedwait(pthread_cond_t *cptr, pthread_mutex_t *mptr, const struct timespec *abstime);`	171
`int pthread_cond_wait(pthread_cond_t *cptr, pthread_mutex_t *mptr);`	167
`int pthread_mutexattr_destroy(pthread_mutexattr_t *attr);`	172
`int pthread_mutexattr_getpshared(const pthread_mutexattr_t *attr, int *valptr);`	173
`int pthread_mutexattr_init(pthread_mutexattr_t *attr);`	172
`int pthread_mutexattr_setpshared(pthread_mutexattr_t *attr, int value);`	173
`int pthread_mutex_destroy(pthread_mutex_t *mptr);`	172
`int pthread_mutex_init(pthread_mutex_t *mptr, const pthread_mutexattr_t *attr);`	172
`int pthread_mutex_lock(pthread_mutex_t *mptr);`	160
`int pthread_mutex_trylock(pthread_mutex_t *mptr);`	160
`int pthread_mutex_unlock(pthread_mutex_t *mptr);`	160
`int pthread_rwlockattr_destroy(pthread_rwlockattr_t *attr);`	179
`int pthread_rwlockattr_getpshared(const pthread_rwlockattr_t *attr, int *valptr);`	179
`int pthread_rwlockattr_init(pthread_rwlockattr_t *attr);`	179
`int pthread_rwlockattr_setpshared(pthread_rwlockattr_t *attr, int value);`	179
`int pthread_rwlock_destroy(pthread_rwlock_t *rwptr);`	179
`int pthread_rwlock_init(pthread_rwlock_t *rwptr, const pthread_rwlockattr_t *attr);`	179
`int pthread_rwlock_rdlock(pthread_rwlock_t *rwptr);`	178
`int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwptr);`	178
`int pthread_rwlock_trywrlock(pthread_rwlock_t *rwptr);`	178
`int pthread_rwlock_unlock(pthread_rwlock_t *rwptr);`	178
`int pthread_rwlock_wrlock(pthread_rwlock_t *rwptr);`	178

UNIX Network Programming Volume 2

Second Edition

Interprocess Communications

by W. Richard Stevens

Prentice Hall PTR
Upper Saddle River, NJ 07458
www.phptr.com

Library of Congress Cataloging-in-Publication Data

Stevens, W. Richard.
UNIX network programming / by W. Richard Stevens. -- 2nd ed.
v. <1 > : ill. : 25 cm.
Includes bibliographical references and index.
Contents: v. 1. Networking APIs : sockets and XTI.
ISBN 0-13-490012-X (v. 1)
1. UNIX (Computer file) 2. Computer networks. 3. Internet programming. I. Title.
QA76.76.063S755 1998
005.7'12768--DC21 97-31761

Editorial/production supervision: *Patti Guerrieri*
Cover design director: *Jerry Votta*
Cover designer: *Scott Weiss*
Manufacturing manager: *Alexis R. Heydt*
Marketing manager: *Miles Williams*
Acquisitions editor: *Mary Franz*
Editorial assistant: *Noreen Regina*

Printed in the United States of America
10 9 8 7 6 5 4 3 2

ISBN 0-13-081081-9

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

To the Usenet community;
for many questions answered,
and many FAQs provided.

Abbreviated Table of Contents

Table of Contents

Preface

Introduction

Most nontrivial programs involve some form of *IPC* or *Interprocess Communication*. This is a natural effect of the design principle that the better approach is to design an application as a group of small pieces that communicate with each other, instead of designing one huge monolithic program. Historically, applications have been built in the following ways:

1. One huge monolithic program that does everything. The various pieces of the program can be implemented as functions that exchange information as function parameters, function return values, and global variables.
2. Multiple programs that communicate with each other using some form of IPC. Many of the standard Unix tools were designed in this fashion, using shell pipelines (a form of IPC) to pass information from one program to the next.
3. One program comprised of multiple threads that communicate with each other using some type of IPC. The term IPC describes this communication even though it is between threads and not between processes.

Combinations of the second two forms of design are also possible: multiple processes, each consisting of one or more threads, involving communication between the threads within a given process and between the different processes.

What I have described is distributing the work involved in performing a given application between multiple processes and perhaps among the threads within a process. On a system containing multiple processors (CPUs), multiple processes might be

able to run at the same time (on different CPUs), or the multiple threads of a given process might be able to run at the same time. Therefore, distributing an application among multiple processes or threads might reduce the amount of time required for an application to perform a given task.

This book describes four different forms of IPC in detail:

1. message passing (pipes, FIFOs, and message queues),
2. synchronization (mutexes, condition variables, read–write locks, file and record locks, and semaphores),
3. shared memory (anonymous and named), and
4. remote procedure calls (Solaris doors and Sun RPC).

This book does not cover the writing of programs that communicate across a computer network. This form of communication normally involves what is called the *sockets API* (application program interface) using the TCP/IP protocol suite; these topics are covered in detail in Volume 1 of this series [Stevens 1998].

One could argue that single-host or nonnetworked IPC (the subject of this volume) should not be used and instead all applications should be written as distributed applications that run on various hosts across a network. Practically, however, single-host IPC is often much faster and sometimes simpler than communicating across a network. Techniques such as shared memory and synchronization are normally available only on a single host, and may not be used across a network. Experience and history have shown a need for both nonnetworked IPC (this volume) and IPC across a network (Volume 1 of this series).

This current volume builds on the foundation of Volume 1 and my other four books, which are abbreviated throughout this text as follows:

- UNPv1: *UNIX Network Programming, Volume 1* [Stevens 1998],
- APUE: *Advanced Programming in the UNIX Environment* [Stevens 1992],
- TCPv1: *TCP/IP Illustrated, Volume 1* [Stevens 1994],
- TCPv2: *TCP/IP Illustrated, Volume 2* [Wright and Stevens 1995], and
- TCPv3: *TCP/IP Illustrated, Volume 3* [Stevens 1996].

Although covering IPC in a text with "network programming" in the title might seem odd, IPC is often used in networked applications. As stated in the Preface of the 1990 edition of *UNIX Network Programming*, "A requisite for understanding how to develop software for a network is an understanding of interprocess communication (IPC)."

Changes from the First Edition

This volume is a complete rewrite and expansion of Chapters 3 and 18 from the 1990 edition of *UNIX Network Programming*. Based on a word count, the material has expanded by a factor of five. The following are the major changes with this new edition:

- In addition to the three forms of "System V IPC" (message queues, semaphores, and shared memory), the newer Posix functions that implement these three types of IPC are also covered. (I say more about the Posix family of standards in Section 1.7.) In the coming years, I expect a movement to the Posix IPC functions, which have several advantages over their System V counterparts.
- The Posix functions for synchronization are covered: mutex locks, condition variables, and read–write locks. These can be used to synchronize either threads or processes and are often used when accessing shared memory.
- This volume assumes a Posix threads environment (called "Pthreads"), and many of the examples are built using multiple threads instead of multiple processes.
- The coverage of pipes, FIFOs, and record locking focuses on their Posix definitions.
- In addition to describing the IPC facilities and showing how to use them, I also develop implementations of Posix message queues, read–write locks, and Posix semaphores (all of which can be implemented as user libraries). These implementations can tie together many different features (e.g., one implementation of Posix semaphores uses mutexes, condition variables, and memory-mapped I/O) and highlight conditions that must often be handled in our applications (such as race conditions, error handling, memory leaks, and variable-length argument lists). Understanding an implementation of a certain feature often leads to a greater knowledge of how to use that feature.
- The RPC coverage focuses on the Sun RPC package. I precede this with a description of the new Solaris doors API, which is similar to RPC but on a single host. This provides an introduction to many of the features that we need to worry about when calling procedures in another process, without having to worry about any networking details.

Readers

This text can be used either as a tutorial on IPC, or as a reference for experienced programmers. The book is divided into four main parts:

- message passing,
- synchronization,
- shared memory, and
- remote procedure calls

but many readers will probably be interested in specific subsets. Most chapters can be read independently of others, although Chapter 2 summarizes many features common to all the Posix IPC functions, Chapter 3 summarizes many features common to all the System V IPC functions, and Chapter 12 is an introduction to both Posix and System V shared memory. All readers should read Chapter 1, especially Section 1.6, which describes some wrapper functions used throughout the text. The Posix IPC chapters are

independent of the System V IPC chapters, and the chapters on pipes, FIFOs, and record locking belong to neither camp. The two chapters on RPC are also independent of the other IPC techniques.

To aid in the use as a reference, a thorough index is provided, along with summaries on the end papers of where to find detailed descriptions of all the functions and structures. To help those reading topics in a random order, numerous references to related topics are provided throughout the text.

Source Code and Errata Availability

The source code for all the examples that appear in this book is available from the author's home page (listed at the end of this Preface). The best way to learn the IPC techniques described in this book is to take these programs, modify them, and enhance them. Actually writing code of this form is the *only* way to reinforce the concepts and techniques. Numerous exercises are also provided at the end of each chapter, and most answers are provided in Appendix D.

A current errata for this book is also available from the author's home page.

Acknowledgments

Although the author's name is the only one to appear on the cover, the combined effort of many people is required to produce a quality text book. First and foremost is the author's family, who put up with the long and weird hours that go into writing a book. Thank you once again, Sally, Bill, Ellen, and David.

My thanks to the technical reviewers who provided invaluable feedback (135 printed pages) catching lots of errors, pointing out areas that needed more explanation, and suggesting alternative presentations, wording, and coding: Gavin Bowe, Allen Briggs, Dave Butenhof, Wan-Teh Chang, Chris Cleeland, Bob Friesenhahn, Andrew Gierth, Scott Johnson, Marty Leisner, Larry McVoy, Craig Metz, Bob Nelson, Steve Rago, Jim Reid, Swamy K. Sitarama, Jon C. Snader, Ian Lance Taylor, Rich Teer, and Andy Tucker.

The following people answered email questions of mine, in some cases *many* questions, all of which improved the accuracy and presentation of the text: David Bausum, Dave Butenhof, Bill Gallmeister, Mukesh Kacker, Brian Kernighan, Larry McVoy, Steve Rago, Keith Skowran, Bart Smaalders, Andy Tucker, and John Wait.

A special thanks to Larry Rafsky at GSquared, for lots of things. My thanks as usual to the National Optical Astronomy Observatories (NOAO), Sidney Wolff, Richard Wolff, and Steve Grandi, for providing access to their networks and hosts. Jim Bound, Matt Thomas, Mary Clouter, and Barb Glover of Digital Equipment Corp. provided the Alpha system used for most of the examples in this text. A subset of the code in this book was tested on other Unix systems: my thanks to Michael Johnson of Red Hat Software for providing the latest releases of Red Hat Linux, and to Dave Marquardt and Jessie Haug of IBM Austin for an RS/6000 system and access to the latest releases of AIX.

My thanks to the wonderful staff at Prentice Hall—my editor Mary Franz, along with Noreen Regina, Sophie Papanikolaou, and Patti Guerrieri—for all their help, especially in bringing everything together on a tight schedule.

Colophon

I produced camera-ready copy of the book (PostScript), which was then typeset for the final book. The formatting system used was James Clark's wonderful `groff` package, on a SparcStation running Solaris 2.6. (Reports of troff's death are greatly exaggerated.) I typed in all 138,897 words using the `vi` editor, created the 72 illustrations using the `gpic` program (using many of Gary Wright's macros), produced the 35 tables using the `gtbl` program, performed all the indexing (using a set of awk scripts written by Jon Bentley and Brian Kernighan), and did the final page layout. Dave Hanson's `loom` program, the GNU `indent` program, and some scripts by Gary Wright were used to include the 8,046 lines of C source code in the book.

I welcome email from any readers with comments, suggestions, or bug fixes.

Tucson, Arizona
July 1998

W. Richard Stevens
rstevens@kohala.com
http://www.kohala.com/~rstevens

Part 1

Introduction

1

Introduction

1.1 Introduction

IPC stands for *interprocess communication*. Traditionally the term describes different ways of *message passing* between different processes that are running on some operating system. This text also describes numerous forms of *synchronization*, because newer forms of communication, such as shared memory, require some form of synchronization to operate.

In the evolution of the Unix operating system over the past 30 years, message passing has evolved through the following stages:

- *Pipes* (Chapter 4) were the first widely used form of IPC, available both within programs and from the shell. The problem with pipes is that they are usable only between processes that have a common ancestor (i.e., a parent–child relationship), but this was fixed with the introduction of *named pipes* or *FIFOs* (Chapter 4).
- *System V message queues* (Chapter 6) were added to System V kernels in the early 1980s. These can be used between related or unrelated processes on a given host. Although these are still referred to with the "System V" prefix, most versions of Unix today support them, regardless of whether their heritage is System V or not.

When describing Unix processes, the term *related* means the processes have some ancestor in common. This is another way of saying that these related processes were generated

from this ancestor by one or more `fork`s. A common example is when a process calls `fork` twice, generating two child processes. We then say that these two children are related. Similarly, each child is related to the parent. With regard to IPC, the parent can establish some form of IPC before calling `fork` (a pipe or message queue, for example), knowing that the two children will inherit this IPC object across the `fork`. We talk more about the inheritance of the various IPC objects with Figure 1.6. We must also note that all Unix processes are theoretically related to the `init` process, which starts everything going when a system is bootstrapped. Practically speaking, however, process relationships start with a login shell (called a *session*) and all the processes generated by that shell. Chapter 9 of APUE talks about sessions and process relationships in more detail.

Throughout the text, we use indented, parenthetical notes such as this one to describe implementation details, historical points, and minutiae.

- *Posix message queues* (Chapter 5) were added by the Posix realtime standard (1003.1b–1993, which we say more about in Section 1.7). These can be used between related or unrelated processes on a given host.
- *Remote Procedure Calls* (RPCs, which we cover in Part 5) appeared in the mid-1980s as a way of calling a function on one system (the server) from a program on another system (the client), and was developed as an alternative to explicit network programming. Since information is normally passed between the client and server (the arguments and return values of the function that is called), and since RPC can be used between a client and server on the same host, RPC can be considered as another form of message passing.

Looking at the evolution of the various forms of synchronization provided by Unix is also interesting.

- Early programs that needed some form of synchronization (often to prevent multiple processes from modifying the same file at the same time) used quirks of the filesystem, some of which we talk about in Section 9.8.
- *Record locking* (Chapter 9) was added to Unix kernels in the early 1980s and then standardized by Posix.1 in 1988.
- *System V semaphores* (Chapter 11) were added along with *System V shared memory* (Chapter 14) at the same time System V message queues were added (early 1980s). Most versions of Unix support these today.
- *Posix semaphores* (Chapter 10) and *Posix shared memory* (Chapter 13) were also added by the Posix realtime standard (1003.1b–1993, which we mentioned with regard to Posix message queues earlier).
- *Mutexes* and *condition variables* (Chapter 7) are two forms of synchronization defined by the Posix threads standard (1003.1c–1995). Although these are often used for synchronization between threads, they can also provide synchronization between different processes.
- *Read–write locks* (Chapter 8) are an additional form of synchronization. These have not yet been standardized by Posix, but probably will be soon.

1.2 Processes, Threads, and the Sharing of Information

In the traditional Unix programming model, we have multiple processes running on a system, with each process having its own address space. Information can be shared between Unix processes in various ways. We summarize these in Figure 1.1.

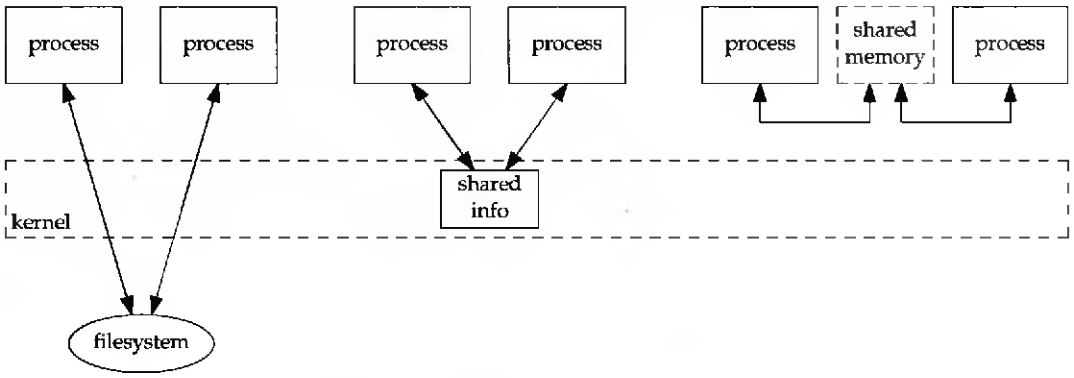

Figure 1.1 Three ways to share information between Unix processes.

1. The two processes on the left are sharing some information that resides in a file in the filesystem. To access this data, each process must go through the kernel (e.g., `read`, `write`, `lseek`, and the like). Some form of synchronization is required when a file is being updated, both to protect multiple writers from each other, and to protect one or more readers from a writer.
2. The two processes in the middle are sharing some information that resides within the kernel. A pipe is an example of this type of sharing, as are System V message queues and System V semaphores. Each operation to access the shared information now involves a system call into the kernel.
3. The two processes on the right have a region of shared memory that each process can reference. Once the shared memory is set up by each process, the processes can access the data in the shared memory without involving the kernel at all. Some form of synchronization is required by the processes that are sharing the memory.

Note that nothing restricts any of the IPC techniques that we describe to only two processes. Any of the techniques that we describe work with any number of processes. We show only two processes in Figure 1.1 for simplicity.

Threads

Although the concept of a process within the Unix system has been used for a long time, the concept of multiple *threads* within a given process is relatively new. The Posix.1 threads standard (called "Pthreads") was approved in 1995. From an IPC perspective,

all the threads within a given process share the same global variables (e.g., the concept of shared memory is inherent to this model). What we must worry about, however, is *synchronizing* access to this global data among the various threads. Indeed, synchronization, though not explicitly a form of IPC, is used with many forms of IPC to control access to some shared data.

In this text, we describe IPC between processes and IPC between threads. We assume a threads environment and make statements of the form "if the pipe is empty, the calling thread is blocked in its call to `read` until some thread writes data to the pipe." If your system does not support threads, you can substitute "process" for "thread" in this sentence, providing the classic Unix definition of blocking in a `read` of an empty pipe. But on a system that supports threads, only the thread that calls `read` on an empty pipe is blocked, and the remaining threads in the process can continue to execute. Writing data to this empty pipe can be done by another thread in the same process or by some thread in another process.

Appendix B summarizes some of the characteristics of threads and the five basic Pthread functions that are used throughout this text.

1.3 Persistence of IPC Objects

We can define the *persistence* of any type of IPC as how long an object of that type remains in existence. Figure 1.2 shows three types of persistence.

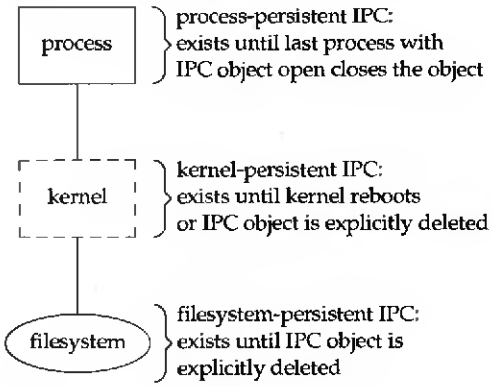

Figure 1.2 Persistence of IPC objects.

1. A *process-persistent* IPC object remains in existence until the last process that holds the object open closes the object. Examples are pipes and FIFOs.
2. A *kernel-persistent* IPC object remains in existence until the kernel reboots or until the object is explicitly deleted. Examples are System V message queues, semaphores, and shared memory. Posix message queues, semaphores, and shared memory must be at least kernel-persistent, but may be file-system-persistent, depending on the implementation.

3. A *filesystem-persistent* IPC object remains in existence until the object is explicitly deleted. The object retains its value even if the kernel reboots. Posix message queues, semaphores, and shared memory have this property, if they are implemented using mapped files (not a requirement).

We must be careful when defining the persistence of an IPC object because it is not always as it seems. For example, the data within a pipe is maintained within the kernel, but pipes have process persistence and not kernel persistence—after the last process that has the pipe open for reading closes the pipe, the kernel discards all the data and removes the pipe. Similarly, even though FIFOs have names within the filesystem, they also have process persistence because all the data in a FIFO is discarded after the last process that has the FIFO open closes the FIFO.

Figure 1.3 summarizes the persistence of the IPC objects that we describe in this text.

Type of IPC	Persistence
Pipe	process
FIFO	process
Posix mutex	process
Posix condition variable	process
Posix read–write lock	process
`fcntl` record locking	process
Posix message queue	kernel
Posix named semaphore	kernel
Posix memory-based semaphore	process
Posix shared memory	kernel
System V message queue	kernel
System V semaphore	kernel
System V shared memory	kernel
TCP socket	process
UDP socket	process
Unix domain socket	process

Figure 1.3 Persistence of various types of IPC objects.

Note that no type of IPC has filesystem persistence, but we have mentioned that the three types of Posix IPC may, depending on the implementation. Obviously, writing data to a file provides filesystem persistence, but this is normally not used as a form of IPC. Most forms of IPC are not intended to survive a system reboot, because the processes do not survive the reboot. Requiring filesystem persistence would probably degrade the performance for a given form of IPC, and a common design goal for IPC is high performance.

1.4 Name Spaces

When two unrelated processes use some type of IPC to exchange information between themselves, the IPC object must have a *name* or *identifier* of some form so that one

process (often a server) can create the IPC object and other processes (often one or more clients) can specify that same IPC object.

Pipes do not have names (and therefore cannot be used between unrelated processes), but FIFOs have a Unix pathname in the filesystem as their identifier (and can therefore be used between unrelated processes). As we move to other forms of IPC in the following chapters, we use additional naming conventions. The set of possible names for a given type of IPC is called its *name space*. The name space is important, because with all forms of IPC other than plain pipes, the name is how the client and server connect with each other to exchange messages.

Figure 1.4 summarizes the naming conventions used by the different forms of IPC.

Type of IPC	Name space to open or create	Identification after IPC opened	Posix.1 1996	Unix 98
Pipe	(no name)	descriptor	•	•
FIFO	pathname	descriptor	•	•
Posix mutex	(no name)	`pthread_mutex_t` ptr	•	•
Posix condition variable	(no name)	`pthread_cond_t` ptr	•	•
Posix read–write lock	(no name)	`pthread_rwlock_t` ptr		•
`fcntl` record locking	pathname	descriptor	•	•
Posix message queue	Posix IPC name	`mqd_t` value	•	•
Posix named semaphore	Posix IPC name	`sem_t` pointer	•	•
Posix memory-based semaphore	(no name)	`sem_t` pointer	•	•
Posix shared memory	Posix IPC name	descriptor	•	•
System V message queue	`key_t` key	System V IPC identifier		•
System V semaphore	`key_t` key	System V IPC identifier		•
System V shared memory	`key_t` key	System V IPC identifier		•
Doors	pathname	descriptor		
Sun RPC	program/version	RPC handle		
TCP socket	IP addr & TCP port	descriptor	.1g	•
UDP socket	IP addr & UDP port	descriptor	.1g	•
Unix domain socket	pathname	descriptor	.1g	•

Figure 1.4 Name spaces for the various forms of IPC.

We also indicate which forms of IPC are standardized by the 1996 version of Posix.1 and Unix 98, both of which we say more about in Section 1.7. For comparison purposes, we include three types of sockets, which are described in detail in UNPv1. Note that the sockets API (application program interface) is being standardized by the Posix.1g working group and should eventually become part of a future Posix.1 standard.

Even though Posix.1 standardizes semaphores, they are an optional feature. Figure 1.5 summarizes which features are specified by Posix.1 and Unix 98. Each feature is mandatory, not defined, or optional. For the optional features, we specify the name of the constant (e.g., `_POSIX_THREADS`) that is defined (normally in the `<unistd.h>` header) if the feature is supported. Note that Unix 98 is a superset of Posix.1.

Type of IPC	Posix.1 1996	Unix 98
Pipe	mandatory	mandatory
FIFO	mandatory	mandatory
Posix mutex	_POSIX_THREADS	mandatory
Posix condition variable	_POSIX_THREADS	mandatory
process-shared mutex/CV	_POSIX_THREAD_PROCESS_SHARED	mandatory
Posix read–write lock	(not defined)	mandatory
fcntl record locking	mandatory	mandatory
Posix message queue	_POSIX_MESSAGE_PASSING	_XOPEN_REALTIME
Posix semaphores	_POSIX_SEMAPHORES	_XOPEN_REALTIME
Posix shared memory	_POSIX_SHARED_MEMORY_OBJECTS	_XOPEN_REALTIME
System V message queue	(not defined)	mandatory
System V semaphore	(not defined)	mandatory
System V shared memory	(not defined)	mandatory
Doors	(not defined)	(not defined)
Sun RPC	(not defined)	(not defined)
mmap	_POSIX_MAPPED_FILES or _POSIX_SHARED_MEMORY_OBJECTS	mandatory
Realtime signals	_POSIX_REALTIME_SIGNALS	_XOPEN_REALTIME

Figure 1.5 Availability of the various forms of IPC.

1.5 Effect of fork, exec, and exit on IPC Objects

We need to understand the effect of the fork, exec, and _exit functions on the various forms of IPC that we discuss. (The latter is called by the exit function.) We summarize this in Figure 1.6.

Most of these features are described later in the text, but we need to make a few points. First, the calling of fork from a multithreaded process becomes messy with regard to unnamed synchronization variables (mutexes, condition variables, read–write locks, and memory-based semaphores). Section 6.1 of [Butenhof 1997] provides the details. We simply note in the table that if these variables reside in shared memory and are created with the process-shared attribute, then they remain accessible to any thread of any process with access to that shared memory. Second, the three forms of System V IPC have no notion of being open or closed. We will see in Figure 6.8 and Exercises 11.1 and 14.1 that all we need to know to access these three forms of IPC is an identifier. So these three forms of IPC are available to any process that knows the identifier, although some special handling is indicated for semaphores and shared memory.

Type of IPC	fork	exec	_exit
Pipes and FIFOs	child gets copies of all parent's open descriptors	all open descriptors remain open unless descriptor's FD_CLOEXEC bit set	all open descriptors closed; all data removed from pipe or FIFO on last close
Posix message queues	child gets copies of all parent's open message queue descriptors	all open message queue descriptors are closed	all open message queue descriptors are closed
System V message queues	no effect	no effect	no effect
Posix mutexes and condition variables	shared if in shared memory and process-shared attribute	vanishes unless in shared memory that stays open and process-shared attribute	vanishes unless in shared memory that stays open and process-shared attribute
Posix read–write locks	shared if in shared memory and process-shared attribute	vanishes unless in shared memory that stays open and process-shared attribute	vanishes unless in shared memory that stays open and process-shared attribute
Posix memory-based semaphores	shared if in shared memory and process-shared attribute	vanishes unless in shared memory that stays open and process-shared attribute	vanishes unless in shared memory that stays open and process-shared attribute
Posix named semaphores	all open in parent remain open in child	any open are closed	any open are closed
System V semaphores	all semadj values in child are set to 0	all semadj values carried over to new program	all semadj values are added to corresponding semaphore value
fcntl record locking	locks held by parent are not inherited by child	locks are unchanged as long as descriptor remains open	all outstanding locks owned by process are unlocked
mmap memory mappings	memory mappings in parent are retained by child	memory mappings are unmapped	memory mappings are unmapped
Posix shared memory	memory mappings in parent are retained by child	memory mappings are unmapped	memory mappings are unmapped
System V shared memory	attached shared memory segments remain attached by child	attached shared memory segments are detached	attached shared memory segments are detached
Doors	child gets copies of all parent's open descriptors but only parent is a server for door invocations on door descriptors	all door descriptors should be closed because they are created with FD_CLOEXEC bit set	all open descriptors closed

Figure 1.6 Effect of calling fork, exec, and _exit on IPC.

1.6 Error Handling: Wrapper Functions

In any real-world program, we must check *every* function call for an error return. Since terminating on an error is the common case, we can shorten our programs by defining a *wrapper function* that performs the actual function call, tests the return value, and terminates on an error. The convention we use is to capitalize the name of the function, as in

```
Sem_post(ptr);
```

Our wrapper function is shown in Figure 1.7.

lib/wrapunix.c

```
387 void
388 Sem_post(sem_t *sem)
389 {
390     if (sem_post(sem) == -1)
391         err_sys("sem_post error");
392 }
```

lib/wrapunix.c

Figure 1.7 Our wrapper function for the `sem_post` function.

Whenever you encounter a function name in the text that begins with a capital letter, that is a wrapper function of our own. It calls a function whose name is the same but begins with the lowercase letter. The wrapper function always terminates with an error message if an error is encountered.

When describing the source code that is presented in the text, we always refer to the lowest-level function being called (e.g., `sem_post`) and not the wrapper function (e.g., `Sem_post`). Similarly the index always refers to the lowest level function being called, and not the wrapper functions.

The format of the source code just shown is used throughout the text. Each nonblank line is numbered. The text describing portions of the code begins with the starting and ending line numbers in the left margin. Sometimes the paragraph is preceded by a short descriptive bold heading, providing a summary statement of the code being described.

The horizontal rules at the beginning and end of the code fragment specify the source code filename: the file `wrapunix.c` in the directory `lib` for this example. Since the source code for all the examples in the text is freely available (see the Preface), you can locate the appropriate source file. Compiling, running, and especially modifying these programs while reading this text is an excellent way to learn the concepts of interprocess communications.

Although these wrapper functions might not seem like a big savings, when we discuss threads in Chapter 7, we will find that the thread functions do not set the standard Unix `errno` variable when an error occurs; instead the `errno` value is the return value of the function. This means that every time we call one of the pthread functions, we must allocate a variable, save the return value in that variable, and then set `errno` to this value before calling our `err_sys` function (Figure C.4). To avoid cluttering the code with braces, we can use C's comma operator to combine the assignment into `errno` and the call of `err_sys` into a single statement, as in the following:

```
int    n;

if ( (n = pthread_mutex_lock(&ndone_mutex)) != 0)
    errno = n, err_sys("pthread_mutex_lock error");
```

Alternately, we could define a new error function that takes the system's error number as an argument. But we can make this piece of code much easier to read as just

```
Pthread_mutex_lock(&ndone_mutex);
```

by defining our own wrapper function, shown in Figure 1.8.

lib/wrappthread.c

```
125 void
126 Pthread_mutex_lock(pthread_mutex_t *mptr)
127 {
128     int     n;

129     if ( (n = pthread_mutex_lock(mptr)) == 0)
130         return;
131     errno = n;
132     err_sys("pthread_mutex_lock error");
133 }
```

lib/wrappthread.c

Figure 1.8 Our wrapper function for `pthread_mutex_lock`.

With careful C coding, we could use macros instead of functions, providing a little run-time efficiency, but these wrapper functions are rarely, if ever, the performance bottleneck of a program.

Our choice of capitalizing the first character of the function name is a compromise. Many other styles were considered: prefixing the function name with an `e` (as done on p. 182 of [Kernighan and Pike 1984]), appending `_e` to the function name, and so on. Our style seems the least distracting while still providing a visual indication that some other function is really being called.

This technique has the side benefit of checking for errors from functions whose error returns are often ignored: `close` and `pthread_mutex_lock`, for example.

Throughout the rest of this book, we use these wrapper functions unless we need to check for an explicit error and handle it in some form other than terminating the process. We do not show the source code for all our wrapper functions, but the code is freely available (see the Preface).

Unix `errno` Value

When an error occurs in a Unix function, the global variable `errno` is set to a positive value, indicating the type of error, and the function normally returns –1. Our `err_sys` function looks at the value of `errno` and prints the corresponding error message string (e.g., "Resource temporarily unavailable" if `errno` equals `EAGAIN`).

The value of `errno` is set by a function only if an error occurs. Its value is undefined if the function does not return an error. All the positive error values are constants with an all-uppercase name beginning with `E` and are normally defined in the

<sys/errno.h> header. No error has the value of 0.

With multiple threads, each thread must have its own errno variable. Providing a per-thread errno is handled automatically, although this normally requires telling the compiler that the program being compiled must be reentrant. Specifying something like -D_REENTRANT or -D_POSIX_C_SOURCE=199506L to the compiler is typically required. Often the <errno.h> header defines errno as a macro that expands into a function call when _REENTRANT is defined, referencing a per-thread copy of the error variable.

Throughout the text, we use phrases of the form "the mq_send function returns EMSGSIZE" as shorthand to mean that the function returns an error (typically a return value of –1) with errno set to the specified constant.

1.7 Unix Standards

Most activity these days with regard to Unix standardization is being done by Posix and The Open Group.

POSIX

Posix is an acronym for "Portable Operating System Interface." Posix is not a single standard, but a family of standards being developed by the Institute for Electrical and Electronics Engineers, Inc., normally called the *IEEE*. The Posix standards are also being adopted as international standards by ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission), called ISO/IEC. The Posix standards have gone through the following iterations.

- IEEE Std 1003.1–1988 (317 pages) was the first of the Posix standards. It specified the C language interface into a Unix-like kernel covering the following areas: process primitives (fork, exec, signals, timers), the environment of a process (user IDs, process groups), files and directories (all the I/O functions), terminal I/O, the system databases (password file and group file), and the tar and cpio archive formats.

  The first Posix standard was a trial use version in 1986 known as "IEEEIX." The name Posix was suggested by Richard Stallman.

- IEEE Std 1003.1–1990 (356 pages) was next and it was also International Standard ISO/IEC 9945–1: 1990. Minimal changes were made from the 1988 version to the 1990 version. Appended to the title was "Part 1: System Application Program Interface (API) [C Language]" indicating that this standard was the C language API.

- IEEE Std 1003.2–1992 was published in two volumes, totaling about 1300 pages, and its title contained "Part 2: Shell and Utilities." This part defines the shell (based on the System V Bourne shell) and about 100 utilities (programs normally executed from a shell, from awk and basename to vi and yacc). Throughout this text, we refer to this standard as *Posix.2*.

- IEEE Std 1003.1b–1993 (590 pages) was originally known as IEEE P1003.4. This was an update to the 1003.1–1990 standard to include the realtime extensions developed by the P1003.4 working group: file synchronization, asynchronous I/O, semaphores, memory management (`mmap` and shared memory), execution scheduling, clocks and timers, and message queues.

- IEEE Std 1003.1, 1996 Edition [IEEE 1996] (743 pages) includes 1003.1–1990 (the base API), 1003.1b–1993 (realtime extensions), 1003.1c–1995 (Pthreads), and 1003.1i–1995 (technical corrections to 1003.1b). This standard is also called ISO/IEC 9945–1: 1996. Three chapters on threads were added, along with additional sections on thread synchronization (mutexes and condition variables), thread scheduling, and synchronization scheduling. Throughout this text, we refer to this standard as *Posix.1*.

  Over one-quarter of the 743 pages are an appendix titled "Rationale and Notes." This rationale contains historical information and reasons why certain features were included or omitted. Often the rationale is as informative as the official standard.

  Unfortunately, the IEEE standards are not freely available on the Internet. Ordering information is given in the Bibliography entry for [IEEE 1996].

  Note that semaphores were defined in the realtime standard, separately from mutexes and condition variables (which were defined in the Pthreads standard), which accounts for some of the differences that we see in their APIs.

  Finally, note that read–write locks are not (yet) part of any Posix standard. We say more about this in Chapter 8.

Sometime in the future, a new version of IEEE Std 1003.1 should be printed to include the P1003.1g standard, the networking APIs (sockets and XTI), which are described in UNPv1.

The Foreword of the 1996 Posix.1 standard states that ISO/IEC 9945 consists of the following parts:

- Part 1: System application program interface (API) [C language],
- Part 2: Shell and utilities, and
- Part 3: System administration (under development).

Parts 1 and 2 are what we call Posix.1 and Posix.2.

Work on all of the Posix standards continues and it is a moving target for any book that attempts to cover it. The current status of the various Posix standards is available from `http://www.pasc.org/standing/sd11.html`.

The Open Group

The Open Group was formed in 1996 by the consolidation of the X/Open Company (founded in 1984) and the Open Software Foundation (OSF, founded in 1988). It is an international consortium of vendors and end-user customers from industry, government, and academia. Their standards have gone through the following iterations:

- X/Open published the *X/Open Portability Guide*, Issue 3 (XPG3) in 1989.
- Issue 4 was published in 1992 followed by Issue 4, Version 2 in 1994. This latest version was also known as "Spec 1170," with the magic number 1170 being the sum of the number of system interfaces (926), the number of headers (70), and the number of commands (174). The latest name for this set of specifications is the "X/Open Single Unix Specification," although it is also called "Unix 95."
- In March 1997, Version 2 of the Single Unix Specification was announced. Products conforming to this specification can be called "Unix 98," which is how we refer to this specification throughout this text. The number of interfaces required by Unix 98 increases from 1170 to 1434, although for a workstation, this jumps to 3030, because it includes the CDE (Common Desktop Environment), which in turn requires the X Window System and the Motif user interface. Details are available in [Josey 1997] and `http://www.UNIX-systems.org/version2`.

Much of the Single Unix Specification is freely available on the Internet from this URL.

Unix Versions and Portability

Most Unix systems today conform to some version of Posix.1 and Posix.2. We use the qualifier "some" because as updates to Posix occur (e.g., the realtime extensions in 1993 and the Pthreads addition in 1996), vendors take a year or two (sometimes more) to incorporate these latest changes.

Historically, most Unix systems show either a Berkeley heritage or a System V heritage, but these differences are slowly disappearing as most vendors adopt the Posix standards. The main differences still existing deal with system administration, one area that no Posix standard currently addresses.

Throughout this text, we use Solaris 2.6 and Digital Unix 4.0B for most examples. The reason is that at the time of this writing (late 1997 to early 1998), these were the only two Unix systems that supported System V IPC, Posix IPC, and Posix threads.

1.8 Road Map to IPC Examples in the Text

Three patterns of interaction are used predominantly throughout the text to illustrate various features:

1. File server: a client–server application in which the client sends the server a pathname and the server returns the contents of that file to the client.
2. Producer–consumer: one or more threads or processes (producers) place data into a shared buffer, and one or more threads or processes (consumers) operate on the data in the shared buffer.

3. Sequence-number-increment: one or more threads or processes increment a shared sequence number. Sometimes the sequence number is in a shared file, and sometimes it is in shared memory.

The first example illustrates the various forms of message passing, whereas the other two examples illustrate the various types of synchronization and shared memory.

To provide a road map for the different topics that are covered in this text, Figures 1.9, 1.10, and 1.11 summarize the programs that we develop, and the starting figure number and page number in which the source code appears.

1.9 Summary

IPC has traditionally been a messy area in Unix. Various solutions have been implemented, none of which are perfect. Our coverage is divided into four main areas:

1. message passing (pipes, FIFOs, message queues),
2. synchronization (mutexes, condition variables, read–write locks, semaphores),
3. shared memory (anonymous, named), and
4. procedure calls (Solaris doors, Sun RPC).

We consider IPC between multiple threads in a single process, and between multiple processes.

The persistence of each type of IPC as either can be process-persistent, kernel-persistent, or filesystem-persistent, based on how long the IPC object stays in existence. When choosing the type of IPC to use for a given application, we must be aware of the persistence of that IPC object.

Another feature of each type of IPC is its name space: how IPC objects are identified by the processes and threads that use the IPC object. Some have no name (pipes, mutexes, condition variables, read–write locks), some have names in the filesystem (FIFOs), some have what we describe in Chapter 2 as Posix IPC names, and some have other types of names (what we describe in Chapter 3 as System V IPC keys or identifiers). Typically, a server creates an IPC object with some name and the clients use that name to access the IPC object.

Throughout the source code in the text, we use the wrapper functions described in Section 1.6 to reduce the size of our code, yet still check every function call for an error return. Our wrapper functions all begin with a capital letter.

The IEEE Posix standards—Posix.1 defining the basic C interface to Unix and Posix.2 defining the standard commands—have been the standards that most vendors are moving toward. The Posix standards, however, are rapidly being absorbed and expanded by the commercial standards, notably The Open Group's Unix standards, such as Unix 98.

Figure	Page	Description
4.8	47	Uses two pipes, parent–child
4.15	53	Uses popen and `cat`
4.16	55	Uses two FIFOs, parent–child
4.18	57	Uses two FIFOs, stand-alone server, unrelated client
4.23	62	Uses FIFOs, stand-alone iterative server, multiple clients
4.25	68	Uses pipe or FIFO: builds records on top of byte stream
6.9	141	Uses two System V message queues
6.15	144	Uses one System V message queue, multiple clients
6.20	148	Uses one System V message queue per client, multiple clients
15.18	381	Uses descriptor passing across a door

Figure 1.9 Different versions of the file server client–server example.

Figure	Page	Description
7.2	162	Mutex only, multiple producers, one consumer
7.6	168	Mutex and condition variable, multiple producers, one consumer
10.17	236	Posix named semaphores, one producer, one consumer
10.20	242	Posix memory-based semaphores, one producer, one consumer
10.21	243	Posix memory-based semaphores, multiple producers, one consumer
10.24	246	Posix memory-based semaphores, multiple producers, multiple consumers
10.33	254	Posix memory-based semaphores, one producer, one consumer: multiple buffers

Figure 1.10 Different versions of the producer–consumer example.

Figure	Page	Description
9.1	194	Seq# in file, no locking
9.3	201	Seq# in file, `fcntl` locking
9.12	215	Seq# in file, filesystem locking using `open`
10.19	239	Seq# in file, Posix named semaphore locking
12.10	312	Seq# in `mmap` shared memory, Posix named semaphore locking
12.12	314	Seq# in `mmap` shared memory, Posix memory-based semaphore locking
12.14	316	Seq# in 4.4BSD anonymous shared memory, Posix named semaphore locking
12.15	316	Seq# in SVR4 `/dev/zero` shared memory, Posix named semaphore locking
13.7	334	Seq# in Posix shared memory, Posix memory-based semaphore locking
A.34	487	Performance measurement: mutex locking between threads
A.36	489	Performance measurement: read–write locking between threads
A.39	491	Performance measurement: Posix memory-based semaphore locking between threads
A.41	493	Performance measurement: Posix named semaphore locking between threads
A.42	494	Performance measurement: System V semaphore locking between threads
A.45	496	Performance measurement: `fcntl` record locking between threads
A.48	499	Performance measurement: mutex locking between processes

Figure 1.11 Different versions of the sequence-number-increment example.

Exercises

1.1 In Figure 1.1 we show two processes accessing a single file. If both processes are just appending new data to the end of the file (a log file perhaps), what kind of synchronization is required?

1.2 Look at your system's `<errno.h>` header and see how it defines `errno`.

1.3 Update Figure 1.5 by noting the features supported by the Unix systems that you use.

2

Posix IPC

2.1 Introduction

The three types of IPC,

- Posix message queues (Chapter 5),
- Posix semaphores (Chapter 10), and
- Posix shared memory (Chapter 13)

are collectively referred to as "Posix IPC." They share some similarities in the functions that access them, and in the information that describes them. This chapter describes all these common properties: the pathnames used for identification, the flags specified when opening or creating, and the access permissions.

A summary of their functions is shown in Figure 2.1.

2.2 IPC Names

In Figure 1.4, we noted that the three types of Posix IPC use "Posix IPC names" for their identification. The first argument to the three functions `mq_open`, `sem_open`, and `shm_open` is such a name, which may or may not be a real pathname in a filesystem. All that Posix.1 says about these names is:

- It must conform to existing rules for pathnames (must consist of at most `PATH_MAX` bytes, including a terminating null byte).
- If it begins with a slash, then different calls to these functions all reference the same queue. If it does not begin with a slash, the effect is implementation dependent.

	Message queues	Semaphores	Shared memory
Header	<mqueue.h>	<semaphore.h>	<sys/mman.h>
Functions to create, open, or delete	mq_open mq_close mq_unlink	sem_open sem_close sem_unlink	shm_open shm_unlink
		sem_init sem_destroy	
Functions for control operations	mq_getattr mq_setattr		ftruncate fstat
Functions for IPC operations	mq_send mq_receive mq_notify	sem_wait sem_trywait sem_post sem_getvalue	mmap munmap

Figure 2.1 Summary of Posix IPC functions.

- The interpretation of additional slashes in the name is implementation defined.

So, for portability, these names must begin with a slash and must not contain any other slashes. Unfortunately, these rules are inadequate and lead to portability problems.

Solaris 2.6 requires the initial slash but forbids any additional slashes. Assuming a message queue, it then creates three files in `/tmp` that begin with `.MQ`. For example, if the argument to `mq_open` is `/queue.1234`, then the three files are `/tmp/.MQDqueue.1234`, `/tmp/.MQLqueue.1234`, and `/tmp/.MQPqueue.1234`. Digital Unix 4.0B, on the other hand, creates the specified pathname in the filesystem.

The portability problem occurs if we specify a *name* with only one slash (as the first character): we must have write permission in that directory, the root directory. For example, `/tmp.1234` abides by the Posix rules and would be OK under Solaris, but Digital Unix would try to create this file, and unless we have write permission in the root directory, this attempt would fail. If we specify a *name* of `/tmp/test.1234`, this will succeed on all systems that create an actual file with that name (assuming that the `/tmp` directory exists and that we have write permission in that directory, which is normal for most Unix systems), but fails under Solaris.

To avoid these portability problems we should always `#define` the *name* in a header that is easy to change if we move our application to another system.

> This case is one in which the standard tries to be so general (in this case, the realtime standard was trying to allow message queue, semaphore, and shared memory implementations all within existing Unix kernels and as stand-alone diskless systems) that the standard's solution is nonportable. Within Posix, this is called "a standard way of being nonstandard."

Posix.1 defines the three macros

```
S_TYPEISMQ(buf)
S_TYPEISSEM(buf)
S_TYPEISSHM(buf)
```

that take a single argument, a pointer to a `stat` structure, whose contents are filled in by the `fstat`, `lstat`, or `stat` functions. These three macros evaluate to a nonzero value if the specified IPC object (message queue, semaphore, or shared memory object) is implemented as a distinct file type and the `stat` structure references such a file type. Otherwise, the macros evaluate to 0.

> Unfortunately, these macros are of little use, since there is no guarantee that these three types of IPC are implemented using a distinct file type. Under Solaris 2.6, for example, all three macros always evaluate to 0.
>
> All the other macros that test for a given file type have names beginning with `S_IS` and their single argument is the `st_mode` member of a `stat` structure. Since these three new macros have a different argument, their names were changed to begin with `S_TYPEIS`.

`px_ipc_name` Function

Another solution to this portability problem is to define our own function named `px_ipc_name` that prefixes the correct directory for the location of Posix IPC names.

```
#include "unpipc.h"

char *px_ipc_name(const char *name);
```

Returns: nonnull pointer if OK, NULL on error

> This is the notation we use for functions of our own throughout this book that are not standard system functions: the box around the function prototype and return value is dashed. The header that is included at the beginning is usually our `unpipc.h` header (Figure C.1).

The *name* argument should not contain any slashes. For example, the call

```
px_ipc_name("test1")
```

returns a pointer to the string `/test1` under Solaris 2.6 or a pointer to the string `/tmp/test1` under Digital Unix 4.0B. The memory for the result string is dynamically allocated and is returned by calling `free`. Additionally, the environment variable `PX_IPC_NAME` can override the default directory.

Figure 2.2 shows our implementation of this function.

> This may be your first encounter with `snprintf`. Lots of existing code calls `sprintf` instead, but `sprintf` cannot check for overflow of the destination buffer. `snprintf`, on the other hand, requires that the second argument be the size of the destination buffer, and this buffer will not be overflowed. Providing input that intentionally overflows a program's `sprintf` buffer has been used for many years by hackers breaking into systems.
>
> `snprintf` is not yet part of the ANSI C standard but is being considered for a revision of the standard, currently called C9X. Nevertheless, many vendors are providing it as part of the standard C library. We use `snprintf` throughout the text, providing our own version that just calls `sprintf` when it is not provided.

lib/px_ipc_name.c

```
 1 #include    "unpipc.h"

 2 char *
 3 px_ipc_name(const char *name)
 4 {
 5     char   *dir, *dst, *slash;

 6     if ( (dst = malloc(PATH_MAX)) == NULL)
 7         return (NULL);

 8         /* can override default directory with environment variable */
 9     if ( (dir = getenv("PX_IPC_NAME")) == NULL) {
10 #ifdef  POSIX_IPC_PREFIX
11         dir = POSIX_IPC_PREFIX; /* from "config.h" */
12 #else
13         dir = "/tmp/";          /* default */
14 #endif
15     }
16         /* dir must end in a slash */
17     slash = (dir[strlen(dir) - 1] == '/') ? "" : "/";
18     snprintf(dst, PATH_MAX, "%s%s%s", dir, slash, name);

19     return (dst);               /* caller can free() this pointer */
20 }
```

lib/px_ipc_name.c

Figure 2.2 Our px_ipc_name function.

2.3 Creating and Opening IPC Channels

The three functions that create or open an IPC object, mq_open, sem_open, and shm_open, all take a second argument named *oflag* that specifies how to open the requested object. This is similar to the second argument to the standard open function. The various constants that can be combined to form this argument are shown in Figure 2.3.

Description	mq_open	sem_open	shm_open
read-only	O_RDONLY		O_RDONLY
write-only	O_WRONLY		
read–write	O_RDWR		O_RDWR
create if it does not already exist	O_CREAT	O_CREAT	O_CREAT
exclusive create	O_EXCL	O_EXCL	O_EXCL
nonblocking mode	O_NONBLOCK		
truncate if it already exists			O_TRUNC

Figure 2.3 Various constants when opening or creating a Posix IPC object.

The first three rows specify how the object is being opened: read-only, write-only, or read–write. A message queue can be opened in any of the three modes, whereas none

of these three constants is specified for a semaphore (read and write access is required for any semaphore operation), and a shared memory object cannot be opened write-only.

The remaining O_*xxx* flags in Figure 2.3 are optional.

O_CREAT Create the message queue, semaphore, or shared memory object if it does not already exist. (Also see the O_EXCL flag, which is described shortly.)

When creating a new message queue, semaphore, or shared memory object at least one additional argument is required, called *mode*. This argument specifies the permission bits and is formed as the bitwise-OR of the constants shown in Figure 2.4.

Constant	Description
S_IRUSR	user read
S_IWUSR	user write
S_IRGRP	group read
S_IWGRP	group write
S_IROTH	other read
S_IWOTH	other write

Figure 2.4 *mode* constants when a new IPC object is created.

These constants are defined in the <sys/stat.h> header. The specified permission bits are modified by the *file mode creation mask* of the process, which can be set by calling the umask function (pp. 83–85 of APUE) or by using the shell's umask command.

As with a newly created file, when a new message queue, semaphore, or shared memory object is created, the user ID is set to the effective user ID of the process. The group ID of a semaphore or shared memory object is set to the effective group ID of the process or to a system default group ID. The group ID of a new message queue is set to the effective group ID of the process. (Pages 77–78 of APUE talk more about the user and group IDs.)

> This difference in the setting of the group ID between the three types of Posix IPC is strange. The group ID of a new file created by open is either the effective group ID of the process or the group ID of the directory in which the file is created, but the IPC functions cannot assume that a pathname in the filesystem is created for an IPC object.

O_EXCL If this flag and O_CREAT are both specified, then the function creates a new message queue, semaphore, or shared memory object only if it does not already exist. If it already exists, and if O_CREAT | O_EXCL is specified, an error of EEXIST is returned.

The check for the existence of the message queue, semaphore, or shared memory object and its creation (if it does not already exist) must be *atomic* with regard to other processes. We will see two similar flags for System V IPC in Section 3.4.

`O_NONBLOCK` This flag makes a message queue nonblocking with regard to a read on an empty queue or a write to a full queue. We talk about this more with the `mq_receive` and `mq_send` functions in Section 5.4.

`O_TRUNC` If an existing shared memory object is opened read–write, this flag specifies that the object be truncated to 0 length.

Figure 2.5 shows the actual logic flow for opening an IPC object. We describe what we mean by the test of the access permissions in Section 2.4. Another way of looking at Figure 2.5 is shown in Figure 2.6.

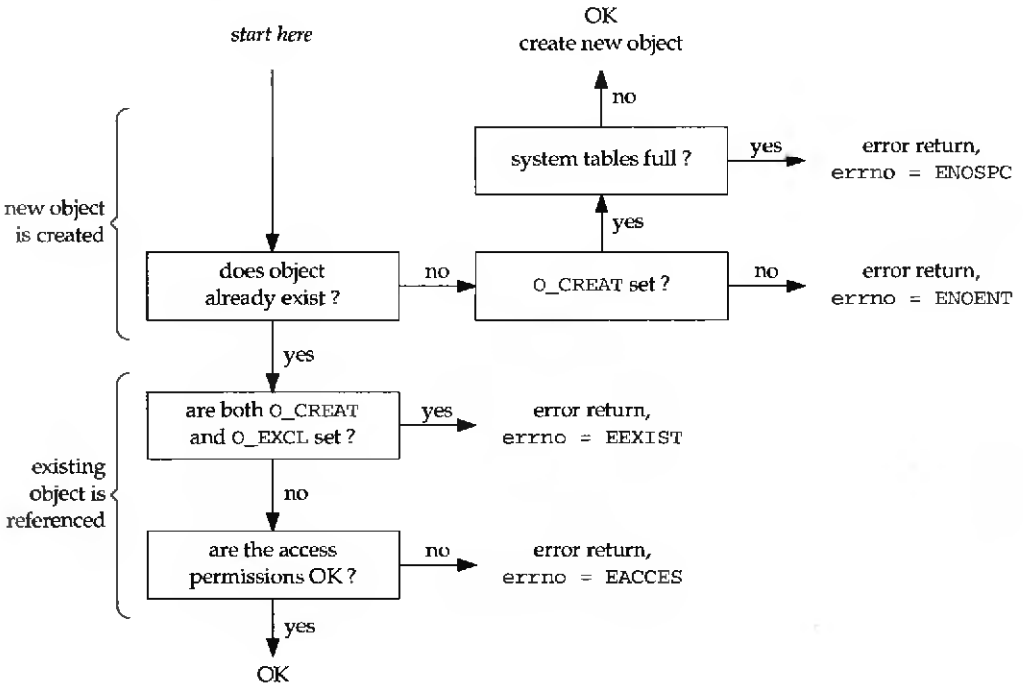

Figure 2.5 Logic for opening or creating an IPC object.

oflag argument	Object does not exist	Object already exists
no special flags	error, `errno = ENOENT`	OK, references existing object
`O_CREAT`	OK, creates new object	OK, references existing object
`O_CREAT | O_EXCL`	OK, creates new object	error, `errno = EEXIST`

Figure 2.6 Logic for creating or opening an IPC object.

Note that in the middle line of Figure 2.6, the O_CREAT flag without O_EXCL, we do not get an indication whether a new entry has been created or whether we are referencing an existing entry.

2.4 IPC Permissions

A new message queue, named semaphore, or shared memory object is created by mq_open, sem_open, or shm_open when the *oflag* argument contains the O_CREAT flag. As noted in Figure 2.4, permission bits are associated with each of these forms of IPC, similar to the permission bits associated with a Unix file.

When an existing message queue, semaphore, or shared memory object is opened by these same three functions (either O_CREAT is not specified, or O_CREAT is specified without O_EXCL and the object already exists), permission testing is performed based on

1. the permission bits assigned to the IPC object when it was created,
2. the type of access being requested (O_RDONLY, O_WRONLY, or O_RDWR), and
3. the effective user ID of the calling process, the effective group ID of the calling process, and the supplementary group IDs of the process (if supported).

The tests performed by most Unix kernels are as follows:

1. If the effective user ID of the process is 0 (the superuser), access is allowed.
2. If the effective user ID of the process equals the owner ID of the IPC object: if the appropriate user access permission bit is set, access is allowed, else access is denied.

   By *appropriate access permission bit*, we mean if the process is opening the IPC object for reading, the user-read bit must be on. If the process is opening the IPC object for writing, the user-write bit must be on.
3. If the effective group ID of the process or one of the supplementary group IDs of the process equals the group ID of the IPC object: if the appropriate group access permission bit is set, access is allowed, else permission is denied.
4. If the appropriate other access permission bit is set, access is allowed, else permission is denied.

These four steps are tried in sequence in the order listed. Therefore, if the process owns the IPC object (step 2), then access is granted or denied based only on the user access permissions—the group permissions are never considered. Similarly, if the process does not own the IPC object, but the process belongs to an appropriate group, then access is granted or denied based only on the group access permissions—the other permissions are not considered.

We note from Figure 2.3 that sem_open does not use the O_RDONLY, O_WRONLY, or O_RDWR flag. We note in Section 10.2, however, that some Unix implementations assume O_RDWR, since any use of a semaphore involves reading and writing the semaphore value.

2.5 Summary

The three types of Posix IPC—message queues, semaphores, and shared memory—are identified by pathnames. But these may or may not be real pathnames in the filesystem, and this discrepancy can be a portability problem. The solution that we employ throughout the text is to use our own px_ipc_name function.

When an IPC object is created or opened, we specify a set of flags that are similar to those for the open function. When a new IPC object is created, we must specify the permissions for the new object, using the same S_*xxx* constants that are used with open (Figure 2.4). When an existing IPC object is opened, the permission testing that is performed is the same as when an existing file is opened.

Exercises

2.1 In what way do the set-user-ID and set-group-ID bits (Section 4.4 of APUE) of a program that uses Posix IPC affect the permission testing described in Section 2.4?

2.2 When a program opens a Posix IPC object, how can it determine whether a new object was created or whether it is referencing an existing object?

3

System V IPC

3.1 Introduction

The three types of IPC,

- System V message queues (Chapter 6),
- System V semaphores (Chapter 11), and
- System V shared memory (Chapter 14)

are collectively referred to as "System V IPC." This term is commonly used for these three IPC facilities, acknowledging their heritage from System V Unix. They share many similarities in the functions that access them, and in the information that the kernel maintains on them. This chapter describes all these common properties.

A summary of their functions is shown in Figure 3.1.

	Message queues	Semaphores	Shared memory
Header	`<sys/msg.h>`	`<sys/sem.h>`	`<sys/shm.h>`
Function to create or open	`msgget`	`semget`	`shmget`
Function for control operations	`msgctl`	`semctl`	`shmctl`
Functions for IPC operations	`msgsnd` `msgrcv`	`semop`	`shmat` `shmdt`

Figure 3.1 Summary of System V IPC functions.

Information on the design and development of the System V IPC functions is hard to find. [Rochkind 1985] provides the following information: System V message queues, semaphores, and shared memory were developed in the late 1970s at a branch laboratory of Bell

Laboratories in Columbus, Ohio, for an internal version of Unix called (not surprisingly) "Columbus Unix" or just "CB Unix." This version of Unix was used for "Operation Support Systems," transaction processing systems that automated telephone company administration and recordkeeping. System V IPC was added to the commercial Unix system with System V around 1983.

3.2 `key_t` Keys and `ftok` Function

In Figure 1.4, the three types of System V IPC are noted as using `key_t` values for their names. The header `<sys/types.h>` defines the `key_t` datatype, as an integer, normally at least a 32-bit integer. These integer values are normally assigned by the `ftok` function.

The function `ftok` converts an existing pathname and an integer identifier into a `key_t` value (called an *IPC key*).

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

key_t ftok(const char *pathname, int id);
```

Returns: IPC key if OK, –1 on error

This function takes information derived from the *pathname* and the low-order 8 bits of *id*, and combines them into an integer IPC key.

This function assumes that for a given application using System V IPC, the server and clients all agree on a single *pathname* that has some meaning to the application. It could be the pathname of the server daemon, the pathname of a common data file used by the server, or some other pathname on the system. If the client and server need only a single IPC channel between them, an *id* of one, say, can be used. If multiple IPC channels are needed, say one from the client to the server and another from the server to the client, then one channel can use an *id* of one, and the other an *id* of two, for example. Once the *pathname* and *id* are agreed on by the client and server, then both can call the `ftok` function to convert these into the same IPC key.

Typical implementations of `ftok` call the `stat` function and then combine

1. information about the filesystem on which *pathname* resides (the `st_dev` member of the `stat` structure),
2. the file's i-node number within the filesystem (the `st_ino` member of the `stat` structure), and
3. the low-order 8 bits of the *id*.

The combination of these three values normally produces a 32-bit key. No guarantee exists that two different pathnames combined with the same *id* generate different keys, because the number of bits of information in the three items just listed (filesystem identifier, i-node, and *id*) can be greater than the number of bits in an integer. (See Exercise 3.5.)

The i-node number is never 0, so most implementations define IPC_PRIVATE (which we describe in Section 3.4) to be 0.

If the *pathname* does not exist, or is not accessible to the calling process, ftok returns –1. Be aware that the file whose *pathname* is used to generate the key must not be a file that is created and deleted by the server during its existence, since each time it is created, it can assume a new i-node number that can change the key returned by ftok to the next caller.

Example

The program in Figure 3.2 takes a pathname as a command-line argument, calls stat, calls ftok, and then prints the st_dev and st_ino members of the stat structure, and the resulting IPC key. These three values are printed in hexadecimal, so we can easily see how the IPC key is constructed from these two values and our *id* of 0x57.

svipc/ftok.c

```
 1 #include    "unpipc.h"

 2 int
 3 main(int argc, char **argv)
 4 {
 5     struct stat stat;

 6     if (argc != 2)
 7         err_quit("usage: ftok <pathname>");

 8     Stat(argv[1], &stat);
 9     printf("st_dev: %lx, st_ino: %lx, key: %x\n",
10            (u_long) stat.st_dev, (u_long) stat.st_ino,
11            Ftok(argv[1], 0x57));

12     exit(0);
13 }
```

svipc/ftok.c

Figure 3.2 Obtain and print filesystem information and resulting IPC key.

Executing this under Solaris 2.6 gives us the following:

```
solaris % ftok /etc/system
st_dev: 800018, st_ino: 4a1b, key: 57018a1b
solaris % ftok /usr/tmp
st_dev: 800015, st_ino: 10b78, key: 57015b78
solaris % ftok /home/rstevens/Mail.out
st_dev: 80001f, st_ino: 3b03, key: 5701fb03
```

Apparently, the *id* is in the upper 8 bits, the low-order 12 bits of st_dev in the next 12 bits, and the low-order 12 bits of st_ino in the low-order 12 bits.

Our purpose in showing this example is not to let us count on this combination of information to form the IPC key, but to let us see how one implementation combines the *pathname* and *id*. Other implementations may do this differently.

FreeBSD uses the lower 8 bits of the *id*, the lower 8 bits of st_dev, and the lower 16 bits of st_ino.

Note that the mapping done by `ftok` is one-way, since some bits from `st_dev` and `st_ino` are not used. That is, given a key, we cannot determine the pathname that was used to create the key.

3.3 `ipc_perm` Structure

The kernel maintains a structure of information for each IPC object, similar to the information it maintains for files.

```
struct ipc_perm {
  uid_t    uid;     /* owner's user id */
  gid_t    gid;     /* owner's group id */
  uid_t    cuid;    /* creator's user id */
  gid_t    cgid;    /* creator's group id */
  mode_t   mode;    /* read-write permissions */
  ulong_t  seq;     /* slot usage sequence number */
  key_t    key;     /* IPC key */
};
```

This structure, and other manifest constants for the System V IPC functions, are defined in the `<sys/ipc.h>` header. We talk about all the members of this structure in this chapter.

3.4 Creating and Opening IPC Channels

The three `getXXX` functions that create or open an IPC object (Figure 3.1) all take an IPC *key* value, whose type is `key_t`, and return an integer *identifier*. This identifier is *not* the same as the *id* argument to the `ftok` function, as we see shortly. An application has two choices for the *key* value that is the first argument to the three `getXXX` functions:

1. call `ftok`, passing it a *pathname* and *id*, or
2. specify a *key* of `IPC_PRIVATE`, which guarantees that a new, unique IPC object is created.

The sequence of steps is shown in Figure 3.3.

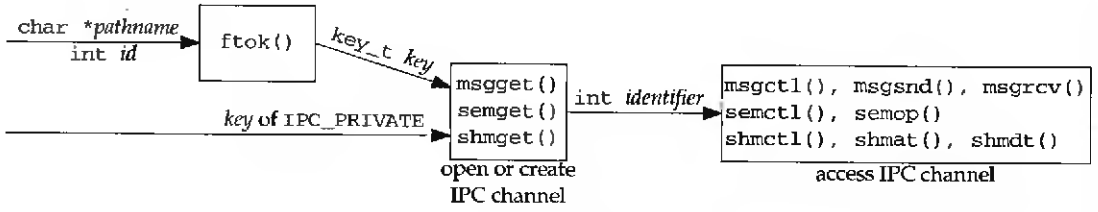

Figure 3.3 Generating IPC identifiers from IPC keys.

All three getXXX functions (Figure 3.1) also take an *oflag* argument that specifies the read–write permission bits (the mode member of the ipc_perm structure) for the IPC object, and whether a new IPC object is being created or an existing one is being referenced. The rules for whether a new IPC object is created or whether an existing one is referenced are as follows:

- Specifying a *key* of IPC_PRIVATE guarantees that a unique IPC object is created. No combinations of *pathname* and *id* exist that cause ftok to generate a *key* value of IPC_PRIVATE.
- Setting the IPC_CREAT bit of the *oflag* argument creates a new entry for the specified *key*, if it does not already exist. If an existing entry is found, that entry is returned.
- Setting both the IPC_CREAT and IPC_EXCL bits of the *oflag* argument creates a new entry for the specified *key*, only if the entry does not already exist. If an existing entry is found, an error of EEXIST is returned, since the IPC object already exists.

  The combination of IPC_CREAT and IPC_EXCL with regard to IPC objects is similar to the combination of O_CREAT and O_EXCL with regard to the open function.

  Setting the IPC_EXCL bit, without setting the IPC_CREAT bit, has no meaning.

The actual logic flow for opening an IPC object is shown in Figure 3.4. Figure 3.5 shows another way of looking at Figure 3.4.

Note that in the middle line of Figure 3.5, the IPC_CREAT flag without IPC_EXCL, we do not get an indication whether a new entry has been created or whether we are referencing an existing entry. In most applications, the server creates the IPC object and specifies either IPC_CREAT (if it does not care whether the object already exists) or IPC_CREAT | IPC_EXCL (if it needs to check whether the object already exists). The clients specify neither flag (assuming that the server has already created the object).

> The System V IPC functions define their own IPC_*xxx* constants, instead of using the O_CREAT and O_EXCL constants that are used by the standard open function along with the Posix IPC functions (Figure 2.3).
>
> Also note that the System V IPC functions combine their IPC_*xxx* constants with the permission bits (which we describe in the next section) into a single *oflag* argument. The open function along with the Posix IPC functions have one argument named *oflag* that specifies the various O_*xxx* flags, and another argument named *mode* that specifies the permission bits.

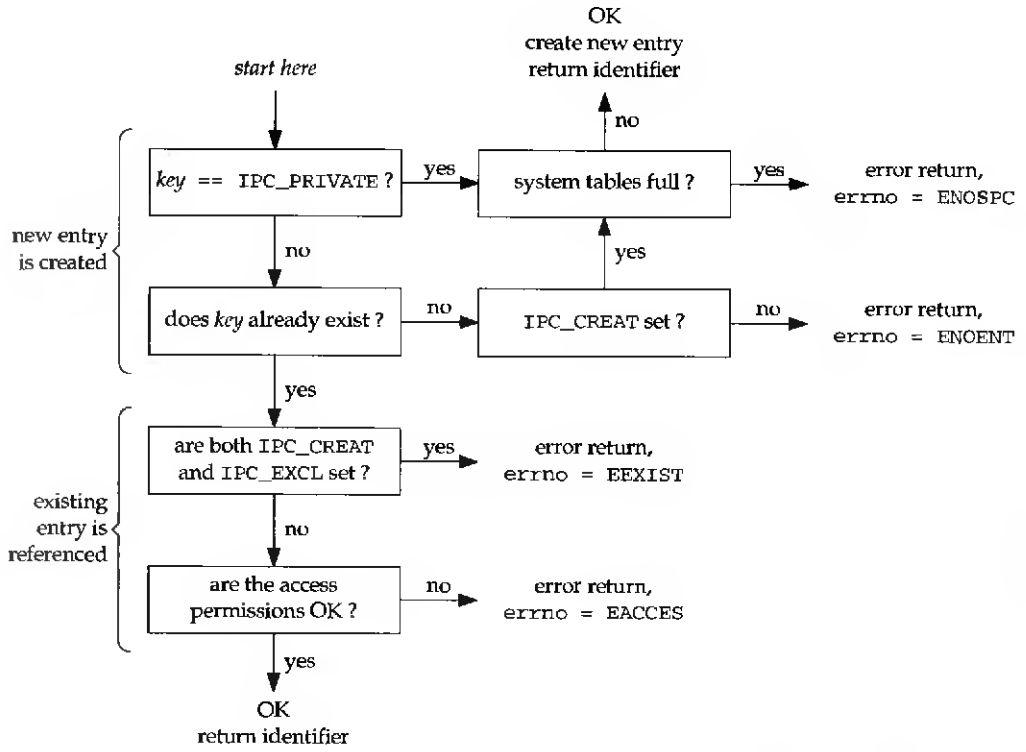

Figure 3.4 Logic for creating or opening an IPC object.

oflag argument	*key* does not exist	*key* already exists
no special flags	error, `errno = ENOENT`	OK, references existing object
`IPC_CREAT`	OK, creates new entry	OK, references existing object
`IPC_CREAT | IPC_EXCL`	OK, creates new entry	error, `errno = EEXIST`

Figure 3.5 Logic for creating or opening an IPC channel.

3.5 IPC Permissions

Whenever a new IPC object is created using one of the `getXXX` functions with the `IPC_CREAT` flag, the following information is saved in the `ipc_perm` structure (Section 3.3):

1. Some of the bits in the *oflag* argument initialize the `mode` member of the `ipc_perm` structure. Figure 3.6 shows the permission bits for the three different IPC mechanisms. (The notation >> 3 means the value is right shifted 3 bits.)

Numeric (octal)	Symbolic values			Description
	Message queue	Semaphore	Shared memory	
0400	MSG_R	SEM_R	SHM_R	read by user
0200	MSG_W	SEM_A	SHM_W	write by user
0040	MSG_R >> 3	SEM_R >> 3	SHM_R >> 3	read by group
0020	MSG_W >> 3	SEM_A >> 3	SHM_W >> 3	write by group
0004	MSG_R >> 6	SEM_R >> 6	SHM_R >> 6	read by others
0002	MSG_W >> 6	SEM_A >> 6	SHM_W >> 6	write by others

Figure 3.6 *mode* values for IPC read–write permissions.

2. The two members `cuid` and `cgid` are set to the effective user ID and effective group ID of the calling process, respectively. These two members are called the *creator IDs*.
3. The two members `uid` and `gid` in the `ipc_perm` structure are also set to the effective user ID and effective group ID of the calling process. These two members are called the *owner IDs*.

The creator IDs never change, although a process can change the owner IDs by calling the `ctlXXX` function for the IPC mechanism with a command of `IPC_SET`. The three `ctlXXX` functions also allow a process to change the permission bits of the `mode` member for the IPC object.

> Most implementations define the six constants `MSG_R`, `MSG_W`, `SEM_R`, `SEM_A`, `SHM_R`, and `SHM_W` shown in Figure 3.6 in the `<sys/msg.h>`, `<sys/sem.h>`, and `<sys/shm.h>` headers. But these are not required by Unix 98. The suffix `A` in `SEM_A` stands for "alter."
>
> The three `getXXX` functions do not use the normal Unix *file mode creation mask*. The permissions of the message queue, semaphore, or shared memory segment are set to exactly what the function specifies.
>
> Posix IPC does not let the creator of an IPC object change the owner. Nothing is like the `IPC_SET` command with Posix IPC. But if the Posix IPC name is stored in the filesystem, then the superuser can change the owner using the `chown` command.

Two levels of checking are done whenever an IPC object is accessed by any process, once when the IPC object is opened (the `getXXX` function) and then each time the IPC object is used:

1. Whenever a process establishes access to an existing IPC object with one of the `getXXX` functions, an initial check is made that the caller's *oflag* argument does not specify any access bits that are not in the `mode` member of the `ipc_perm` structure. This is the bottom box in Figure 3.4. For example, a server process can set the `mode` member for its input message queue so that the group-read and other-read permission bits are off. Any process that tries to specify an *oflag* argument that includes these bits gets an error return from the `msgget` function. But this test that is done by the `getXXX` functions is of little use. It implies that

the caller knows which permission category it falls into—user, group, or other. If the creator specifically turns off certain permission bits, and if the caller specifies these bits, the error is detected by the `getXXX` function. Any process, however, can totally bypass this check by just specifying an *oflag* argument of 0 if it knows that the IPC object already exists.

2. Every IPC operation does a permission test for the process using the operation. For example, every time a process tries to put a message onto a message queue with the `msgsnd` function, the following tests are performed in the order listed. As soon as a test grants access, no further tests are performed.
   a. The superuser is always granted access.
   b. If the effective user ID equals either the `uid` value or the `cuid` value for the IPC object, and if the appropriate access bit is on in the `mode` member for the IPC object, permission is granted. By "appropriate access bit," we mean the read-bit must be set if the caller wants to do a read operation on the IPC object (receiving a message from a message queue, for example), or the write-bit must be set for a write operation.
   c. If the effective group ID equals either the `gid` value or the `cgid` value for the IPC object, and if the appropriate access bit is on in the `mode` member for the IPC object, permission is granted.
   d. If none of the above tests are true, the appropriate "other" access bit must be on in the `mode` member for the IPC object, for permission to be allowed.

3.6 Identifier Reuse

The `ipc_perm` structure (Section 3.3) also contains a variable named `seq`, which is a slot usage sequence number. This is a counter that is maintained by the kernel for every potential IPC object in the system. Every time an IPC object is removed, the kernel increments the slot number, cycling it back to zero when it overflows.

> What we are describing in this section is the common SVR4 implementation. This implementation technique is not mandated by Unix 98.

This counter is needed for two reasons. First, consider the file descriptors maintained by the kernel for open files. They are small integers, but have meaning only within a single process—they are process-specific values. If we try to read from file descriptor 4, say, in a process, this approach works only if that process has a file open on this descriptor. It has no meaning whatsoever for a file that might be open on file descriptor 4 in some other unrelated process. System V IPC identifiers, however, are *systemwide* and not process-specific.

We obtain an IPC identifier (similar to a file descriptor) from one of the `get` functions: `msgget`, `semget`, and `shmget`. These identifiers are also integers, but their meaning applies to *all* processes. If two unrelated processes, a client and server, for example, use a single message queue, the message queue identifier returned by the

`msgget` function must be the same integer value in both processes in order to access the same message queue. This feature means that a rogue process could try to read a message from some other application's message queue by trying different small integer identifiers, hoping to find one that is currently in use that allows world read access. If the potential values for these identifiers were small integers (like file descriptors), then the probability of finding a valid identifier would be about 1 in 50 (assuming a maximum of about 50 descriptors per process).

To avoid this problem, the designers of these IPC facilities decided to increase the possible range of identifier values to include *all* integers, not just small integers. This increase is implemented by incrementing the identifier value that is returned to the calling process, by the number of IPC table entries, each time a table entry is reused. For example, if the system is configured for a maximum of 50 message queues, then the first time the first message queue table entry in the kernel is used, the identifier returned to the process is zero. After this message queue is removed and the first table entry is reused, the identifier returned is 50. The next time, the identifier is 100, and so on. Since `seq` is often implemented as an unsigned long integer (see the `ipc_perm` structure shown in Section 3.3), it cycles after the table entry has been used 85,899,346 times ($2^{32}/50$, assuming 32-bit long integers).

A second reason for incrementing the slot usage sequence number is to avoid short term reuse of the System V IPC identifiers. This helps ensure that a server that prematurely terminates and is then restarted, does not reuse an identifier.

As an example of this feature, the program in Figure 3.7 prints the first 10 identifier values returned by `msgget`.

svmsg/slot.c

```
 1 #include    "unpipc.h"

 2 int
 3 main(int argc, char **argv)
 4 {
 5     int     i, msqid;

 6     for (i = 0; i < 10; i++) {
 7         msqid = Msgget(IPC_PRIVATE, SVMSG_MODE | IPC_CREAT);
 8         printf("msqid = %d\n", msqid);

 9         Msgctl(msqid, IPC_RMID, NULL);
10     }
11     exit(0);
12 }
```

svmsg/slot.c

Figure 3.7 Print kernel assigned message queue identifier 10 times in a row.

Each time around the loop `msgget` creates a message queue, and then `msgctl` with a command of `IPC_RMID` deletes the queue. The constant `SVMSG_MODE` is defined in our `unpipc.h` header (Figure C.1) and specifies our default permission bits for a System V message queue. The program's output is

```
solaris % slot
msqid = 0
msqid = 50
```

```
msqid = 100
msqid = 150
msqid = 200
msqid = 250
msqid = 300
msqid = 350
msqid = 400
msqid = 450
```

If we run the program again, we see that this slot usage sequence number is a kernel variable that persists between processes.

```
solaris % slot
msqid = 500
msqid = 550
msqid = 600
msqid = 650
msqid = 700
msqid = 750
msqid = 800
msqid = 850
msqid = 900
msqid = 950
```

3.7 `ipcs` and `ipcrm` Programs

Since the three types of System V IPC are not identified by pathnames in the filesystem, we cannot look at them or remove them using the standard `ls` and `rm` programs. Instead, two special programs are provided by any system that implements these types of IPC: `ipcs`, which prints various pieces of information about the System V IPC features, and `ipcrm`, which removes a System V message queue, semaphore set, or shared memory segment. The former supports about a dozen command-line options, which affect which of the three types of IPC is reported and what information is output, and the latter supports six command-line options. Consult your manual pages for the details of all these options.

Since System V IPC is not part of Posix, these two commands are not standardized by Posix.2. But these two commands are part of Unix 98.

3.8 Kernel Limits

Most implementations of System V IPC have inherent kernel limits, such as the maximum number of message queues and the maximum number of semaphores per semaphore set. We show some typical values for these limits in Figures 6.25, 11.9, and 14.5. These limits are often derived from the original System V implementation.

Section 11.2 of [Bach 1986] and Chapter 8 of [Goodheart and Cox 1994] both describe the System V implementation of messages, semaphores, and shared memory. Some of these limits are described therein.

Unfortunately, these kernel limits are often too small, because many are derived from their original implementation on a small address system (the 16-bit PDP-11). Fortunately, most systems allow the administrator to change some or all of these default limits, but the required steps are different for each flavor of Unix. Most require rebooting the running kernel after changing the values. Unfortunately, some implementations still use 16-bit integers for some of the limits, providing a hard limit that cannot be exceeded.

Solaris 2.6, for example, has 20 of these limits. Their current values are printed by the `sysdef` command, although the values are printed as 0 if the corresponding kernel module has not been loaded (i.e., the facility has not yet been used). These may be changed by placing any of the following statements in the `/etc/system` file, which is read when the kernel bootstraps.

```
4et msgsys:msginfo_msgseg = value
set msgsys:msginfo_msgssz = value
set msgsys:msginfo_msgtql = value
set msgsys:msginfo_msgmap = value
set msgsys:msginfo_msgmax = value
set msgsys:msginfo_msgmnb = value
set msgsys:msginfo_msgmni = value

set semsys:seminfo_semopm = value
set semsys:seminfo_semume = value
set semsys:seminfo_semaem = value
set semsys:seminfo_semmap = value
set semsys:seminfo_semvmx = value
set semsys:seminfo_semmsl = value
set semsys:seminfo_semmni = value
set semsys:seminfo_semmns = value
set semsys:seminfo_semmnu = value

set shmsys:shminfo_shmmin = value
set shmsys:shminfo_shmseg = value
set shmsys:shminfo_shmmax = value
set shmsys:shminfo_shmmni = value
```

The last six characters of the name on the left-hand side of the equals sign are the variables listed in Figures 6.25, 11.9, and 14.5.

With Digital Unix 4.0B, the `sysconfig` program can query or modify many kernel parameters and limits. Here is the output of this program with the `-q` option, which queries the kernel for the current limits, for the `ipc` subsystem. We have omitted some lines unrelated to the System V IPC facility.

```
alpha % /sbin/sysconfig -q ipc
ipc:
msg-max = 8192
msg-mnb = 16384
msg-mni = 64
msg-tql = 40

shm-max = 4194304
shm-min = 1
shm-mni = 128
shm-seg = 32
```

```
sem-mni = 16
sem-msl = 25
sem-opm = 10
sem-ume = 10
sem-vmx = 32767
sem-aem = 16384
num-of-sems = 60
```

Different defaults for these parameters can be specified in the `/etc/sysconfigtab` file, which should be maintained using the `sysconfigdb` program. This file is read when the system bootstraps.

3.9 Summary

The first argument to the three functions, `msgget`, `semget`, and `shmget`, is a System V IPC key. These keys are normally created from a pathname using the system's `ftok` function. The key can also be the special value of `IPC_PRIVATE`. These three functions create a new IPC object or open an existing IPC object and return a System V IPC identifier: an integer that is then used to identify the object to the remaining IPC functions. These integers are not per-process identifiers (like descriptors) but are systemwide identifiers. These identifiers are also reused by the kernel after some time.

Associated with every System V IPC object is an `ipc_perm` structure that contains information such as the owner's user ID, group ID, read–write permissions, and so on. One difference between Posix IPC and System V IPC is that this information is always available for a System V IPC object (by calling one of the three `XXXctl` functions with an argument of `IPC_STAT`), but access to this information for a Posix IPC object depends on the implementation. If the Posix IPC object is stored in the filesystem, and if we know its name in the filesystem, then we can access this same information using the existing filesystem tools.

When a new System V IPC object is created or an existing object is opened, two flags are specified to the `getXXX` function (`IPC_CREAT` and `IPC_EXCL`), combined with nine permission bits.

Undoubtedly, the biggest problem in using System V IPC is that most implementations have artificial kernel limits on the sizes of these objects, and these limits date back to their original implementation. These mean that most applications that make heavy use of System V IPC require that the system administrator modify these kernel limits, and accomplishing this change differs for each flavor of Unix.

Exercises

3.1 Read about the `msgctl` function in Section 6.5 and modify the program in Figure 3.7 to print the `seq` member of the `ipc_perm` structure in addition to the assigned identifier.

3.2 Immediately after running the program in Figure 3.7, we run a program that creates two message queues. Assuming no other message queues have been used by any other applications since the kernel was booted, what two values are returned by the kernel as the message queue identifiers?

3.3 We noted in Section 3.5 that the System V IPC `getXXX` functions do not use the file mode creation mask. Write a test program that creates a FIFO (using the `mkfifo` function described in Section 4.6) and a System V message queue, specifying a permission of (octal) 666 for both. Compare the permissions of the resulting FIFO and message queue. Make certain your shell `umask` value is nonzero before running this program.

3.4 A server wants to create a unique message queue for its clients. Which is preferable—using some constant pathname (say the server's executable file) as an argument to `ftok`, or using `IPC_PRIVATE`?

3.5 Modify Figure 3.2 to print just the IPC key and pathname. Run the `find` program to print all the pathnames on your system and run the output through the program just modified. How many pathnames map to the same key?

3.6 If your system supports the `sar` program ("system activity reporter"), run the command

```
sar -m 5 6
```

This prints the number of message queue operations per second and the number of semaphore operations per second, sampled every 5 seconds, 6 times.

Part 2

Message Passing

4

Pipes and FIFOs

4.1 Introduction

Pipes are the original form of Unix IPC, dating back to the Third Edition of Unix in 1973 [Salus 1994]. Although useful for many operations, their fundamental limitation is that they have no name, and can therefore be used only by related processes. This was corrected in System III Unix (1982) with the addition of FIFOs, sometimes called *named pipes*. Both pipes and FIFOs are accessed using the normal `read` and `write` functions.

> Technically, pipes can be used between unrelated processes, given the ability to pass descriptors between processes (which we describe in Section 15.8 of this text as well as Section 14.7 of UNPv1). But for practical purposes, pipes are normally used between processes that have a common ancestor.

This chapter describes the creation and use of pipes and FIFOs. We use a simple file server example and also look at some client–server design issues: how many IPC channels are needed, iterative versus concurrent servers, and byte streams versus message interfaces.

4.2 A Simple Client–Server Example

The client–server example shown in Figure 4.1 is used throughout this chapter and Chapter 6 to illustrate pipes, FIFOs, and System V message queues.

The client reads a pathname from the standard input and writes it to the IPC channel. The server reads this pathname from the IPC channel and tries to open the file for reading. If the server can open the file, the server responds by reading the file and writing it to the IPC channel; otherwise, the server responds with an error message. The

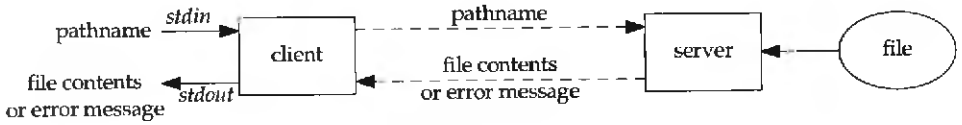

Figure 4.1 Client–server example.

client then reads from the IPC channel, writing what it receives to the standard output. If the file cannot be read by the server, the client reads an error message from the IPC channel. Otherwise, the client reads the contents of the file. The two dashed lines between the client and server in Figure 4.1 are the IPC channel.

4.3 Pipes

Pipes are provided with all flavors of Unix. A pipe is created by the `pipe` function and provides a one-way (unidirectional) flow of data.

```
#include <unistd.h>

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

Returns: 0 if OK, −1 on error

Two file descriptors are returned: *fd[0]*, which is open for reading, and *fd[1]*, which is open for writing.

> Some versions of Unix, notably SVR4, provide full-duplex pipes, in which case, both ends are available for reading and writing. Another way to create a full-duplex IPC channel is with the `socketpair` function, described in Section 14.3 of UNPv1, and this works on most current Unix systems. The most common use of pipes, however, is with the various shells, in which case, a half-duplex pipe is adequate.
>
> Posix.1 and Unix 98 require only half-duplex pipes, and we assume so in this chapter.

The `S_ISFIFO` macro can be used to determine if a descriptor or file is either a pipe or a FIFO. Its single argument is the `st_mode` member of the `stat` structure and the macro evaluates to true (nonzero) or false (0). For a pipe, this structure is filled in by the `fstat` function. For a FIFO, this structure is filled in by the `fstat`, `lstat`, or `stat` functions.

Figure 4.2 shows how a pipe looks in a single process.

Although a pipe is created by one process, it is rarely used within a single process. (We show an example of a pipe within a single process in Figure 5.14.) Pipes are typically used to communicate between two different processes (a parent and child) in the following way. First, a process (which will be the parent) creates a pipe and then `fork`s to create a copy of itself, as shown in Figure 4.3.

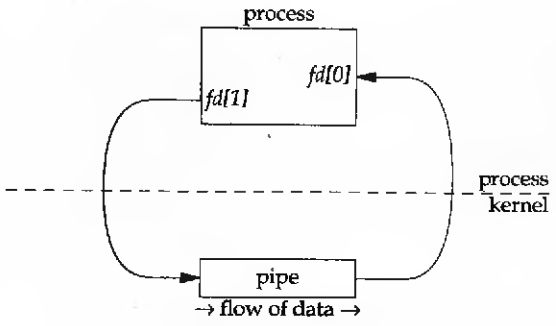

Figure 4.2 A pipe in a single process.

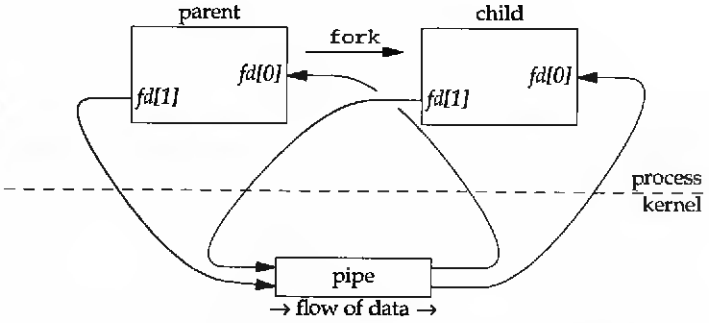

Figure 4.3 Pipe in a single process, immediately after `fork`.

Next, the parent process closes the read end of one pipe, and the child process closes the write end of that same pipe. This provides a one-way flow of data between the two processes, as shown in Figure 4.4.

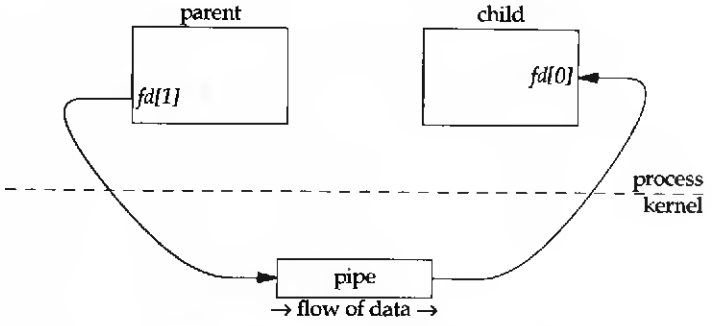

Figure 4.4 Pipe between two processes.

When we enter a command such as

```
who | sort | lp
```

to a Unix shell, the shell performs the steps described previously to create three

processes with two pipes between them. The shell also duplicates the read end of each pipe to standard input and the write end of each pipe to standard output. We show this pipeline in Figure 4.5.

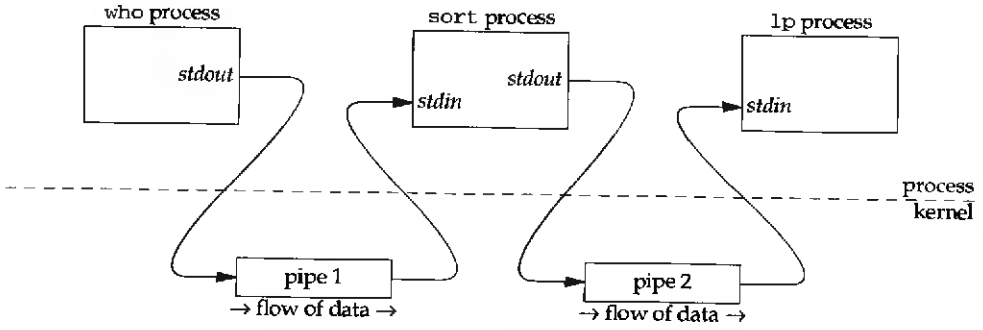

Figure 4.5 Pipes between three processes in a shell pipeline.

All the pipes shown so far have been half-duplex or unidirectional, providing a one-way flow of data only. When a two-way flow of data is desired, we must create two pipes and use one for each direction. The actual steps are as follows:

1. create pipe 1 (*fd1[0]* and *fd1[1]*), create pipe 2 (*fd2[0]* and *fd2[1]*),
2. `fork`,
3. parent closes read end of pipe 1 (*fd1[0]*),
4. parent closes write end of pipe 2 (*fd2[1]*),
5. child closes write end of pipe 1 (*fd1[1]*), and
6. child closes read end of pipe 2 (*fd2[0]*).

We show the code for these steps in Figure 4.8. This generates the pipe arrangement shown in Figure 4.6.

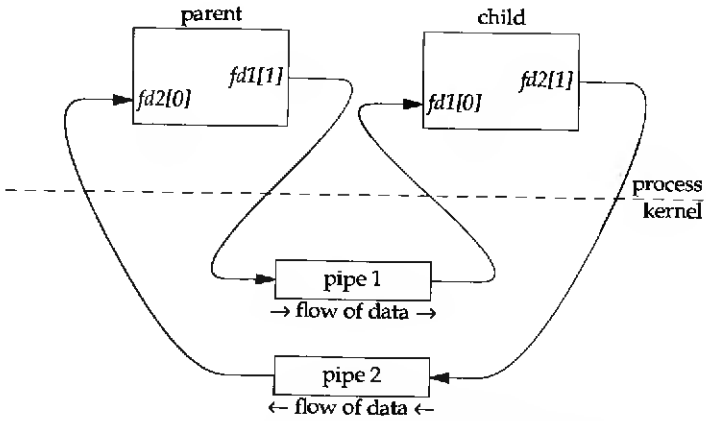

Figure 4.6 Two pipes to provide a bidirectional flow of data.

Example

Let us now implement the client–server example described in Section 4.2 using pipes. The `main` function creates two pipes and `fork`s a child. The client then runs in the parent process and the server runs in the child process. The first pipe is used to send the pathname from the client to the server, and the second pipe is used to send the contents of that file (or an error message) from the server to the client. This setup gives us the arrangement shown in Figure 4.7.

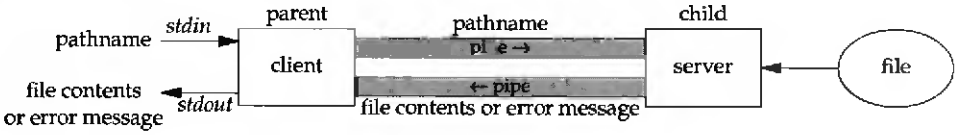

Figure 4.7 Implementation of Figure 4.1 using two pipes.

Realize that in this figure we show the two pipes connecting the two processes, but each pipe goes through the kernel, as shown previously in Figure 4.6. Therefore, each byte of data from the client to the server, and vice versa, crosses the user–kernel interface twice: once when written to the pipe, and again when read from the pipe.

Figure 4.8 shows our `main` function for this example.

pipe/mainpipe.c

```
 1 #include    "unpipc.h"

 2 void    client(int, int), server(int, int);

 3 int
 4 main(int argc, char **argv)
 5 {
 6     int     pipe1[2], pipe2[2];
 7     pid_t   childpid;

 8     Pipe(pipe1);                /* create two pipes */
 9     Pipe(pipe2);

10     if ( (childpid = Fork()) == 0) {      /* child */
11         Close(pipe1[1]);
12         Close(pipe2[0]);

13         server(pipe1[0], pipe2[1]);
14         exit(0);
15     }
16         /* parent */
17     Close(pipe1[0]);
18     Close(pipe2[1]);

19     client(pipe2[0], pipe1[1]);

20     Waitpid(childpid, NULL, 0); /* wait for child to terminate */
21     exit(0);
22 }
```

pipe/mainpipe.c

Figure 4.8 `main` function for client–server using two pipes.

Create pipes, `fork`

8–19 Two pipes are created and the six steps that we listed with Figure 4.6 are performed. The parent calls the `client` function (Figure 4.9) and the child calls the `server` function (Figure 4.10).

`waitpid` for child

20 The server (the child) terminates first, when it calls `exit` after writing the final data to the pipe. It then becomes a *zombie*: a process that has terminated, but whose parent is still running but has not yet waited for the child. When the child terminates, the kernel also generates a `SIGCHLD` signal for the parent, but the parent does not catch this signal, and the default action of this signal is to be ignored. Shortly thereafter, the parent's `client` function returns after reading the final data from the pipe. The parent then calls `waitpid` to fetch the termination status of the terminated child (the zombie). If the parent did not call `waitpid`, but just terminated, the child would be inherited by the `init` process, and another `SIGCHLD` signal would be sent to the `init` process, which would then fetch the termination status of the zombie.

The `client` function is shown in Figure 4.9.

pipe/client.c

```
 1 #include    "unpipc.h"

 2 void
 3 client(int readfd, int writefd)
 4 {
 5     size_t  len;
 6     ssize_t n;
 7     char    buff[MAXLINE];

 8         /* read pathname */
 9     Fgets(buff, MAXLINE, stdin);
10     len = strlen(buff);         /* fgets() guarantees null byte at end */
11     if (buff[len - 1] == '\n')
12         len--;                  /* delete newline from fgets() */

13         /* write pathname to IPC channel */
14     Write(writefd, buff, len);

15         /* read from IPC, write to standard output */
16     while ( (n = Read(readfd, buff, MAXLINE)) > 0)
17         Write(STDOUT_FILENO, buff, n);
18 }
```

pipe/client.c

Figure 4.9 `client` function for client–server using two pipes.

Read pathname from standard input

8–14 The pathname is read from standard input and written to the pipe, after deleting the newline that is stored by `fgets`.

Copy from pipe to standard output

15–17 The client then reads everything that the server writes to the pipe, writing it to

standard output. Normally this is the contents of the file, but if the specified pathname cannot be opened, what the server returns is an error message.

Figure 4.10 shows the `server` function.

pipe/server.c

```
 1 #include    "unpipc.h"

 2 void
 3 server(int readfd, int writefd)
 4 {
 5     int     fd;
 6     ssize_t n;
 7     char    buff[MAXLINE + 1];

 8         /* read pathname from IPC channel */
 9     if ( (n = Read(readfd, buff, MAXLINE)) == 0)
10         err_quit("end-of-file while reading pathname");
11     buff[n] = '\0';             /* null terminate pathname */

12     if ( (fd = open(buff, O_RDONLY)) < 0) {
13             /* error: must tell client */
14         snprintf(buff + n, sizeof(buff) - n, ": can't open, %s\n",
15                  strerror(errno));
16         n = strlen(buff);
17         Write(writefd, buff, n);

18     } else {
19             /* open succeeded: copy file to IPC channel */
20         while ( (n = Read(fd, buff, MAXLINE)) > 0)
21             Write(writefd, buff, n);
22         Close(fd);
23     }
24 }
```

pipe/server.c

Figure 4.10 `server` function for client–server using two pipes.

Read pathname from pipe

8–11 The pathname written by the client is read from the pipe and null terminated. Note that a `read` on a pipe returns as soon as some data is present; it need not wait for the requested number of bytes (`MAXLINE` in this example).

Open file, handle error

12–17 The file is opened for reading, and if an error occurs, an error message string is returned to the client across the pipe. We call the `strerror` function to return the error message string corresponding to `errno`. (Pages 690–691 of UNPv1 talk more about the `strerror` function.)

Copy file to pipe

18–23 If the open succeeds, the contents of the file are copied to the pipe.

We can see the output from the program when the pathname is OK, and when an error occurs.

```
solaris % mainpipe
/etc/inet/ntp.conf                          a file consisting of two lines
multicastclient 224.0.1.1
driftfile /etc/inet/ntp.drift
solaris % mainpipe
/etc/shadow                                 a file we cannot read
/etc/shadow: can't open, Permission denied
solaris % mainpipe
/no/such/file                               a nonexistent file
/no/such/file: can't open, No such file or directory
```

4.4 Full-Duplex Pipes

We mentioned in the previous section that some systems provide full-duplex pipes: SVR4's `pipe` function and the `socketpair` function provided by many kernels. But what exactly does a full-duplex pipe provide? First, we can think of a half-duplex pipe as shown in Figure 4.11, a modification of Figure 4.2, which omits the process.

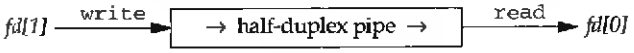

Figure 4.11 Half-duplex pipe.

A full-duplex pipe could be implemented as shown in Figure 4.12. This implies that only one buffer exists for the pipe and everything written to the pipe (on either descriptor) gets appended to the buffer and any read from the pipe (on either descriptor) just takes data from the front of the buffer.

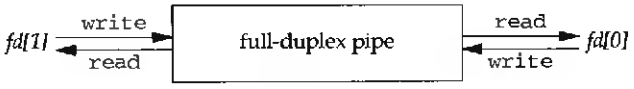

Figure 4.12 One possible (incorrect) implementation of a full-duplex pipe.

The problem with this implementation becomes apparent in a program such as Figure A.29. We want two-way communication but we need two independent data streams, one in each direction. Otherwise, when a process writes data to the full-duplex pipe and then turns around and issues a `read` on that pipe, it could read back what it just wrote.

Figure 4.13 shows the actual implementation of a full-duplex pipe.

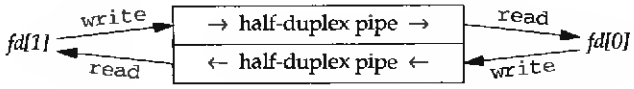

Figure 4.13 Actual implementation of a full-duplex pipe.

Here, the full-duplex pipe is constructed from two half-duplex pipes. Anything written

to *fd[1]* will be available for reading by *fd[0]*, and anything written to *fd[0]* will be available for reading by *fd[1]*.

The program in Figure 4.14 demonstrates that we can use a single full-duplex pipe for two-way communication.

pipe/fduplex.c

```
 1 #include    "unpipc.h"

 2 int
 3 main(int argc, char **argv)
 4 {
 5     int     fd[2], n;
 6     char    c;
 7     pid_t   childpid;

 8     Pipe(fd);                   /* assumes a full-duplex pipe (e.g., SVR4) */
 9     if ( (childpid = Fork()) == 0) {     /* child */
10         sleep(3);
11         if ( (n = Read(fd[0], &c, 1)) != 1)
12             err_quit("child: read returned %d", n);
13         printf("child read %c\n", c);
14         Write(fd[0], "c", 1);
15         exit(0);
16     }
17         /* parent */
18     Write(fd[1], "p", 1);
19     if ( (n = Read(fd[1], &c, 1)) != 1)
20         err_quit("parent: read returned %d", n);
21     printf("parent read %c\n", c);
22     exit(0);
23 }
```

pipe/fduplex.c

Figure 4.14 Test a full-duplex pipe for two-way communication.

We create a full-duplex pipe and `fork`. The parent writes the character `p` to the pipe, and then reads a character from the pipe. The child sleeps for 3 seconds, reads a character from the pipe, and then writes the character `c` to the pipe. The purpose of the sleep in the child is to allow the parent to call `read` before the child can call `read`, to see whether the parent reads back what it wrote.

If we run this program under Solaris 2.6, which provides full-duplex pipes, we observe the desired behavior.

```
solaris % fduplex
child read p
parent read c
```

The character `p` goes across the half-duplex pipe shown in the top of Figure 4.13, and the character `c` goes across the half-duplex pipe shown in the bottom of Figure 4.13. The parent does not read back what it wrote (the character `p`).

If we run this program under Digital Unix 4.0B, which by default provides half-duplex pipes (it also provides full-duplex pipes like SVR4, if different options are specified at compile time), we see the expected behavior of a half-duplex pipe.

```
alpha % fduplex
read error: Bad file number
alpha % child read p
write error: Bad file number
```

The parent writes the character p, which the child reads, but then the parent aborts when it tries to read from *fd[1]*, and the child aborts when it tries to write to *fd[0]* (recall Figure 4.11). The error returned by read is EBADF, which means that the descriptor is not open for reading. Similarly, write returns the same error if its descriptor is not open for writing.

4.5 popen and pclose Functions

As another example of pipes, the standard I/O library provides the popen function that creates a pipe and initiates another process that either reads from the pipe or writes to the pipe.

```
#include <stdio.h>

FILE *popen(const char *command, const char *type);
```

Returns: file pointer if OK, NULL on error

```
int pclose(FILE *stream);
```

Returns: termination status of shell or −1 on error

command is a shell command line. It is processed by the sh program (normally a Bourne shell), so the PATH environment variable is used to locate the *command*. A pipe is created between the calling process and the specified command. The value returned by popen is a standard I/O FILE pointer that is used for either input or output, depending on the character string *type*.

- If *type* is r, the calling process reads the standard output of the *command*.
- If *type* is w, the calling process writes to the standard input of the *command*.

The pclose function closes a standard I/O *stream* that was created by popen, waits for the command to terminate, and then returns the termination status of the shell.

Section 14.3 of APUE provides an implementation of popen and pclose.

Example

Figure 4.15 shows another solution to our client–server example using the popen function and the Unix cat program.

pipe/mainpopen.c

```
 1 #include    "unpipc.h"

 2 int
 3 main(int argc, char **argv)
 4 {
 5     size_t  n;
 6     char    buff[MAXLINE], command[MAXLINE];
 7     FILE   *fp;

 8         /* read pathname */
 9     Fgets(buff, MAXLINE, stdin);
10     n = strlen(buff);           /* fgets() guarantees null byte at end */
11     if (buff[n - 1] == '\n')
12         n--;                    /* delete newline from fgets() */

13     snprintf(command, sizeof(command), "cat %s", buff);
14     fp = Popen(command, "r");

15         /* copy from pipe to standard output */
16     while (Fgets(buff, MAXLINE, fp) != NULL)
17         Fputs(buff, stdout);

18     Pclose(fp);
19     exit(0);
20 }
```

pipe/mainpopen.c

Figure 4.15 Client-server using popen.

8-17 The pathname is read from standard input, as in Figure 4.9. A command is built and passed to popen. The output from either the shell or the `cat` program is copied to standard output.

One difference between this implementation and the implementation in Figure 4.8 is that now we are dependent on the error message generated by the system's `cat` program, which is often inadequate. For example, under Solaris 2.6, we get the following error when trying to read a file that we do not have permission to read:

```
solaris % cat /etc/shadow
cat: cannot open /etc/shadow
```

But under BSD/OS 3.1, we get a more descriptive error when trying to read a similar file:

```
bsdi % cat /etc/master.passwd
cat: /etc/master.passwd: cannot open [Permission denied]
```

Also realize that the call to popen succeeds in such a case, but `fgets` just returns an end-of-file the first time it is called. The `cat` program writes its error message to standard error, and popen does nothing special with it—only standard output is redirected to the pipe that it creates.

4.6 FIFOs

Pipes have no names, and their biggest disadvantage is that they can be used only between processes that have a parent process in common. Two unrelated processes cannot create a pipe between them and use it for IPC (ignoring descriptor passing).

FIFO stands for *first in, first out*, and a Unix FIFO is similar to a pipe. It is a one-way (half-duplex) flow of data. But unlike pipes, a FIFO has a pathname associated with it, allowing unrelated processes to access a single FIFO. FIFOs are also called *named pipes*.

A FIFO is created by the `mkfifo` function.

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

int mkfifo(const char *pathname, mode_t mode);
```

Returns: 0 if OK, –1 on error

The *pathname* is a normal Unix pathname, and this is the name of the FIFO.

The *mode* argument specifies the file permission bits, similar to the second argument to `open`. Figure 2.4 shows the six constants from the `<sys/stat.h>` header used to specify these bits for a FIFO.

The `mkfifo` function implies `O_CREAT | O_EXCL`. That is, it creates a new FIFO or returns an error of `EEXIST` if the named FIFO already exists. If the creation of a new FIFO is not desired, call `open` instead of `mkfifo`. To open an existing FIFO or create a new FIFO if it does not already exist, call `mkfifo`, check for an error of `EEXIST`, and if this occurs, call `open` instead.

The `mkfifo` command also creates a FIFO. This can be used from shell scripts or from the command line.

Once a FIFO is created, it must be opened for reading or writing, using either the `open` function, or one of the standard I/O open functions such as `fopen`. A FIFO must be opened either read-only or write-only. It must not be opened for read–write, because a FIFO is half-duplex.

A `write` to a pipe or FIFO always appends the data, and a `read` always returns what is at the beginning of the pipe or FIFO. If `lseek` is called for a pipe or FIFO, the error `ESPIPE` is returned.

Example

We now redo our client–server from Figure 4.8 to use two FIFOs instead of two pipes. Our `client` and `server` functions remain the same; all that changes is the `main` function, which we show in Figure 4.16.

pipe/mainfifo.c

```
1 #include    "unpipc.h"

2 #define FIFO1   "/tmp/fifo.1"
3 #define FIFO2   "/tmp/fifo.2"

4 void    client(int, int), server(int, int);
```

```
 5 int
 6 main(int argc, char **argv)
 7 {
 8     int     readfd, writefd;
 9     pid_t   childpid;

10         /* create two FIFOs; OK if they already exist */
11     if ((mkfifo(FIFO1, FILE_MODE) < 0) && (errno != EEXIST))
12         err_sys("can't create %s", FIFO1);
13     if ((mkfifo(FIFO2, FILE_MODE) < 0) && (errno != EEXIST)) {
14         unlink(FIFO1);
15         err_sys("can't create %s", FIFO2);
16     }
17     if ( (childpid = Fork()) == 0) {      /* child */
18         readfd = Open(FIFO1, O_RDONLY, 0);
19         writefd = Open(FIFO2, O_WRONLY, 0);

20         server(readfd, writefd);
21         exit(0);
22     }
23         /* parent */
24     writefd = Open(FIFO1, O_WRONLY, 0);
25     readfd = Open(FIFO2, O_RDONLY, 0);

26     client(readfd, writefd);

27     Waitpid(childpid, NULL, 0); /* wait for child to terminate */

28     Close(readfd);
29     Close(writefd);

30     Unlink(FIFO1);
31     Unlink(FIFO2);
32     exit(0);
33 }
```

pipe/mainfifo.c

Figure 4.16 `main` function for our client–server that uses two FIFOs.

Create two FIFOs

10–16 Two FIFOs are created in the `/tmp` filesystem. If the FIFOs already exist, that is OK. The `FILE_MODE` constant is defined in our `unpipc.h` header (Figure C.1) as

```
#define  FILE_MODE  (S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)
                    /* default permissions for new files */
```

This allows user-read, user-write, group-read, and other-read. These permission bits are modified by the *file mode creation mask* of the process.

fork

17–27 We call `fork`, the child calls our `server` function (Figure 4.10), and the parent calls our `client` function (Figure 4.9). Before executing these calls, the parent opens the first FIFO for writing and the second FIFO for reading, and the child opens the first FIFO for reading and the second FIFO for writing. This is similar to our pipe example, and Figure 4.17 shows this arrangement.

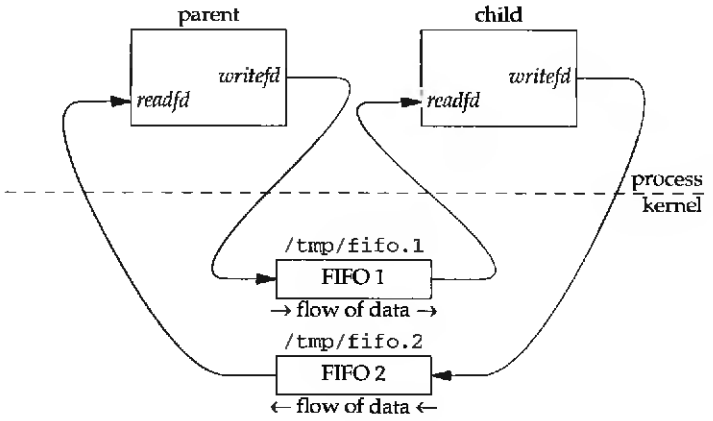

Figure 4.17 Client–server example using two FIFOs.

The changes from our pipe example to this FIFO example are as follows:

- To create and open a pipe requires one call to `pipe`. To create and open a FIFO requires one call to `mkfifo` followed by a call to `open`.
- A pipe automatically disappears on its last close. A FIFO's name is deleted from the filesystem only by calling `unlink`.

The benefit in the extra calls required for the FIFO is that a FIFO has a name in the filesystem allowing one process to create a FIFO and another unrelated process to open the FIFO. This is not possible with a pipe.

Subtle problems can occur with programs that do not use FIFOs correctly. Consider Figure 4.16: if we swap the order of the two calls to `open` in the parent, the program does not work. The reason is that the open of a FIFO for reading blocks if no process currently has the FIFO open for writing. If we swap the order of these two `open`s in the parent, both the parent and the child are opening a FIFO for reading when no process has the FIFO open for writing, so both block. This is called a *deadlock*. We discuss this scenario in the next section.

Example: Unrelated Client and Server

In Figure 4.16, the client and server are still related processes. But we can redo this example with the client and server unrelated. Figure 4.18 shows the server program. This program is nearly identical to the server portion of Figure 4.16.

The header `fifo.h` is shown in Figure 4.19 and provides the definitions of the two FIFO names, which both the client and server must know.

Figure 4.20 shows the client program, which is nearly identical to the client portion of Figure 4.16. Notice that the client, not the server, deletes the FIFOs when done, because the client performs the last operation on the FIFOs.

pipe/server_main.c

```
1 #include    "fifo.h"
2 void    server(int, int);
3 int
4 main(int argc, char **argv)
5 {
6     int     readfd, writefd;
7         /* create two FIFOs; OK if they already exist */
8     if ((mkfifo(FIFO1, FILE_MODE) < 0) && (errno != EEXIST))
9         err_sys("can't create %s", FIFO1);
10    if ((mkfifo(FIFO2, FILE_MODE) < 0) && (errno != EEXIST)) {
11        unlink(FIFO1);
12        err_sys("can't create %s", FIFO2);
13    }
14    readfd = Open(FIFO1, O_RDONLY, 0);
15    writefd = Open(FIFO2, O_WRONLY, 0);
16    server(readfd, writefd);
17    exit(0);
18 }
```

pipe/server_main.c

Figure 4.18 Stand-alone server `main` function.

pipe/fifo.h.c

```
1 #include    "unpipc.h"
2 #define FIFO1   "/tmp/fifo.1"
3 #define FIFO2   "/tmp/fifo.2"
```

pipe/fifo.h.c

Figure 4.19 `fifo.h` header that both the client and server include.

pipe/client_main.c

```
1 #include    "fifo.h"
2 void    client(int, int);
3 int
4 main(int argc, char **argv)
5 {
6     int     readfd, writefd;
7     writefd = Open(FIFO1, O_WRONLY, 0);
8     readfd = Open(FIFO2, O_RDONLY, 0);
9     client(readfd, writefd);
10    Close(readfd);
11    Close(writefd);
12    Unlink(FIFO1);
13    Unlink(FIFO2);
14    exit(0);
15 }
```

pipe/client_main.c

Figure 4.20 Stand-alone client `main` function.

In the case of a pipe or FIFO, where the kernel keeps a reference count of the number of open descriptors that refer to the pipe or FIFO, either the client or server could call `unlink` without a problem. Even though this function removes the pathname from the filesystem, this does not affect open descriptors that had previously opened the pathname. But for other forms of IPC, such as System V message queues, no counter exists and if the server were to delete the queue after writing its final message to the queue, the queue could be gone when the client tries to read the final message.

To run this client and server, start the server in the background

```
% server_fifo &
```

and then start the client. Alternately, we could start only the client and have it invoke the server by calling `fork` and then `exec`. The client could also pass the names of the two FIFOs to the server as command-line arguments through the `exec` function, instead of coding them into a header. But this scenario would make the server a child of the client, in which case, a pipe could just as easily be used.

4.7 Additional Properties of Pipes and FIFOs

We need to describe in more detail some properties of pipes and FIFOs with regard to their opening, reading, and writing. First, a descriptor can be set nonblocking in two ways.

1. The `O_NONBLOCK` flag can be specified when `open` is called. For example, the first call to `open` in Figure 4.20 could be

```
writefd = Open(FIFO1, O_WRONLY | O_NONBLOCK, 0);
```

2. If a descriptor is already open, `fcntl` can be called to enable the `O_NONBLOCK` flag. This technique must be used with a pipe, since `open` is not called for a pipe, and no way exists to specify the `O_NONBLOCK` flag in the call to `pipe`. When using `fcntl`, we first fetch the current file status flags with the `F_GETFL` command, bitwise-OR the `O_NONBLOCK` flag, and then store the file status flags with the `F_SETFL` command:

```
int    flags;

if ( (flags = fcntl(fd, F_GETFL, 0)) < 0)
    err_sys("F_GETFL error");
flags |= O_NONBLOCK;
if (fcntl(fd, F_SETFL, flags) < 0)
    err_sys("F_SETFL error");
```

Beware of code that you may encounter that simply sets the desired flag, because this also clears all the other possible file status flags:

```
    /* wrong way to set nonblocking */
if (fcntl(fd, F_SETFL, O_NONBLOCK) < 0)
    err_sys("F_SETFL error");
```

Figure 4.21 shows the effect of the nonblocking flag for the opening of a FIFO and for the reading of data from an empty pipe or from an empty FIFO.

Current operation	Existing opens of pipe or FIFO	Return	
		Blocking (default)	O_NONBLOCK set
open FIFO read-only	FIFO open for writing	returns OK	returns OK
	FIFO not open for writing	blocks until FIFO is opened for writing	returns OK
open FIFO write-only	FIFO open for reading	returns OK	returns OK
	FIFO not open for reading	blocks until FIFO is opened for reading	returns an error of ENXIO
read empty pipe or empty FIFO	pipe or FIFO open for writing	blocks until data is in the pipe or FIFO, or until the pipe or FIFO is no longer open for writing	returns an error of EAGAIN
	pipe or FIFO not open for writing	read returns 0 (end-of-file)	read returns 0 (end-of-file)
write to pipe or FIFO	pipe or FIFO open for reading	(see text)	(see text)
	pipe or FIFO not open for reading	SIGPIPE generated for thread	SIGPIPE generated for thread

Figure 4.21 Effect of O_NONBLOCK flag on pipes and FIFOs.

Note a few additional rules regarding the reading and writing of a pipe or FIFO.

- If we ask to read more data than is currently available in the pipe or FIFO, only the available data is returned. We must be prepared to handle a return value from read that is less than the requested amount.
- If the number of bytes to write is less than or equal to PIPE_BUF (a Posix limit that we say more about in Section 4.11), the write is guaranteed to be *atomic*. This means that if two processes each write to the same pipe or FIFO at about the same time, either all the data from the first process is written, followed by all the data from the second process, or vice versa. The system does not intermix the data from the two processes. If, however, the number of bytes to write is greater than PIPE_BUF, there is no guarantee that the write operation is atomic.

  Posix.1 requires that PIPE_BUF be at least 512 bytes. Commonly encountered values range from 1024 for BSD/OS 3.1 to 5120 for Solaris 2.6. We show a program in Section 4.11 that prints this value.

- The setting of the O_NONBLOCK flag has no effect on the atomicity of writes to a pipe or FIFO—atomicity is determined solely by whether the requested number of bytes is less than or equal to PIPE_BUF. But when a pipe or FIFO is set nonblocking, the return value from write depends on the number of bytes to write

and the amount of space currently available in the pipe or FIFO. If the number of bytes to `write` is less than or equal to `PIPE_BUF`:

a. If there is room in the pipe or FIFO for the requested number of bytes, all the bytes are transferred.

b. If there is not enough room in the pipe or FIFO for the requested number of bytes, return is made immediately with an error of `EAGAIN`. Since the `O_NONBLOCK` flag is set, the process does not want to be put to sleep. But the kernel cannot accept part of the data and still guarantee an atomic `write`, so the kernel must return an error and tell the process to try again later.

If the number of bytes to `write` is greater than `PIPE_BUF`:

a. If there is room for at least 1 byte in the pipe or FIFO, the kernel transfers whatever the pipe or FIFO can hold, and that is the return value from `write`.

b. If the pipe or FIFO is full, return is made immediately with an error of `EAGAIN`.

- If we `write` to a pipe or FIFO that is not open for reading, the `SIGPIPE` signal is generated:

a. If the process does not catch or ignore `SIGPIPE`, the default action of terminating the process is taken.

b. If the process ignores the `SIGPIPE` signal, or if it catches the signal and returns from its signal handler, then `write` returns an error of `EPIPE`.

> `SIGPIPE` is considered a synchronous signal, that is, a signal attributable to one specific thread, the one that called `write`. But the easiest way to handle this signal is to ignore it (set its disposition to `SIG_IGN`) and let `write` return an error of `EPIPE`. An application should always detect an error return from `write`, but detecting the termination of a process by `SIGPIPE` is harder. If the signal is not caught, we must look at the termination status of the process from the shell to determine that the process was killed by a signal, and which signal. Section 5.13 of UNPv1 talks more about `SIGPIPE`.

4.8 One Server, Multiple Clients

The real advantage of a FIFO is when the server is a long-running process (e.g., a daemon, as described in Chapter 12 of UNPv1) that is unrelated to the client. The daemon creates a FIFO with a well-known pathname, opens the FIFO for reading, and the client then starts at some later time, opens the FIFO for writing, and sends its commands or whatever to the daemon through the FIFO. One-way communication of this form (client to server) is easy with a FIFO, but it becomes harder if the daemon needs to send something back to the client. Figure 4.22 shows the technique that we use with our example.

The server creates a FIFO with a well-known pathname, `/tmp/fifo.serv` in this example. The server will read client requests from this FIFO. Each client creates its own FIFO when it starts, with a pathname containing its process ID. Each client writes its

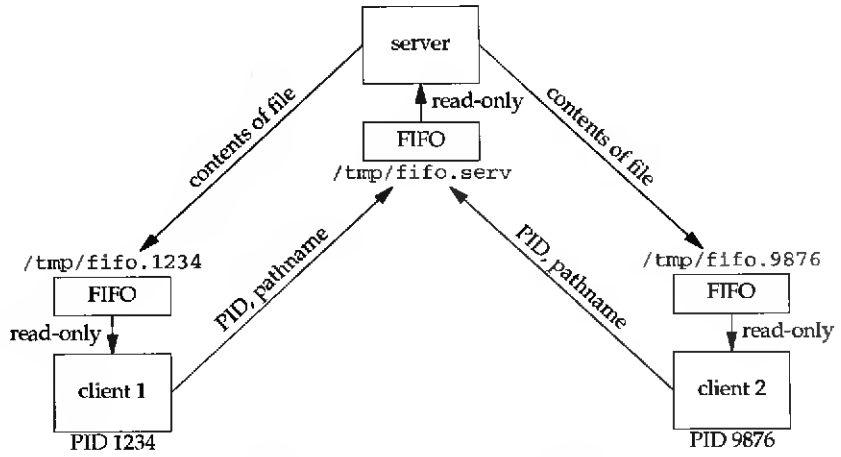

Figure 4.22 One server, multiple clients.

request to the server's well-known FIFO, and the request contains the client process ID along with the pathname of the file that the client wants the server to open and send to the client.

Figure 4.23 shows the server program.

Create well-known FIFO and open for read-only and write-only

10–15 The server's well-known FIFO is created, and it is OK if it already exists. We then open the FIFO twice, once read-only and once write-only. The `readfifo` descriptor is used to read each client request that arrives at the FIFO, but the `dummyfd` descriptor is never used. The reason for opening the FIFO for writing can be seen in Figure 4.21. If we do not open the FIFO for writing, then each time a client terminates, the FIFO becomes empty and the server's `read` returns 0 to indicate an end-of-file. We would then have to `close` the FIFO and call `open` again with the `O_RDONLY` flag, and this will block until the next client request arrives. But if we always have a descriptor for the FIFO that was opened for writing, `read` will never return 0 to indicate an end-of-file when no clients exist. Instead, our server will just block in the call to `read`, waiting for the next client request. This trick therefore simplifies our server code and reduces the number of calls to `open` for its well-known FIFO.

When the server starts, the first `open` (with the `O_RDONLY` flag) blocks until the first client opens the server's FIFO write-only (recall Figure 4.21). The second `open` (with the `O_WRONLY` flag) then returns immediately, because the FIFO is already open for reading.

Read client request

16 Each client request is a single line consisting of the process ID, one space, and then the pathname. We read this line with our `readline` function (which we show on p. 79 of UNPv1).

fifocliserv/mainserver.c

```
 1 #include    "fifo.h"
 2 void    server(int, int);
 3 int
 4 main(int argc, char **argv)
 5 {
 6     int     readfifo, writefifo, dummyfd, fd;
 7     char   *ptr, buff[MAXLINE], fifoname[MAXLINE];
 8     pid_t   pid;
 9     ssize_t n;
10         /* create server's well-known FIFO; OK if already exists */
11     if ((mkfifo(SERV_FIFO, FILE_MODE) < 0) && (errno != EEXIST))
12         err_sys("can't create %s", SERV_FIFO);
13         /* open server's well-known FIFO for reading and writing */
14     readfifo = Open(SERV_FIFO, O_RDONLY, 0);
15     dummyfd = Open(SERV_FIFO, O_WRONLY, 0);     /* never used */
16     while ( (n = Readline(readfifo, buff, MAXLINE)) > 0) {
17         if (buff[n - 1] == '\n')
18             n--;                /* delete newline from readline() */
19         buff[n] = '\0';         /* null terminate pathname */
20         if ( (ptr = strchr(buff, ' ')) == NULL) {
21             err_msg("bogus request: %s", buff);
22             continue;
23         }
24         *ptr++ = 0;             /* null terminate PID, ptr = pathname */
25         pid = atol(buff);
26         snprintf(fifoname, sizeof(fifoname), "/tmp/fifo.%ld", (long) pid);
27         if ( (writefifo = open(fifoname, O_WRONLY, 0)) < 0) {
28             err_msg("cannot open: %s", fifoname);
29             continue;
30         }
31         if ( (fd = open(ptr, O_RDONLY)) < 0) {
32                 /* error: must tell client */
33             snprintf(buff + n, sizeof(buff) - n, ": can't open, %s\n",
34                     strerror(errno));
35             n = strlen(ptr);
36             Write(writefifo, ptr, n);
37             Close(writefifo);
38         } else {
39                 /* open succeeded: copy file to FIFO */
40             while ( (n = Read(fd, buff, MAXLINE)) > 0)
41                 Write(writefifo, buff, n);
42             Close(fd);
43             Close(writefifo);
44         }
45     }
46 }
```

fifocliserv/mainserver.c

Figure 4.23 FIFO server that handles multiple clients.

Parse client's request

17–26 The newline that is normally returned by `readline` is deleted. This newline is missing only if the buffer was filled before the newline was encountered, or if the final line of input was not terminated by a newline. The `strchr` function returns a pointer to the first blank in the line, and `ptr` is incremented to point to the first character of the pathname that follows. The pathname of the client's FIFO is constructed from the process ID, and the FIFO is opened for write-only by the server.

Open file for client, send file to client's FIFO

27–44 The remainder of the server is similar to our `server` function from Figure 4.10. The file is opened and if this fails, an error message is returned to the client across the FIFO. If the `open` succeeds, the file is copied to the client's FIFO. When done, we must `close` the server's end of the client's FIFO, which causes the client's `read` to return 0 (end-of-file). The server does not delete the client's FIFO; the client must do so after it reads the end-of-file from the server.

We show the client program in Figure 4.24.

Create FIFO

10–14 The client's FIFO is created with the process ID as the final part of the pathname.

Build client request line

15–21 The client's request consists of its process ID, one blank, the pathname for the server to send to the client, and a newline. This line is built in the array `buff`, reading the pathname from the standard input.

Open server's FIFO and write request

22–24 The server's FIFO is opened and the request is written to the FIFO. If this client is the first to open this FIFO since the server was started, then this `open` unblocks the server from its call to `open` (with the `O_RDONLY` flag).

Read file contents or error message from server

25–31 The server's reply is read from the FIFO and written to standard output. The client's FIFO is then closed and deleted.

We can start our server in one window and run the client in another window, and it works as expected. We show only the client interaction.

```
solaris % mainclient
/etc/shadow                              a file we cannot read
/etc/shadow: can't open, Permission denied
solaris % mainclient
/etc/inet/ntp.conf                       a 2-line file
multicastclient 224.0.1.1
driftfile /etc/inet/ntp.drift
```

We can also interact with the server from the shell, because FIFOs have names in the filesystem.

fifocliserv/mainclient.c

```
1 #include    "fifo.h"
2 int
3 main(int argc, char **argv)
4 {
5     int     readfifo, writefifo;
6     size_t  len;
7     ssize_t n;
8     char    *ptr, fifoname[MAXLINE], buff[MAXLINE];
9     pid_t   pid;
10        /* create FIFO with our PID as part of name */
11    pid = getpid();
12    snprintf(fifoname, sizeof(fifoname), "/tmp/fifo.%ld", (long) pid);
13    if ((mkfifo(fifoname, FILE_MODE) < 0) && (errno != EEXIST))
14        err_sys("can't create %s", fifoname);
15        /* start buffer with pid and a blank */
16    snprintf(buff, sizeof(buff), "%ld ", (long) pid);
17    len = strlen(buff);
18    ptr = buff + len;
19        /* read pathname */
20    Fgets(ptr, MAXLINE - len, stdin);
21    len = strlen(buff);         /* fgets() guarantees null byte at end */
22        /* open FIFO to server and write PID and pathname to FIFO */
23    writefifo = Open(SERV_FIFO, O_WRONLY, 0);
24    Write(writefifo, buff, len);
25        /* now open our FIFO; blocks until server opens for writing */
26    readfifo = Open(fifoname, O_RDONLY, 0);
27        /* read from IPC, write to standard output */
28    while ( (n = Read(readfifo, buff, MAXLINE)) > 0)
29        Write(STDOUT_FILENO, buff, n);
30    Close(readfifo);
31    Unlink(fifoname);
32    exit(0);
33 }
```

fifocliserv/mainclient.c

Figure 4.24 FIFO client that works with the server in Figure 4.23.

```
solaris % Pid=$$                                    process ID of this shell
solaris % mkfifo /tmp/fifo.$Pid                     make the client's FIFO
solaris % echo "$Pid /etc/inet/ntp.conf" > /tmp/fifo.serv
solaris % cat < /tmp/fifo.$Pid                      and read server's reply
multicastclient 224.0.1.1
driftfile /etc/inet/ntp.drift
solaris % rm /tmp/fifo.$Pid
```

We send our process ID and pathname to the server with one shell command (`echo`) and read the server's reply with another (`cat`). Any amount of time can occur between these two commands. Therefore, the server appears to write the file to the FIFO, and the client later executes `cat` to read the data from the FIFO, which might make us think

that the data remains in the FIFO somehow, even when no process has the FIFO open. This is not what is happening. Indeed, the rule is that when the final `close` of a pipe or FIFO occurs, any remaining data in the pipe or FIFO is discarded. What is happening in our shell example is that after the server reads the request line from the client, the server blocks in its call to `open` on the client's FIFO, because the client (our shell) has not yet opened the FIFO for reading (recall Figure 4.21). Only when we execute `cat` sometime later, which opens the client FIFO for reading, does the server's call to `open` for this FIFO return. This timing also leads to a *denial-of-service* attack, which we discuss in the next section.

Using the shell also allows simple testing of the server's error handling. We can easily send a line to the server without a process ID, and we can also send a line to the server specifying a process ID that does not correspond to a FIFO in the `/tmp` directory. For example, if we invoke the client and enter the following lines

```
solaris % cat > /tmp/fifo.serv
/no/process/id
999999 /invalid/process/id
```

then the server's output (in another window) is

```
solaris % server
bogus request: /no/process/id
cannot open: /tmp/fifo.999999
```

Atomicity of FIFO `writes`

Our simple client–server also lets us see why the atomicity property of `writes` to pipes and FIFOs is important. Assume that two clients send requests at about the same time to the server. The first client's request is the line

```
1234 /etc/inet/ntp.conf
```

and the second client's request is the line

```
9876 /etc/passwd
```

If we assume that each client issues one `write` function call for its request line, and that each line is less than or equal to `PIPE_BUF` (which is reasonable, since this limit is usually between 1024 and 5120 and since pathnames are often limited to 1024 bytes), then we are guaranteed that the data in the FIFO will be either

```
1234 /etc/inet/ntp.conf
9876 /etc/passwd
```

or

```
9876 /etc/passwd
1234 /etc/inet/ntp.conf
```

The data in the FIFO will *not* be something like

```
1234 /etc/inet9876 /etc/passwd
/ntp.conf
```

FIFOs and NFS

FIFOs are a form of IPC that can be used on a single host. Although FIFOs have names in the filesystem, they can be used only on local filesystems, and not on NFS-mounted filesystems.

```
solaris % mkfifo /nfs/bsdi/usr/rstevens/fifo.temp
mkfifo: I/O error
```

In this example, the filesystem `/nfs/bsdi/usr` is the `/usr` filesystem on the host `bsdi`.

Some systems (e.g., BSD/OS) do allow FIFOs to be created on an NFS-mounted filesystem, but data cannot be passed between the two systems through one of these FIFOs. In this scenario, the FIFO would be used only as a rendezvous point in the filesystem between clients and servers on the same host. A process on one host *cannot* send data to a process on another host through a FIFO, even though both processes may be able to open a FIFO that is accessible to both hosts through NFS.

4.9 Iterative versus Concurrent Servers

The server in our simple example from the preceding section is an *iterative server*. It iterates through the client requests, completely handling each client's request before proceeding to the next client. For example, if two clients each send a request to the server at about the same time—the first for a 10-megabyte file that takes 10 seconds (say) to send to the client, and the second for a 10-byte file—the second client must wait at least 10 seconds for the first client to be serviced.

The alternative is a *concurrent server*. The most common type of concurrent server under Unix is called a *one-child-per-client* server, and it has the server call `fork` to create a new child each time a client request arrives. The new child handles the client request to completion, and the multiprogramming features of Unix provide the concurrency of all the different processes. But there are other techniques that are discussed in detail in Chapter 27 of UNPv1:

- create a pool of children and service a new client with an idle child,
- create one thread per client, and
- create a pool of threads and service a new client with an idle thread.

Although the discussion in UNPv1 is for network servers, the same techniques apply to IPC servers whose clients are on the same host.

Denial-of-Service Attacks

We have already mentioned one problem with an iterative server—some clients must wait longer than expected because they are in line following other clients with longer requests—but another problem exists. Recall our shell example following Figure 4.24 and our discussion of how the server blocks in its call to `open` for the client FIFO if the client has not yet opened this FIFO (which did not happen until we executed our `cat`

command). This means that a malicious client could tie up the server by sending it a request line, but never opening its FIFO for reading. This is called a *denial-of-service* (DoS) attack. To avoid this, we must be careful when coding the iterative portion of any server, to note where the server might block, and for how long it might block. One way to handle the problem is to place a timeout on certain operations, but it is usually simpler to code the server as a concurrent server, instead of as an iterative server, in which case, this type of denial-of-service attack affects only one child, and not the main server. Even with a concurrent server, denial-of-service attacks can still occur: a malicious client could send lots of independent requests, causing the server to reach its limit of child processes, causing subsequent `fork`s to fail.

4.10 Streams and Messages

The examples shown so far, for pipes and FIFOs, have used the stream I/O model, which is natural for Unix. No record boundaries exist—reads and writes do not examine the data at all. A process that reads 100 bytes from a FIFO, for example, cannot tell whether the process that wrote the data into the FIFO did a single write of 100 bytes, five writes of 20 bytes, two writes of 50 bytes, or some other combination of writes that totals 100 bytes., One process could also write 55 bytes into the FIFO, followed by another process writing 45 bytes. The data is a *byte stream* with no interpretation done by the system. If any interpretation is desired, the writing process and the reading process must agree to it a priori and do it themselves.

Sometimes an application wants to impose some structure on the data being transferred. This can happen when the data consists of variable-length messages and the reader must know where the message boundaries are so that it knows when a single message has been read. The following three techniques are commonly used for this:

1. Special termination sequence in-band: many Unix applications use the newline character to delineate each message. The writing process appends a newline to each message, and the reading process reads one line at a time. This is what our client and server did in Figures 4.23 and 4.24 to separate the client requests. In general, this requires that any occurrence of the delimiter in the data must be escaped (that is, somehow flagged as data and not as a delimiter).

   Many Internet applications (FTP, SMTP, HTTP, NNTP) use the 2-character sequence of a carriage return followed by a linefeed (CR/LF) to delineate text records.

2. Explicit length: each record is preceded by its length. We will use this technique shortly. This technique is also used by Sun RPC when used with TCP. One advantage to this technique is that escaping a delimiter that appears in the data is unnecessary, because the receiver does not need to scan all the data, looking for the end of each record.

3. One record per connection: the application closes the connection to its peer (its TCP connection, in the case of a network application, or its IPC connection) to

indicate the end of a record. This requires a new connection for every record, but is used with HTTP 1.0.

The standard I/O library can also be used to read or write a pipe or FIFO. Since the only way to open a pipe is with the `pipe` function, which returns an open descriptor, the standard I/O function `fdopen` must be used to create a new standard I/O *stream* that is then associated with this open descriptor. Since a FIFO has a name, it can be opened using the standard I/O `fopen` function.

More structured messages can also be built, and this capability is provided by both Posix message queues and System V message queues. We will see that each message has a length and a priority (System V calls the latter a "type"). The length and priority are specified by the sender, and after the message is read, both are returned to the reader. Each message is a *record*, similar to UDP datagrams (UNPv1).

We can also add more structure to either a pipe or FIFO ourselves. We define a message in our `mesg.h` header, as shown in Figure 4.25.

pipemesg/mesg.h

```
1 #include    "unpipc.h"

2 /* Our own "messages" to use with pipes, FIFOs, and message queues. */

3         /* want sizeof(struct mymesg) <= PIPE_BUF */
4 #define MAXMESGDATA (PIPE_BUF - 2*sizeof(long))

5         /* length of mesg_len and mesg_type */
6 #define MESGHDRSIZE (sizeof(struct mymesg) - MAXMESGDATA)

7 struct mymesg {
8     long    mesg_len;           /* #bytes in mesg_data, can be 0 */
9     long    mesg_type;          /* message type, must be > 0 */
10    char    mesg_data[MAXMESGDATA];
11 };

12 ssize_t mesg_send(int, struct mymesg *);
13 void    Mesg_send(int, struct mymesg *);
14 ssize_t mesg_recv(int, struct mymesg *);
15 ssize_t Mesg_recv(int, struct mymesg *);
```

pipemesg/mesg.h

Figure 4.25 Our `mymesg` structure and related definitions.

Each message has a *mesg_type*, which we define as an integer whose value must be greater than 0. We ignore the type field for now, but return to it in Chapter 6, when we describe System V message queues. Each message also has a length, and we allow the length to be zero. What we are doing with the `mymesg` structure is to precede each message with its length, instead of using newlines to separate the messages. Earlier, we mentioned two benefits of this design: the receiver need not scan each received byte looking for the end of the message, and there is no need to escape the delimiter (a newline) if it appears in the message.

Figure 4.26 shows a picture of the `mymesg` structure, and how we use it with pipes, FIFOs, and System V message queues.

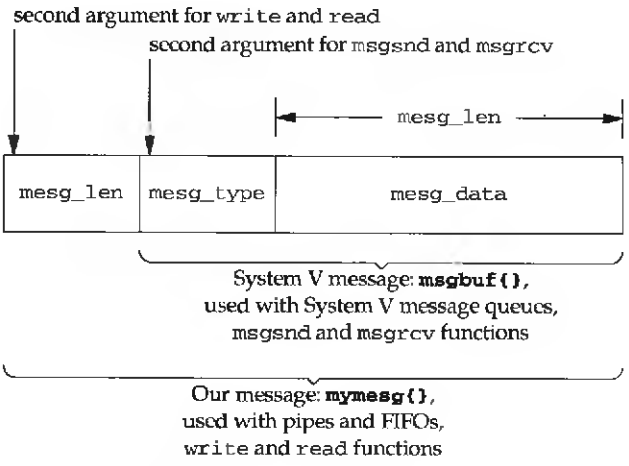

Figure 4.26 Our mymesg structure.

We define two functions to send and receive messages. Figure 4.27 shows our mesg_send function, and Figure 4.28 shows our mesg_recv function.

pipemesg/mesg_send.c

```
1 #include    "mesg.h"

2 ssize_t
3 mesg_send(int fd, struct mymesg *mptr)
4 {
5     return (write(fd, mptr, MESGHDRSIZE + mptr->mesg_len));
6 }
```

pipemesg/mesg_send.c

Figure 4.27 mesg_send function.

pipemesg/mesg_recv.c

```
1 #include    "mesg.h"

2 ssize_t
3 mesg_recv(int fd, struct mymesg *mptr)
4 {
5     size_t  len;
6     ssize_t n;

7         /* read message header first, to get len of data that follows */
8     if ( (n = Read(fd, mptr, MESGHDRSIZE)) == 0)
9         return (0);             /* end of file */
10    else if (n != MESGHDRSIZE)
11        err_quit("message header: expected %d, got %d", MESGHDRSIZE, n);

12    if ( (len = mptr->mesg_len) > 0)
13        if ( (n = Read(fd, mptr->mesg_data, len)) != len)
14            err_quit("message data: expected %d, got %d", len, n);
15    return (len);
16 }
```

pipemesg/mesg_recv.c

Figure 4.28 mesg_recv function.

It now takes two `read`s for each message, one to read the length, and another to read the actual message (if the length is greater than 0).

> Careful readers may note that `mesg_recv` checks for all possible errors and terminates if one occurs. Nevertheless, we still define a wrapper function named `Mesg_recv` and call it from our programs, for consistency.

We now change our client and server functions to use the `mesg_send` and `mesg_recv` functions. Figure 4.29 shows our client.

pipemesg/client.c

```
 1 #include    "mesg.h"

 2 void
 3 client(int readfd, int writefd)
 4 {
 5     size_t  len;
 6     ssize_t n;
 7     struct mymesg mesg;

 8         /* read pathname */
 9     Fgets(mesg.mesg_data, MAXMESGDATA, stdin);
10     len = strlen(mesg.mesg_data);
11     if (mesg.mesg_data[len - 1] == '\n')
12         len--;                  /* delete newline from fgets() */
13     mesg.mesg_len = len;
14     mesg.mesg_type = 1;

15         /* write pathname to IPC channel */
16     Mesg_send(writefd, &mesg);

17         /* read from IPC, write to standard output */
18     while ( (n = Mesg_recv(readfd, &mesg)) > 0)
19         Write(STDOUT_FILENO, mesg.mesg_data, n);
20 }
```

pipemesg/client.c

Figure 4.29 Our `client` function that uses messages.

Read pathname, send to server

8–16 The pathname is read from standard input and then sent to the server using `mesg_send`.

Read file's contents or error message from server

17–19 The client calls `mesg_recv` in a loop, reading everything that the server sends back. By convention, when `mesg_recv` returns a length of 0, this indicates the end of data from the server. We will see that the server includes the newline in each message that it sends to the client, so a blank line will have a message length of 1.

Figure 4.30 shows our server.

pipemesg/server.c

```
1 #include    "mesg.h"
2 void
3 server(int readfd, int writefd)
4 {
5     FILE   *fp;
6     ssize_t n;
7     struct mymesg mesg;
8         /* read pathname from IPC channel */
9     mesg.mesg_type = 1;
10    if ( (n = Mesg_recv(readfd, &mesg)) == 0)
11        err_quit("pathname missing");
12    mesg.mesg_data[n] = '\0';   /* null terminate pathname */
13    if ( (fp = fopen(mesg.mesg_data, "r")) == NULL) {
14            /* error: must tell client */
15        snprintf(mesg.mesg_data + n, sizeof(mesg.mesg_data) - n,
16                 ": can't open, %s\n", strerror(errno));
17        mesg.mesg_len = strlen(mesg.mesg_data);
18        Mesg_send(writefd, &mesg);
19    } else {
20            /* fopen succeeded: copy file to IPC channel */
21        while (Fgets(mesg.mesg_data, MAXMESGDATA, fp) != NULL) {
22            mesg.mesg_len = strlen(mesg.mesg_data);
23            Mesg_send(writefd, &mesg);
24        }
25        Fclose(fp);
26    }
27        /* send a 0-length message to signify the end */
28    mesg.mesg_len = 0;
29    Mesg_send(writefd, &mesg);
30 }
```

pipemesg/server.c

Figure 4.30 Our `server` function that uses messages.

Read pathname from IPC channel, open file

8–18 The pathname is read from the client. Although the assignment of 1 to `mesg_type` appears useless (it is overwritten by `mesg_recv` in Figure 4.28), we call this same function when using System V message queues (Figure 6.10), in which case, this assignment is needed (e.g., Figure 6.13). The standard I/O function `fopen` opens the file, which differs from Figure 4.10, where we called the Unix I/O function `open` to obtain a descriptor for the file. The reason we call the standard I/O library here is to call `fgets` to read the file one line at a time, and then send each line to the client as a message.

Copy file to client

19–26 If the call to `fopen` succeeds, the file is read using `fgets` and sent to the client, one line per message. A message with a length of 0 indicates the end of the file.

When using either pipes or FIFOs, we could also close the IPC channel to notify the peer that the end of the input file was encountered. We send back a message with a length of 0, however, because we will encounter other types of IPC that do not have the concept of an end-of-file.

The `main` functions that call our `client` and `server` functions do not change at all. We can use either the pipe version (Figure 4.8) or the FIFO version (Figure 4.16).

4.11 Pipe and FIFO Limits

The only system-imposed limits on pipes and FIFOs are

`OPEN_MAX` the maximum number of descriptors open at any time by a process (Posix requires that this be at least 16), and

`PIPE_BUF` the maximum amount of data that can be written to a pipe or FIFO atomically (we described this in Section 4.7; Posix requires that this be at least 512).

The value of `OPEN_MAX` can be queried by calling the `sysconf` function, as we show shortly. It can normally be changed from the shell by executing the `ulimit` command (Bourne shell and KornShell, as we show shortly) or the `limit` command (C shell). It can also be changed from a process by calling the `setrlimit` function (described in detail in Section 7.11 of APUE).

The value of `PIPE_BUF` is often defined in the `<limits.h>` header, but it is considered a *pathname variable* by Posix. This means that its value can differ, depending on the pathname that is specified (for a FIFO, since pipes do not have names), because different pathnames can end up on different filesystems, and these filesystems might have different characteristics. The value can therefore be obtained at run time by calling either `pathconf` or `fpathconf`. Figure 4.31 shows an example that prints these two limits.

pipe/pipeconf.c

```
1 #include    "unpipc.h"

2 int
3 main(int argc, char **argv)
4 {
5     if (argc != 2)
6         err_quit("usage: pipeconf <pathname>");

7     printf("PIPE_BUF = %ld, OPEN_MAX = %ld\n",
8            Pathconf(argv[1], _PC_PIPE_BUF), Sysconf(_SC_OPEN_MAX));
9     exit(0);
10 }
```

pipe/pipeconf.c

Figure 4.31 Determine values of `PIPE_BUF` and `OPEN_MAX` at run time.

Here are some examples, specifying different filesystems:

```
solaris % pipeconf /                    Solaris 2.6 default values
PIPE_BUF = 5120, OPEN_MAX = 64
solaris % pipeconf /home
PIPE_BUF = 5120, OPEN_MAX = 64
solaris % pipeconf /tmp
PIPE_BUF = 5120, OPEN_MAX = 64

alpha % pipeconf /                      Digital Unix 4.0B default values
PIPE_BUF = 4096, OPEN_MAX = 4096
alpha % pipeconf /usr
PIPE_BUF = 4096, OPEN_MAX = 4096
```

We now show how to change the value of OPEN_MAX under Solaris, using the Korn-Shell.

```
solaris % ulimit -nS                    display max # descriptors, soft limit
64
solaris % ulimit -nH                    display max # descriptors, hard limit
1024
solaris % ulimit -nS 512                set soft limit to 512
solaris % pipeconf /                    verify that change has occurred
PIPE_BUF = 5120, OPEN_MAX = 512
```

Although the value of PIPE_BUF can change for a FIFO, depending on the underlying filesystem in which the pathname is stored, this should be extremely rare.

Chapter 2 of APUE describes the fpathconf, pathconf, and sysconf functions, which provide run-time information on certain kernel limits. Posix.1 defines 12 constants that begin with _PC_ and 52 that begin with _SC_. Digital Unix 4.0B and Solaris 2.6 both extend the latter, defining about 100 run-time constants that can be queried with sysconf.

The getconf command is defined by Posix.2, and it prints the value of most of these implementation limits. For example

```
alpha % getconf OPEN_MAX
4096
alpha % getconf PIPE_BUF /
4096
```

4.12 Summary

Pipes and FIFOs are fundamental building blocks for many applications. Pipes are commonly used with the shells, but also used from within programs, often to pass information from a child back to a parent. Some of the code involved in using a pipe (pipe, fork, close, exec, and waitpid) can be avoided by using popen and pclose, which handle all the details and invoke a shell.

FIFOs are similar to pipes, but are created by `mkfifo` and then opened by `open`. We must be careful when opening a FIFO, because numerous rules (Figure 4.21) govern whether an `open` blocks or not.

Using pipes and FIFOs, we looked at some client–server designs: one server with multiple clients, and iterative versus concurrent servers. An iterative server handles one client request at a time, in a serial fashion, and these types of servers are normally open to denial-of-service attacks. A concurrent server has another process or thread handle each client request.

One characteristic of pipes and FIFOs is that their data is a byte stream, similar to a TCP connection. Any delineation of this byte stream into records is left to the application. We will see in the next two chapters that message queues provide record boundaries, similar to UDP datagrams.

Exercises

4.1 In the transition from Figure 4.3 to Figure 4.4, what could happen if the child did not `close(fd[1])`?

4.2 In describing `mkfifo` in Section 4.6, we said that to open an existing FIFO or create a new FIFO if it does not already exist, call `mkfifo`, check for an error of `EEXIST`, and if this occurs, call `open`. What can happen if the logic is changed, calling `open` first and then `mkfifo` if the FIFO does not exist?

4.3 What happens in the call to `popen` in Figure 4.15 if the shell encounters an error?

4.4 Remove the `open` of the server's FIFO in Figure 4.23 and verify that this causes the server to terminate when no more clients exist.

4.5 In Figure 4.23, we noted that when the server starts, it blocks in its first call to `open` until the first client `open`s this FIFO for writing. How can we get around this, causing both `open`s to return immediately, and block instead in the first call to `readline`?

4.6 What happens to the client in Figure 4.24 if it swaps the order of its two calls to `open`?

4.7 Why is a signal generated for the writer of a pipe or FIFO after the reader disappears, but not for the reader of a pipe or FIFO after its writer disappears?

4.8 Write a small test program to determine whether `fstat` returns the number of bytes of data currently in a FIFO as the `st_size` member of the `stat` structure.

4.9 Write a small test program to determine what `select` returns when you select for writability on a pipe descriptor whose read end has been closed.

5

Posix Message Queues

5.1 Introduction

A message queue can be thought of as a linked list of messages. Threads with adequate permission can put messages onto the queue, and threads with adequate permission can remove messages from the queue. Each message is a *record* (recall our discussion of streams versus messages in Section 4.10), and each message is assigned a priority by the sender. No requirement exists that someone be waiting for a message to arrive on a queue before some process writes a message to that queue. This is in contrast to both pipes and FIFOs, for which it having a writer makes no sense unless a reader also exists.

A process can write some messages to a queue, terminate, and have the messages read by another process at a later time. We say that message queues have *kernel persistence* (Section 1.3). This differs from pipes and FIFOs. We said in Chapter 4 that any data remaining in a pipe or FIFO when the last close of the pipe or FIFO takes place, is discarded.

This chapter looks at Posix message queues and Chapter 6 looks at System V message queues. Many similarities exist between the two sets of functions, with the main differences being:

- A read on a Posix message queue always returns the oldest message of the highest priority, whereas a read on a System V message queue can return a message of any desired priority.
- Posix message queues allow the generation of a signal or the initiation of a thread when a message is placed onto an empty queue, whereas nothing similar is provided by System V message queues.

Every message on a queue has the following attributes:

- an unsigned integer priority (Posix) or a long integer type (System V),
- the length of the data portion of the message (which can be 0), and
- the data itself (if the length is greater than 0).

Notice that these characteristics differ from pipes and FIFOs. The latter two are byte streams with no message boundaries, and no type associated with each message. We discussed this in Section 4.10 and added our own message interface to pipes and FIFOs.

Figure 5.1 shows one possible arrangement of a message queue.

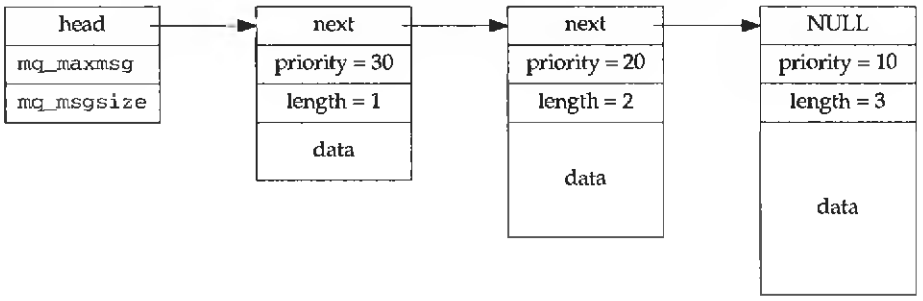

Figure 5.1 Possible arrangement of a Posix message queue containing three messages.

We are assuming a linked list, and the head of the list contains the two attributes of the queue: the maximum number of messages allowed on the queue, and the maximum size of a message. We say more about these attributes in Section 5.3.

In this chapter, we use a technique that we use in later chapters when looking at message queues, semaphores, and shared memory. Since all of these IPC objects have at least kernel persistence (recall Section 1.3), we can write small programs that use these techniques, to let us experiment with them and learn more about their operation. For example, we can write a program that creates a Posix message queue, write another program that adds a message to a Posix message queue, and write another that reads from one of these queues. By writing messages with different priorities, we can see how these messages are returned by the `mq_receive` function.

5.2 `mq_open`, `mq_close`, and `mq_unlink` Functions

The `mq_open` function creates a new message queue or opens an existing message queue.

```
#include <mqueue.h>

mqd_t mq_open(const char *name, int oflag, ...
              /* mode_t mode, struct mq_attr *attr */ );
```

Returns: message queue descriptor if OK, –1 on error